Texts in Computer Science

Stefano Crespi Reghizzi
Luca Breveglieri
Angelo Morzenti



Formal Languages and Compilation

Third Edition

Springer

Texts in Computer Science

Series Editors

David Gries, Department of Computer Science, Cornell University, Ithaca, NY, USA
Orit Hazzan, Faculty of Education in Technology and Science, Technion—Israel Institute of Technology, Haifa, Israel

More information about this series at http://www.springer.com/series/3191

Stefano Crespi Reghizzi •
Luca Breveglieri • Angelo Morzenti

Formal Languages and Compilation

Third Edition

Springer

Stefano Crespi Reghizzi
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Milan, Italy

Luca Breveglieri
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Milan, Italy

Angelo Morzenti
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Milan, Italy

ISSN 1868-0941 ISSN 1868-095X (electronic)
Texts in Computer Science
ISBN 978-3-030-04878-5 ISBN 978-3-030-04879-2 (eBook)
https://doi.org/10.1007/978-3-030-04879-2

Library of Congress Control Number: 2018965427

1st & 2nd editions: © Springer-Verlag London 2009, 2013
3rd edition: © Springer Nature Switzerland AG 2019
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

This third edition updates, enriches, and revises the 2013 printing [1], without dulling the original structure of the book. The selection of materials and the presentation style reflect many years of teaching compiler courses and of doing research on formal language theory and formal methods, on compiler and language design, and to a lesser extent on natural language processing. In the turmoil of information technology developments, the subject of the book has kept the same fundamental principles since half a century, yet preserving its conceptual importance and practical relevance.

This state of affairs of a topic, which is central to computer science and information processing and is based on established principles, might lead some people to believe that the corresponding textbooks are by now consolidated, much as the classical books on calculus or physics. In reality, this is not the case: there exist fine classical books on the mathematical aspects of language and automata theory, but, for what concerns application to compiling and data representation, the best books are sort of encyclopedias of algorithms, design methods, and practical tricks used in compiler design. Indeed, a compiler is a microcosm, and it features many different aspects ranging from algorithmic wisdom to computer hardware, system software, and network interfaces. As a consequence, the textbooks have grown in size and compete with respect to their coverage of the latest developments in programming languages, processor and network architectures and clever mappings from the former to the latter.

To put things into order, it is better to separate such complex topics into two parts, that we may call *basic* or fundamental, and *advanced* or technological, which to some extent correspond to the two subsystems that make a compiler: the user-language level specific front-end and the machine-level specific back-end. The basic part is the subject of this book; it covers the principles and algorithms to be used for defining the syntax of languages and for implementing simple translators. It does not include: the specialized know-how needed for various classes of programming languages (procedural, functional, object oriented, etc.), the computer architecture related aspects, and the optimization methods used to improve the performance of both the compilation process and the executable code produced.

In other textbooks on compilers, the bias toward practical aspects has reduced the attention to fundamental concepts. This has prevented their authors from taking

advantage of the improvements and simplifications made possible by decades of both extensive use and theoretical polishing, to avoid the irritating variants and repetitions that are found in the historical fundamental papers. Moving from these premises, we decided to present, in a simple minimalist way, the principles and methods currently used in designing syntax-directed applications such as parsing and regular expression matching. Thus, we have substantially improved the standard presentation of parsing algorithms for both regular expressions and grammars, by unifying the concepts and notations used in various approaches, and by extending method coverage with a reduced definitional apparatus. An example that expert readers should appreciate is the unification of top-down and bottom-up parsing algorithms within a new rigorous yet practical framework.

In this way, the reduced effort and space needed to cover classical concepts has made room for new advanced methods typically not present in similar textbooks. First, we have introduced in this edition a new efficient algorithm for string matching using regular expressions, which are increasingly important in such applications as Web browsing and data filtering. Second, our presentation of parsing algorithms focuses on the more expressive extended *BNF* grammars, which are the de facto standard in language reference manuals. Third, we present a parallel parsing algorithm that takes advantage of the many processing units of modern multi-core devices to speed up the syntax analysis of large files. At last, we mention the addition in this edition of the recent formal models of input-driven automata and languages (also known as visibly pushdown model), which suits the mark-up languages (such as *XML* and *JSON*) and has been developed also in other domains to formally verify critical systems.

The presentation is illustrated by many small yet realistic examples, to ease the understanding of the theory and the transfer to application. Theoretical models of automata, transducers, and formal grammars are extensively used, whenever practical motivations exist. Algorithms are described in a pseudo-code to avoid the disturbing details of a programming language, yet they are straightforward to convert into executable procedures, and for some, we have made available their code on the Web.

The book is not restricted to syntax. The study of translations, semantic functions (attribute grammars), and static program analysis by data flow equations enlarges the understanding of the compilation process and covers the essentials of a syntax-directed translator.

A large collection of problems and solutions is available at the authors' course site.

This book should be welcome by those willing to teach or learn the essential concepts of syntax-directed compilation, without needing to rely on software tools and implementations. We believe that learning by doing is not always the best approach and that an overcommitment to practical work may sometimes obscure the conceptual foundations. In the case of formal languages, the elegance and essentiality of the underlying theory allows students to acquire the fundamental paradigms of language structures, to avoid pitfalls such as ambiguity, and to adequately map structure to meaning. In this field, many if not all relevant algorithms

are simple enough to be practiced by paper and pencil. Of course, students should be encouraged to enroll in a hands-on laboratory and to experiment syntax-directed tools (like *flex* and *bison*) on realistic cases.

The authors thank the colleagues Alessandro Barenghi and Marcello Bersani, who have taught compilers to computer engineering students by using this book.

The first author remembers the late Antonio Grasselli, the computer science pioneer who first fascinated him with the subject of this book, combining linguistic, mathematical, and technological aspects.

Milan, Italy
April 2019

Stefano Crespi Reghizzi
Luca Breveglieri
Angelo Morzenti

Reference

1. Crespi Reghizzi S, Breveglieri L, Morzenti A (2013) Formal languages and compilation, 2nd edn. Springer

Contents

1 Introduction 1
1.1 Intended Scope and Readership 1
1.2 Compiler Parts and Corresponding Concepts 2

2 Syntax 5
2.1 Introduction 5
2.1.1 Artificial and Formal Languages 5
2.1.2 Language Types 6
2.1.3 Chapter Outline 7
2.2 Formal Language Theory 8
2.2.1 Alphabet and Language 8
2.2.2 Language Operations 11
2.2.3 Set Operations 13
2.2.4 Star and Cross 14
2.2.5 Language Quotient 17
2.3 Regular Expressions and Languages 17
2.3.1 Definition of Regular Expression 18
2.3.2 Derivation and Language 20
2.3.3 Other Operators 24
2.3.4 Closure Properties of Family *REG* 25
2.4 Linguistic Abstraction: Abstract and Concrete Lists 26
2.4.1 Lists with Separators and Opening/Closing Marks 27
2.4.2 Language Substitution 28
2.4.3 Hierarchical or Precedence Lists 29
2.5 Context-Free Generative Grammars 31
2.5.1 Limits of Regular Languages 31
2.5.2 Introduction to Context-Free Grammars 32
2.5.3 Conventional Grammar Representations 34
2.5.4 Derivation and Language Generation 37
2.5.5 Erroneous Grammars and Useless Rules 39
2.5.6 Recursion and Language Infinity 41
2.5.7 Syntax Tree and Canonical Derivation 43
2.5.8 Parenthesis Languages 47

2.5.9 Regular Composition of Context-Free Languages 51
2.5.10 Ambiguity in Grammars 52
2.5.11 Catalogue of Ambiguous Forms and Remedies 54
2.5.12 Weak and Structural Equivalence 62
2.5.13 Grammar Transformations and Normal Forms 65
2.6 Grammars of Regular Languages 82
2.6.1 From Regular Expression to Grammar 82
2.6.2 Linear and Unilinear Grammars 84
2.6.3 Linear Language Equations 87
2.7 Comparison of Regular and Context-Free Languages 89
2.7.1 Limits of Context-Free Languages 93
2.7.2 Closure Properties of Families *REG* and *CF* 95
2.7.3 Alphabetic Transformations 97
2.7.4 Grammars with Regular Expressions 100
2.8 More General Grammars and Language Families 104
2.8.1 Chomsky Classification of Grammars 105
2.8.2 Further Developments and Final Remarks 109
References 112

3 Finite Automata as Regular Language Recognizers 115
3.1 Introduction 115
3.2 Recognition Algorithms and Automata 116
3.2.1 A General Automaton 117
3.3 Introduction to Finite Automata 120
3.4 Deterministic Finite Automata 122
3.4.1 Error State and Total Automaton 123
3.4.2 Clean Automaton 124
3.4.3 Minimal Automaton 125
3.4.4 From Automaton to Grammar 129
3.5 Nondeterministic Automata 130
3.5.1 Motivation of Nondeterminism 131
3.5.2 Nondeterministic Recognizers 133
3.5.3 Automata with Spontaneous Moves 135
3.5.4 Correspondence Between Automata and Grammars 136
3.5.5 Ambiguity of Automata 137
3.5.6 Left-Linear Grammars and Automata 138
3.6 From Automaton to Regular Expression: *BMC* Method 139
3.7 Elimination of Nondeterminism 142
3.7.1 Elimination of Spontaneous Moves 142
3.7.2 Construction of Accessible Subsets 144
3.8 From Regular Expression to Recognizer 147
3.8.1 Thompson Structural Method 148
3.8.2 Berry–Sethi Method 150

3.9 String Matching of Ambiguous Regular Expressions . . . 165
3.9.1 Trees for Ambiguous r.e. and Their Phrases . . . 165
3.9.2 Local Recognizer of Linearized Syntax Trees . . . 172
3.9.3 The Berry–Sethi Parser . . . 172
3.10 Expressions with Complement and Intersection . . . 188
3.10.1 Product of Automata . . . 190
3.11 Summary of the Relations Between Regular Expressions, Grammars and Automata . . . 195
References . . . 196

4 Pushdown Automata and Parsing . . . 199
4.1 Introduction . . . 199
4.2 Pushdown Automaton . . . 200
4.2.1 From Grammar to Pushdown Automaton . . . 201
4.2.2 Definition of Pushdown Automaton . . . 205
4.2.3 One Family for Context-Free Languages and Pushdown Automata . . . 209
4.2.4 Intersection of Regular and Context-Free Languages . . . 213
4.3 Deterministic Pushdown Automata and Languages . . . 215
4.3.1 Closure Properties of Deterministic Languages . . . 216
4.3.2 Nondeterministic Languages . . . 218
4.3.3 Determinism and Language Unambiguity . . . 220
4.3.4 Subclasses of Deterministic Pushdown Automata and Languages . . . 221
4.4 Syntax Analysis: Top-Down and Bottom-Up Constructions . . . 230
4.5 Grammar as Network of Finite Automata . . . 233
4.5.1 Syntax Charts . . . 237
4.5.2 Derivation for Machine Nets . . . 239
4.5.3 Initial and Look-Ahead Characters . . . 241
4.6 Bottom-Up Deterministic Analysis . . . 244
4.6.1 From Finite Recognizers to Bottom-Up Parser . . . 245
4.6.2 Construction of *ELR* (1) Parsers . . . 250
4.6.3 *ELR* (1) Condition . . . 253
4.6.4 Simplifications for *BNF* Grammars . . . 267
4.6.5 Parser Implementation Using a Vector Stack . . . 271
4.6.6 Lengthening the Look-Ahead . . . 275
4.7 Deterministic Top-Down Parsing . . . 282
4.7.1 *ELL* (1) Condition . . . 286
4.7.2 Step-by-Step Derivation of *ELL* (1) Parsers . . . 288
4.7.3 Direct Construction of Top-Down Predictive Parsers . . . 305
4.7.4 A Graphical Method for Computing Guide Sets . . . 314
4.7.5 Increasing Look-Ahead in Top-Down Parsers . . . 322

4.8 Operator-Precedence Grammars and Parallel Parsing 325
4.8.1 Floyd Operator-Precedence Grammars and Parsers 326
4.8.2 Sequential Operator-Precedence Parser 330
4.8.3 Comparisons and Closure Properties 333
4.8.4 Parallel Parsing Algorithm 339
4.9 Deterministic Language Families: A Comparison 346
4.10 Discussion of Parsing Methods 350
4.11 A General Parsing Algorithm 352
4.11.1 Introductory Presentation 353
4.11.2 Earley Algorithm 357
4.11.3 Syntax Tree Construction 366
4.12 Managing Syntactic Errors and Changes 373
4.12.1 Errors 373
4.12.2 Incremental Parsing 379
References 379

5 Translation Semantics and Static Analysis 383
5.1 Introduction 383
5.2 Translation Relation and Function 386
5.3 Transliteration 388
5.4 Purely Syntactic Translation 389
5.4.1 Infix and Polish Notation 391
5.4.2 Ambiguity of Source Grammar and Translation 395
5.4.3 Translation Grammar and Pushdown Transducer 397
5.4.4 Syntax Analysis with Online Translation 402
5.4.5 Top-Down Deterministic Translation by Recursive Procedures 403
5.4.6 Bottom-Up Deterministic Translation 406
5.5 Regular Translation 414
5.5.1 Two-Input Automaton 416
5.5.2 Translation Function and Finite Transducer 421
5.5.3 Closure Properties of Translation 426
5.6 Semantic Translation 427
5.6.1 Attribute Grammars 429
5.6.2 Left and Right Attributes 431
5.6.3 Definition of Attribute Grammar 435
5.6.4 Dependence Graph and Attribute Evaluation 437
5.6.5 One-Sweep Semantic Evaluation 442
5.6.6 Other Evaluation Methods 446
5.6.7 Combined Syntax and Semantic Analysis 447
5.6.8 Typical Applications of Attribute Grammar 457

5.7 Static Program Analysis 467
5.7.1 A Program as an Automaton 468
5.7.2 Liveness Intervals of a Variable 471
5.7.3 Reaching Definition 477
References 486

Index 487



Introduction

1

1.1 Intended Scope and Readership

The information technology revolution was made possible by the invention of electronic digital machines, yet without programming languages their use would have been restricted to the few people able to write binary machine code. A programming language as a text contains features coming both from human languages and from mathematical logic. The translation from a programming language to machine code is known as *compilation*.[1] Language compilation is a very complex process, which would be impossible to master without systematic design methods. Such methods and their theoretical foundations are the argument of this book. They make up a consistent and largely consolidated body of concepts and algorithms, which are applied not just in compilers, but also in other fields. Automata theory is pervasive in all the branches of informatics, to model situations or phenomena classifiable as time and space discrete systems. Formal grammars, on the other hand, are originated in linguistic research, and together with automata, they are widely applied in document processing and in data representation and searching, in particular for the Web.

Coming to the prerequisites, the reader should have a good background in programming, though a detailed knowledge of a specific programming language is not required, because our presentation of algorithms relies on self-explanatory pseudocode. The reader is expected to be familiar with basic mathematical theories and notations, namely from set theory, algebra and logic. The above prerequisites are typically met by computer science/engineering or mathematics students with a university education of two or more years.

[1] This term may sound strange; it originates in the early approach to the compilation of correspondence tables between a command in the language and a series of machine operations.

© Springer Nature Switzerland AG 2019

S. Crespi Reghizzi et al., *Formal Languages and Compilation*,
Texts in Computer Science, https://doi.org/10.1007/978-3-030-04879-2_1

The selection of topics and the presentation based on rigorous definitions and algorithms illustrated by many motivating examples qualifies the book for a university course, aiming to expose students to the importance of good theories and of efficient algorithms for designing effective systems. In our experience some fifty hours of lecture suffice to cover the entire book.

The authors' long experience in teaching the subject to different audiences brings out the importance of combining theoretical concepts and examples. Moreover, it is advisable that the students take advantage of well-known and documented software tools (such as the classical *Flex* and *Bison* generators of lexical and syntactic analyzers), to implement and experiment the main algorithm on realistic case studies.

With regard to the reach and limits, the book covers the essential concepts and methods needed to design simple translators based on the very common syntax-directed paradigm. Such systems analyze into syntactical units a source text, written in a formally specified language, and translate it into a different target representation. The translation function is nicely formulated as the composition of simpler functions keyed to the syntactical units of the source text.

It goes without saying that a real compiler for a programming language includes other technological aspects and know-how, in particular related to processor and computer architecture, which are not covered. Such know-how is essential for automatically translating a program into machine instructions and for transforming a program in order to make the best use of the computational resources of a computer. The study of methods for program transformation and optimization is a more advanced topic, which follows the present introduction to compiler methods. The next section outlines the contents of the book.

1.2 Compiler Parts and Corresponding Concepts

There are two external interfaces to a compiler: the source language to be analyzed and translated and the target language produced by the translator.

Chapter 2 describes the so-called syntactic methods that are universally adopted in order to provide a rigorous definition of the texts (or character strings) written in the source language. The methods to be presented are regular expressions and context-free (*BNF*) grammars. To avoid the pitfall of ambiguous definitions, we examine the relevant causes of ambiguity in the regular expressions and in the grammars. Useful transformations from regular expressions to grammars and between grammars of different forms are covered, which are needed in the following chapters. In particular we introduce the combination of regular expressions and grammars, which is the central model for the parsing algorithms. Chapter 2 ends with a critical hint to more powerful, but rarely applied models: the Chomsky context-sensitive type grammars and the recently proposed conjunctive grammars.

The first task of a compiler is to check the correctness of the source text, that is, whether the text agrees with the syntactic definition of the source language. In order to perform such a check, the algorithm scans the source text character by character,

and at the end it rejects or accepts the input depending on the result of the analysis. By a minimalist abstract approach, such recognition algorithms are conveniently described as mathematical machines or automata, in the tradition of the well-known Turing machine. Different automaton types correspond to the various families of formal languages, and the types practically relevant for compilation are considered in the following two chapters.

Chapter 3 covers finite automata, which are machines with a finite random access memory. They are the recognizers of the languages defined by regular expressions. Within compilation, they are used for lexical analysis or scanning, to extract from the source text the keywords, numbers, identifiers and in general the pieces of text that encode the lexical units of the language, called lexemes or tokens.

The basic models of finite automata, deterministic and nondeterministic, real-time and with spontaneous moves are presented along with the transformations between different models. The algorithms for converting a finite automaton into a regular expression and conversely are well covered. The other main topics of Chap. 3 are: the string matching problem for ambiguous regular expressions, and the use of Boolean operations, intersection and complement, in the regular expressions.

Chapter 4 is devoted to the recognition problem for the languages defined by context-free grammars, which is the backbone of syntax-directed translators and justifies the salient length of the chapter.

The chapter starts with the model of nondeterministic pushdown automata and its direct correspondence to context-free grammars. The deterministic version of the model follows, which defines the leading family of deterministic languages. As a special case, the input-driven or visibly pushdown automata have been included, because of their relevance for markup languages such as *XML* and *JSON*. A rich choice of parsing algorithms is presented. First, there is a unified modern presentation of the deterministic parsers, both bottom-up (or *LR* (1)) and top-down (or *LL* (1)), for context-free grammars extended with regular languages. The third deterministic parser is the operator-precedence method, which is here presented in the parallel version suitable for multi-core computers. The last parsing method by Earley is the most general and accepts any context-free language. Chapter 4 includes comparisons of the different language families suitable for each parsing method, a discussion of grammar transformations, and the basic concepts on error handling by parsers.

The ultimate job of a compiler is to translate a source text into another language. The compiler module responsible for finalizing the verification of the source language rules and for producing the translation is called the semantic analyzer. It operates on the structural representation, the syntax tree, produced by the parser.

The formal translation models and the methods used to implement semantic analyzers are illustrated in Chap. 5. Actually, we describe two kinds of transformations. First, the pure syntactic translations are modeled by transducers that are either finite automata or pushdown automata extended with an output, and that compute a string. Second, the semantic translations are performed by functions or methods that operate on the syntax tree of the source text and may produce as output not just a string, but any sort of data values. Semantic translations are specified by a practical semi-formal extension to context-free grammars, which is called attribute grammar. This

approach, by combining the accuracy of formal syntax and the flexibility of programming, conveniently expresses the typical analysis and transformation of syntax trees that is required by compilers.

To give a concrete idea of compilation, typical simple examples are included: the type consistency check between the variables declared and used in a programming language, the translation of high-level statements into machine instructions and semantics-directed parsing.

For sure, compilers do much more than syntax-directed translation. Static program analysis is an important example: it consists in examining a program to determine, ahead of execution, some properties, or to detect errors not detected by syntactic and semantic analysis. The purpose is to improve the robustness, reliability and efficiency of the program. An example of error detection is the identification of uninitialized variables used in the arithmetic expressions. For code improvement, an example is the elimination of useless assignment statements.

Chapter 5 terminates with an introduction to the static analysis of programs modeled by their control-flow graph, viewed as a finite automaton. Several interesting problems can be formalized and statically analyzed by a common approach based on flow equations and their solution by iterative approximations converging to the least fixed point.

Last, the book comes with an up-to-date essential bibliography at the end of each chapter and is accompanied by a detailed index.



Syntax

2

2.1 Introduction

2.1.1 Artificial and Formal Languages

Many centuries after the spontaneous emergence of natural language for human communication, mankind has purposively constructed other communication systems and languages, to be called artificial, intended for very specific tasks. A few artificial languages, like the logical propositions of Aristotle or the music sheet notation of Guittone d'Arezzo, are very ancient, but their number has exploded with the invention of computers. Many artificial languages are intended for man–machine communication, to instruct a programmable machine to do some task: to perform a computation, to prepare a document, to search a database or the Web, to control a robot, and so on. Other languages serve as interfaces between devices or between software applications, e.g., *PDF* is a language that many document and picture processors read and write.

Any designed language is artificial by definition, but not all artificial languages are formalized: thus, a programming language like Java is formalized, but Esperanto, although designed by man, is not so.

But how should we classify the language of genes or *DNA*, which deals with natural biological systems though it has been invented by scientists? The structures of molecular biology are three-dimensional, yet viewing them as *RNA* sequences has opened the way to the methods of formal languages.[1]

For a language to be formalized (or formal), the form of its sentences (or syntax) and their meaning (or semantics) must be precisely and algorithmically defined. In

[1] For a discussion of the role of linguistic methods in molecular biology, see Searls [1].

© Springer Nature Switzerland AG 2019

S. Crespi Reghizzi et al., *Formal Languages and Compilation*,
Texts in Computer Science, https://doi.org/10.1007/978-3-030-04879-2_2

other words, it should be possible for a computer to check that the sentences are grammatically correct, and to determine their meaning.

Meaning, however, is a difficult and controversial notion. For our purposes, the meaning of a sentence can be taken to be the translation into another language, which is known to the computer or to a well-established software system. For instance, the meaning of a Java program is its translation into the machine language of the computer executing the program.

In this book, the term *formal language* is used in a narrower sense, which excludes semantics. In the field of syntax, a formal language is a mathematical structure, defined on top of an alphabet, by means of certain axiomatic rules (formal grammar) or by using abstract machines such as the famous one due to A. Turing. The notions and methods of formal language are analogous to those used in number theory and in logic.

Thus, formal language theory is primarily concerned with the form or syntax of sentences, not with meaning, at least not directly. A string (or text) is either valid or illegal, that is, it either belongs to the formal language or it does not. Such a theory makes a first important step toward the ultimate goal: the study of language translation and meaning, which will however require additional methods.

2.1.2 Language Types

In this book, a language is a one-dimensional communication medium, made by sequences of symbolic elements of an alphabet, called terminal characters. Actually, people often refer to language as other non-textual communication media, which are more or less formalized by means of rules. Thus, iconic languages focus on road traffic signs or video display icons. Musical language is concerned with sounds, rhythm, and harmony. Architects and industrial designers of buildings and things are interested in their spatial relations, which they describe as the language of design. Early child drawings are often considered as sentences of a pictorial language, which can be partially formalized in accordance with developmental psychology theories. Certainly, the formal approach to syntax to be developed has some interest for nontextual languages too, which however are out of scope for this book.

Within computer science, the term language applies, as said, to a text made by a set of characters orderly written from, say, left to right. In addition, the term is used to refer to other discrete structures, such as graphs, trees, or arrays of pixels describing a digital picture. Formal language theories have been proposed and used more or less successfully also for such nontextual languages.[2]

Reverting to the main stream of textual languages, a frequent request directed to the specialist is to define and specify an artificial language. The specification may have several uses: as a language reference manual for future users, as an official

[2] Just three examples and their references: graph grammars and languages [2], tree languages [3,4], and picture (or two-dimensional) languages [5,6].

standard definition, or as a contractual document for compiler designers to ensure consistency of specification and implementation.

It is not an easy task to write a complete and rigorous definition of a language. Of course, the exhaustive approach, to list all the possible sentences or phrases, is unfeasible because the possibilities are infinitely many, since the length of the sentences is usually unbounded. As a native language speaker, a programmer is not constrained by any strict limit on the length of the phrases he writes. The ensuing problem to represent an infinite number of cases by a finite description can be addressed by an enumeration procedure, as in logic. When executed, the procedure generates longer and longer sentences, in an unending process if the language to be modeled is not finite.

This chapter presents a simple and established manner to express the rules of the enumeration procedure in the form of rules of a generative grammar (or syntax).

2.1.3 Chapter Outline

The chapter starts with the basic components of language theory: alphabet, string, and operations, such as concatenation and repetition, on strings and sets of strings.

The first model presented are the regular expressions, which define the family of regular languages. Then the lists are introduced as a fundamental and pervasive syntax structure in all the kinds of languages. From the exemplification of list variants, the idea of linguistic abstraction grows out naturally. This is a powerful reasoning tool to reduce the varieties of existing languages to a few paradigms.

After discussing the limits of regular languages, the chapter moves to context-free grammars. Following the basic definitions, the presentation focuses on structural properties, namely equivalence, ambiguity, and recursion. Exemplification continues with important linguistic paradigms such as hierarchical lists, parenthesized structures, polish notations, and operator precedence expressions. Their combination produces the variety of forms to be found in artificial languages.

Then the classification of some common forms of ambiguity and corresponding remedies is offered as a practical guide for grammar designers.

Various rule transformations (normal forms) are introduced, which should familiarize the reader with the modifications, to adjust a grammar without affecting the language, often needed for the analysis algorithms studied in Chap. 4.

Returning to regular languages from the grammar perspective, the chapter evidences the greater descriptive capacity of context-free grammars. The comparison of regular and context-free languages continues by considering the operations that may cause a language to exit or remain in one or the other family. Alphabetical transformations anticipate the operations studied in Chap. 5 as translations. A discussion about the unavoidable regularities found in very long strings completes the theoretical picture.

The last section mentions the Chomsky classification of grammar types and exemplifies context-sensitive grammars, stressing the difficulty of such rarely used model and discussing some recent attempts at more expressive grammar models.

2.2 Formal Language Theory

Formal language theory starts from the elementary notions of alphabet, string operations, and aggregate operations on sets of strings. By such operations, complex languages can be obtained starting from simpler ones.

2.2.1 Alphabet and Language

An *alphabet* is a finite set of elements called *terminal symbols* or *characters*. Let $\Sigma = \{ a_1, a_2, \ldots, a_k \}$ be an alphabet with k elements, i.e., its *cardinality* is $| \Sigma | = k$. A *string*, also called a *word*, is a sequence, i.e., an ordered set possibly with repetitions, of characters.

Example 2.1 (*binary strings*) Let $\Sigma = \{ a, b \}$ be an alphabet. Some strings are: $a\,a\,b\,a$, $a\,a\,a$, $a\,b\,a\,a$, and b. □

A *language* is a set of strings over a specified alphabet.

Example 2.2 Three sample languages over the same alphabet $\Sigma = \{ a, b \}$:

$$
\begin{aligned}
L_1 &= \{ a\,a,\ a\,a\,a \} \\
L_2 &= \{ a\,b\,a,\ a\,a\,b \} \\
L_3 &= \{ a\,b,\ b\,a,\ a\,a\,b\,b,\ a\,b\,a\,b,\ \ldots,\ a\,a\,a\,b\,b\,b,\ \ldots \} \\
&= \text{set of the strings having as many letters } a \text{ as letters } b
\end{aligned}
$$

□

Notice that a formal language viewed as a set has two layers. At the first level, there is an unordered set of nonelementary elements, the strings. At the second level, each string is an ordered set of atomic elements, the terminal characters.

Given a language, a string belonging to it is called a *sentence* or *phrase*. Thus, $b\,b\,a\,a \in L_3$ is a sentence of L_3, whereas $a\,b\,b \notin L_3$ is an *incorrect* string.

The *cardinality* or size of a language is the number of sentences it contains. For instance, $| L_2 | = | \{ a\,b\,a,\ a\,a\,b \} | = 2$. If the cardinality is finite, the language is called *finite*, too. Otherwise, there is no finite bound on the number of sentences and the language is termed *infinite*. To illustrate, languages L_1 and L_2 are finite, but language L_3 is infinite.

One can observe that a finite language is essentially a collection of words[3] sometimes called a *vocabulary*. A special finite language is the *empty set* or *language* $\emptyset$, which contains no sentence, i.e., $| \emptyset | = 0$. Usually, when a language contains just one element, the set braces are omitted by writing, e.g., $a\,b\,b$ instead of $\{ a\,b\,b \}$.

[3] In mathematical writings, the terms *word* and *string* are synonymous, while in linguistics a word is a string that has a meaning.

It is convenient to introduce the notation $|\,x\,|_b$ for the number of characters b present in a string x. For instance,

$$|\,a\,a\,b\,|_a = 2 \qquad |\,a\,b\,a\,|_a = 2 \qquad |\,b\,a\,a\,|_c = 0$$

The *length* $|\,x\,|$ of a string x is the number of characters it contains, e.g., $|\,a\,b\,| = 2$ and $|\,a\,b\,a\,a\,| = 4$. Two strings of length h and k, respectively:

$$x = a_1\,a_2\,\ldots\,a_h \qquad y = b_1\,b_2\,\ldots\,b_k$$

are *equal* if $h = k$ and $a_i = b_i$ $(1 \leq i \leq h)$. In words, by examining the strings from left to right, their respective characters coincide. Thus, we obtain:

$$a\,b\,a \neq b\,a\,a \qquad b\,a\,a \neq b\,a$$

2.2.1.1 String Operations

In order to manipulate strings, it is convenient to introduce several operations. For strings:

$$x = a_1\,a_2\,\ldots\,a_h \qquad y = b_1\,b_2\,\ldots\,b_k$$

concatenation[4] is defined as:

$$x \cdot y = a_1\,a_2\,\ldots\,a_h\,b_1\,b_2\,\ldots\,b_k$$

The dot '$\cdot$' may be dropped, writing $x\,y$ in place of $x \cdot y$. This operation, essential for formal languages, plays the role addition has in number theory.

Example 2.3 (*string concatenation*) For strings:

$$x = \text{well} \qquad y = \text{in} \qquad z = \text{formed}$$

we obtain:

$$x\,y = \text{wellin} \qquad y\,x = \text{inwell} \neq x\,y$$

$$\begin{aligned}(x\,y)\,z &= \text{wellin} \cdot \text{formed} = x\,(y\,z) = \text{well} \cdot \text{informed} \\ &= \text{wellinformed}\end{aligned}$$ □

Concatenation is clearly noncommutative, that is, the identity $x\,y \neq y\,x$ does not hold in general. The *associative* property holds:

$$(\,x\,y\,)\,z = x\,(\,y\,z\,)$$

Associativity permits to write without parentheses the concatenation of three or more strings. The length of the result is the sum of the lengths of the concatenated strings:

$$|\,x\,y\,| = |\,x\,| + |\,y\,| \tag{2.1}$$

[4] Also termed *product* in mathematical works.

2.2.1.2 Empty String

It is useful to introduce the concept of *empty* (or *null*) *string*, denoted by the Greek epsilon ε, as the only string satisfying, for every string x, the identity:

$$x\,\varepsilon = \varepsilon\,x = x$$

From (2.1), it follows that the empty string has length zero: $|\,\varepsilon\,| = 0$.

From an algebraic perspective, the empty string ε is the neutral element with respect to concatenation, because any string is unaffected by concatenating ε to the left or right.

The empty string should not be confused with the empty set $\emptyset$. In fact, the empty set as a language contains no string, whereas the set $\{\,\varepsilon\,\}$ contains one, the empty string; equivalently, the cardinalities of sets $\emptyset$ and $\{\,\varepsilon\,\}$ are 0 and 1, respectively. A language L is said to be *nullable* if and only if it includes the empty string, i.e., $\varepsilon \in L$.

2.2.1.3 Substring

Let string $x = u\,y\,v$ be written as the concatenation of three, possibly empty, strings u, y, and v. Then strings u, y, v are *substrings* of x. Moreover, string u is a *prefix* of x and string v is a *suffix* of x. A nonempty substring (or prefix or suffix) is called *proper* if it does not coincide with string x.

Let x be a nonempty string of length at least k, i.e., $|\,x\,| \geq k \geq 1$. The notation $Ini_k\,(x)$ denotes the prefix u of x having length k, to be termed the *initial* (of x) of length k.

Example 2.4 The string $x = a\,a\,b\,a\,c\,b\,a$ contains the following components:

prefixes	a, $a\,a$, $a\,a\,b$, $a\,a\,b\,a$, $a\,a\,b\,a\,c$, $a\,a\,b\,a\,c\,b$, and $a\,a\,b\,a\,c\,b\,a$
suffixes	a, $b\,a$, $c\,b\,a$, $a\,c\,b\,a$, $b\,a\,c\,b\,a$, $a\,b\,a\,c\,b\,a$, and $a\,a\,b\,a\,c\,b\,a$
substrings	all the prefixes and suffixes listed above, and the internal strings such as a, $a\,b$, $b\,a$, $b\,a\,c\,b$, …

Notice that the pair $b\,c$ is not a substring of string x although both letters b and c occur in x. The initial of length two is $Ini_2\,(a\,a\,b\,a\,c\,b\,a) = a\,a$. □

2.2.1.4 Reversal or Mirror Reflection

The characters of a string are usually read from left to right, but it is sometimes requested to reverse the order. The *reversal* or *reflection* of a string $x = a_1\,a_2\,\ldots\,a_h$ is the string $x^R = a_h\,a_{h-1}\,\ldots\,a_1$. For instance, it is:

$$x = \text{roma} \qquad\qquad x^R = \text{amor}$$

The following string identities are immediate:

$$(x^R)^R = x \qquad (x\,y)^R = y^R\,x^R \qquad \varepsilon^R = \varepsilon$$

2.2.1.5 Repetition

When a string contains repetitions, it is handy to have an operator expressing them. The m-th *power* x^m (for an integer $m \geq 1$) of a string x is the concatenation of x with itself for $m-1$ times:

$$x^m = \underbrace{x \cdot x \cdot \cdots \cdot x}_{m \text{ times}}$$

By stipulation, the zero power of any string is defined to be the empty string. Thus, the complete definition is:

$$x^m = x^{m-1} \cdot x \quad \text{for } m \geq 1 \qquad x^0 = \varepsilon$$

Here are a few examples:

$$\begin{array}{llll} x = a\,b & x^0 = \varepsilon & x^1 = x = a\,b & x^2 = (a\,b)^2 = a\,b\,a\,b \\ y = a^2 = a\,a & y^3 = a^2\,a^2\,a^2 = a^6 & \varepsilon^0 = \varepsilon & \varepsilon^2 = \varepsilon \end{array}$$

When writing formulas, the string to be repeated must be parenthesized, if it is longer than one. Thus, to express the 2-nd power of string $a\,b$, i.e., $a\,b\,a\,b$, one should write $(a\,b)^2$, not $a\,b^2$, which is the string $a\,b\,b$.

Expressed differently, we assume that the power operation takes *precedence* over concatenation. Similarly, reversal takes precedence over concatenation, e.g., $a\,b^R$ returns $a\,b$ since $b^R = b$, while $(a\,b)^R = b\,a$.

2.2.2 Language Operations

It is straightforward to extend an operation, originally defined on a string, to an entire language: just apply the operation to each one of the language sentences. By means of this general principle, the string operations previously defined can be revisited, starting from those that have one argument.

The reversal L^R of a language L is the set of the strings that are the reversal of a sentence of L:

$$L^R = \{\, x \mid \underbrace{x = y^R \ \wedge\ y \in L}_{\text{characteristic predicate}} \,\}$$

Here the strings x are specified by the property expressed in the so-called characteristic predicate.

Similarly, the set of the proper prefixes of a language L is:

$$\mathit{prefixes}\,(L) = \{\, y \mid \ x = y\,z \ \wedge\ x \in L \ \wedge\ y \neq \varepsilon \ \wedge\ z \neq \varepsilon \,\}$$

Example 2.5 (*prefix-free language*) In some applications, the loss of one or more final characters of a language sentence is required to produce an incorrect string. The motivation is that the compiler is then able to detect the inadvertent truncation of a sentence.

A language is *prefix-free* if none of the proper prefixes of its sentences is in the language, i.e., the set $\mathit{prefixes}(L)$ is disjoint from L. Thus, language $L_1 = \{ x \mid \ x = a^n b^n \ \wedge \ n \geq 1 \}$ is prefix-free since every prefix takes the form $a^n b^m$, with $n > m \geq 0$, and does not satisfy the characteristic predicate.

On the other hand, the language $L_2 = \{ a^m b^n \mid \ m > n \geq 1 \}$ contains $a^3 b^2$ as well as the proper prefix $a^3 b$. □

Similarly, the operations on two strings can be extended to two languages, by letting the former and latter argument span the respective language. For instance, the *concatenation* of languages L' and L'' is defined as:

$$L' \cdot L'' = \left\{ x \cdot y \mid \ x \in L' \ \wedge \ y \in L'' \right\}$$

From this, extending the m-th *power* operation to a language is straightforward ($m \geq 0$):

$$L^m = L^{m-1} \cdot L \quad \text{for } m \geq 1 \qquad\qquad L^0 = \{ \varepsilon \}$$

Some special cases follow from the previous definitions:

$$\emptyset^0 = \{ \varepsilon \} \qquad\qquad L \cdot \emptyset = \emptyset \cdot L = \emptyset \qquad\qquad L \cdot \{ \varepsilon \} = \{ \varepsilon \} \cdot L = L$$

Example 2.6 (*language concatenation*) Consider languages L_1 and L_2:

$$\begin{aligned} L_1 &= \left\{ a^i \mid \ i \geq 0 \ \wedge \ i \text{ is even} \right\} = \{ \varepsilon, \ a\,a, \ a\,a\,a\,a, \ \ldots \} \\ L_2 &= \left\{ b^j\,a \mid \ j \geq 1 \ \wedge \ j \text{ is odd} \right\} = \{ b\,a, \ b\,b\,b\,a, \ \ldots \} \end{aligned}$$

We obtain this concatenation $L_1 \cdot L_2$:

$$\begin{aligned} L_1 \cdot L_2 &= \left\{ a^i \cdot b^j\,a \mid \ (i \geq 0 \ \wedge \ i \text{ is even}) \ \wedge \ (j \geq 1 \ \wedge \ j \text{ is odd}) \right\} \\ &= \left\{ \varepsilon\,b\,a, \ a^2\,b\,a, \ a^4\,b\,a, \ \ldots, \ \varepsilon\,b^3\,a, \ a^2\,b^3\,a, \ a^4\,b^3\,a, \ \ldots \right\} \end{aligned}$$ □

A common error when computing a power of a base language is to take the *same* string for m times. The result is a different set, included in the power of the base:

$$\left\{ x \mid \ x = y^m \ \wedge \ y \in L \right\} \subseteq L^m \qquad m \geq 2 \tag{2.2}$$

Thus, in (2.2) for $L = \{ a, \ b \}$ and $m = 2$, the set to the left of '$\subseteq$' is $\{ a\,a, \ b\,b \}$, while the value of L^m is $\{ a\,a, \ a\,b, \ b\,a, \ b\,b \}$.

Example 2.7 (*strings of finite length*) The power operation allows a concise definition of the strings of length not exceeding some integer $k \geq 0$. Consider the alphabet $\Sigma = \{a, b\}$. For $k = 3$, the language L:

$$\begin{aligned} L &= \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb\} \\ &= \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \end{aligned}$$

can also be defined as:

$$L = \{\varepsilon, a, b\}^3$$

Notice that the sentences shorter than 3 are obtained by using the empty string ε of the base language.

By slightly changing the example, the language $L' = \{x \mid \ 1 \leq |x| \leq 3\}$ is defined, with concatenation and power, by the formula:

$$L' = \{a, b\} \cdot \{\varepsilon, a, b\}^2 \qquad \square$$

2.2.3 Set Operations

Since a language is a set, the classical set operations of union '$\cup$', intersection '$\cap$', and difference '$\setminus$' apply to languages. The set relations of inclusion '$\subseteq$', strict inclusion '$\subset$', and equality '$=$' apply as well.

Before introducing the complement of a language, the notion of *universal language* is needed: it is defined as the set of all the strings, over an alphabet Σ, of any length including zero. Clearly, the universal language is infinite and can be viewed as the union of all the powers of the alphabet:

$$L_{universal} = \Sigma^0 \cup \Sigma \cup \Sigma^2 \cup \ldots$$

The *complement* of a language L over an alphabet Σ, denoted by $\neg L$, is the set difference:

$$\neg L = L_{universal} \setminus L$$

that is, the set of the strings over alphabet Σ that are not in L. When the alphabet is understood, the universal language can be expressed as the complement of the empty language:

$$L_{universal} = \neg \emptyset$$

Example 2.8 (*language complement*) The complement of a finite language is always infinite; for instance, the set of strings of any length except two is:

$$\neg \left(\{a, b\}^2\right) = \{\varepsilon\} \cup \{a, b\} \cup \{a, b\}^3 \cup \ldots$$

On the other hand, the complement of an infinite language may or may not be finite, as shown on one side by the complement of the universal language, and on the other side by the complement of the set of the strings of even length over alphabet $\{a\}$:

$$L = \left\{ a^{2n} \mid \ n \geq 0 \right\} \qquad \neg L = \left\{ a^{2n+1} \mid \ n \geq 0 \right\}$$

Moving to set difference, consider an alphabet $\Sigma = \{a, b, c\}$ and languages:

$$L_1 = \{x \mid \ |x|_a = |x|_b = |x|_c \geq 0\}$$
$$L_2 = \{x \mid \ |x|_a = |x|_b \ \wedge \ |x|_c = 1\}$$

Then the set differences are:

$$L_1 \setminus L_2 = \{\varepsilon\} \cup \{x \mid \ |x|_a = |x|_b = |x|_c \geq 2\}$$

which represents the set of the strings that have the same number (except one) of occurrences of letters a, b, and c, and:

$$L_2 \setminus L_1 = \{x \mid \ |x|_a = |x|_b \neq |x|_c \ \wedge \ |x|_c = 1\}$$

which is the set of the strings that have one c and the same number of occurrences (except one) of a and b. □

2.2.4 Star and Cross

Most artificial and natural languages include sentences that can be lengthened at will, so causing the number of sentences in the language to be unbounded. On the other hand, all the operations so far defined, with the exception of complement, do not allow to write a finite formula denoting an infinite language. In order to enable the definition of an infinite language, the next essential development extends the power operation to the limit.

2.2.4.1 Star

The *star*[5] operation '$*$' is defined as the union of all the powers of the base language:

$$L^* = \bigcup_{h=0}^{\infty} L^h = L^0 \cup L^1 \cup L^2 \cup \ldots = \{\varepsilon\} \cup L \cup L^2 \cup \ldots$$

[5] Also known as Kleene star and as reflexive transitive closure by concatenation.

Example 2.9 (*language star*) For the language $L = \{ a\,b,\ b\,a \}$, the star is:

$$L^* = \{ \varepsilon,\ a\,b,\ b\,a,\ a\,b\,a\,b,\ a\,b\,b\,a,\ b\,a\,a\,b,\ b\,a\,b\,a,\ \ldots \}$$

Every nonempty string of the 'starred' language L^* can be segmented into substrings, which are the sentences of the base language L.

Notice that, if the base language contains at least one nonempty string, the starred language is infinite.

It may happen that the starred and base languages are identical, as in:

$$L = \left\{ a^{2n} \mid \ n \geq 0 \right\} \qquad\qquad L^* = \left\{ a^{2n} \mid \ n \geq 0 \right\} = L \qquad \square$$

An interesting special case occurs when the base is an alphabet Σ; then the star Σ^* contains all the strings[6] obtained by concatenating terminal characters. This language is the same as the previous *universal language*[7] of alphabet Σ.

It is obvious that any formal language is a subset of the universal language over the same alphabet, and the relation $L \subseteq \Sigma^*$ is often written to say that L is a language over an alphabet Σ. Some useful properties of star follow:

properties of star	*meaning*
$L \subseteq L^*$	monotonicity
if $x \in L^* \ \wedge \ y \in L^*$ **then** $x\,y \in L^*$	closure by concatenation
$(L^*)^* = L^*$	idempotence
$(L^*)^R = \left(L^R\right)^*$	commutativity with reversal

Example 2.10 (*idempotence*) The monotonicity property states that any language is included in its star. But for language $L = \left\{ a^{2n} \mid \ n \geq 0 \right\}$, the equality $L^* = L$ follows from the idempotence property and from the fact that L can be equivalently defined by the starred formula $\{ a\,a \}^*$. $\square$

For the empty language $\emptyset$ and empty string ε, we have the set identities:

$$\emptyset^* = \{ \varepsilon \} \qquad\qquad \{ \varepsilon \}^* = \{ \varepsilon \}$$

[6] The length of a sentence in Σ^* is unbounded, yet it may not be considered infinite. A specialized branch of formal language theory (see Perrin and Pin [7]) is devoted to the so-called infinitary or omega-languages, which include also sentences of infinite length. They effectively model the situations when a perpetual system can receive or produce messages of infinite length.

[7] Another name for it is *free monoid*. In algebra, a monoid is a structure provided with an associative composition law (concatenation) and a neutral element (empty string).

Example 2.11 (*identifiers*) Many artificial languages assign a name or identifier to each entity, i.e., variable, file, document, subprogram, and object. A common naming rule prescribes that an identifier is a string that has the initial character in the set $\{A, B, \ldots, Z\}$ and contains any number of letters or digits $\{0, 1, \ldots, 9\}$, e.g., LOOP3A2.

By using the alphabets Σ_A and Σ_N:

$$\Sigma_A = \{A, B, \ldots, Z\} \qquad \Sigma_N = \{0, 1, \ldots, 9\}$$

the language of identifiers $I \subseteq (\Sigma_A \cup \Sigma_N)^*$ is:

$$I = \Sigma_A (\Sigma_A \cup \Sigma_N)^*$$

To introduce a variant, prescribe that the length of the identifiers should not exceed five. By defining an alphabet $\Sigma = \Sigma_A \cup \Sigma_N$, the language is:

$$\begin{aligned} I_5 &= \Sigma_A \left(\Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \Sigma^4\right) \\ &= \Sigma_A \left(\varepsilon \cup \Sigma \cup \Sigma^2 \cup \Sigma^3 \cup \Sigma^4\right) \end{aligned}$$

The formula expresses the concatenation of language Σ_A, the sentences of which are single characters, with the language constructed as the union of powers. A more elegant writing is:

$$I_5 = \Sigma_A (\varepsilon \cup \Sigma)^4$$

□

2.2.4.2 Cross

A useful though dispensable operator, derived from star, is the *cross*[8] '+':

$$L^+ = \bigcup_{h=1}^{\infty} L^h = L \cup L^2 \cup \ldots$$

Cross differs from star because the union excludes the power zero. The following relations hold:

$$L^+ \subseteq L^* \qquad \varepsilon \in L^+ \text{ if and only if } \varepsilon \in L \qquad L^+ = L L^* = L^* L$$

Star and cross are called *iteration operators*.

Example 2.12 (*language cross*) Two applications of cross to finite sets:

$$\begin{aligned} \{a\,b,\ b\,b\}^+ &= \left\{a\,b,\ b^2,\ a\,b^3,\ b^2\,a\,b,\ a\,b\,a\,b,\ b^4,\ \ldots\right\} \\ \{\varepsilon,\ a\,a\}^+ &= \left\{\varepsilon,\ a^2,\ a^4,\ \ldots\right\} = \left\{a^{2n} \mid\ n \geq 0\right\} \end{aligned}$$

□

[8] Or irreflexive closure by concatenation.

Not surprisingly, a language can usually be defined by various formulas, which differ by their use of operators.

Example 2.13 (*controlling string length*) Two ways of defining the strings of four or more characters:

- concatenating the strings of length 4 with arbitrary strings: $\Sigma^4 \cdot \Sigma^*$
- constructing the 4-th power of the set of nonempty strings: $\left(\Sigma^+\right)^4$ □

2.2.5 Language Quotient

Operations like concatenation, star, or union lengthen the strings or increase the cardinality of the set of strings they operate upon. On the other hand, given two languages, the *right quotient* operation $L'/_R L''$ shortens the sentences of the first language L' by cutting a suffix, which is a sentence of the second language L''. The right quotient of L' with respect to L'' is defined as:

$$L = L'/_R L'' = \left\{ y \mid \ \exists z \ \text{ such that } \ y\,z \in L' \ \wedge \ z \in L'' \right\}$$

Example 2.14 (*quotient language*) Consider two languages L' and L'':

$$L' = \left\{ a^{2n}\, b^{2n} \mid \ n > 0 \right\} \qquad\qquad L'' = \left\{ b^{2n+1} \mid \ n \geq 0 \right\}$$

Their right quotients are:

$$\begin{aligned} L'/_R L'' &= \{\, a^r\, b^s \mid \ \ (\, r \geq 2 \text{ is even}\,) \ \wedge \ (\, 1 \leq s < r \text{ is odd}\,) \,\} \\ &= \left\{ a^2\, b,\ a^4\, b,\ a^4\, b^3,\ \ldots \right\} \\ L''/_R L' &= \emptyset \end{aligned}$$

□

A dual operation is the *left quotient* $L''/_L L'$ that shortens the sentences of language L' by cutting a prefix, which is a sentence of language L''.

More operations will be introduced later, in order to transform or translate a formal language by replacing the terminal characters with other characters or strings.

2.3 Regular Expressions and Languages

Theoretical investigation on formal languages has invented various categories of languages, in a way reminiscent of the classification of numerical domains introduced much earlier by number theory. Such categories are characterized by mathematical and algorithmic properties.

The first family of formal languages to be considered is called *regular* (or rational) and can be defined by an astonishing number of different approaches. Regular languages have been independently discovered in disparate scientific fields: the study of input signals driving a sequential circuit[9] to a certain state, the lexicon of programming languages modeled by simple grammar rules, and the simplified analysis of neural behavior. Later such approaches have been complemented by a logical definition based on a restricted form of predicates.

To introduce the family, the first definition will be algebraic, by using the operations of union, concatenation, and star; then the family will be defined again by means of certain simple grammar rules; last, Chap. 3 describes the algorithm for recognizing regular languages in the form of an abstract machine, the finite automaton.[10]

2.3.1 Definition of Regular Expression

A language over alphabet $\Sigma = \{ a_1, a_2, \ldots, a_n \}$ is *regular* if it can be expressed by applying for a finite number of times the operations of concatenation, union, and star, starting with the unitary languages[11] $\{ a_1 \}, \{ a_2 \}, \ldots, \{ a_n \}$ or the empty string ε.

More precisely, a *regular expression* (r.e.) is a string r containing the terminal characters of the alphabet Σ and the following metasymbols[12]:

"$\cup$" union "$\cdot$" concatenation "$*$" star

"ε" empty (or null) string "( )" parentheses

in accordance with the following rules:

#	*rule*	*meaning*
1	$r = \varepsilon$	empty (or null) string
2	$r = a$	unitary language
3	$r = (s \cup t)$	union of expressions
4	$r = (s \cdot t)$ or $r = (s\,t)$	concatenation of expressions
5	$r = (s)^*$	iteration (star) of an expression

where the symbols s and t are regular (sub)expressions.

For expressivity, the metasymbol cross is allowed in an r.e., since it can be expressed using rules 4 and 5 as $(s)^+ = (s\ (s^*))$. For economy, some parentheses

[9] A digital component incorporating a memory.

[10] The language family can also be defined by the form of the logical predicates characterizing language sentences, e.g., as in [8].

[11] A unitary language contains one sentence.

[12] In order to prevent confusion between terminals and metasymbols, the latter should not be in the alphabet. If not so, the metasymbols must be suitably recoded to make them distinguishable.

can be dropped by stipulating the following precedence between operators: first apply star, then concatenation, and last union. Furthermore, the union of three (or more) terms, e.g., $((r \cup s) \cup t)$ and $(r \cup (s \cup t))$, can be simplified as $(r \cup s \cup t)$ because union is an associative operation. The same remark applies to concatenation.

It is customary to write the cup '$\cup$' symbol as a vertical bar '|', called an *alternative*.

In more theoretical books on formal languages, the metasymbol '$\emptyset$' (empty set) is allowed in an r.e., which permits to obtain the empty string 'ε' from the basic operations through the identity $\{\varepsilon\} = \emptyset^*$. We prefer the more direct and practical presentation shown above.

The rules from 1 to 5 compose the syntax of the r.e., to be formalized later by means of a grammar (Example 2.31 on p. 33). The meaning or denotation of an r.e. r is a language L_r over the alphabet Σ, defined by the correspondence shown in Table 2.1.

Example 2.15 (*language of the multiples of* 3) Let the alphabet be $\Sigma = \{1\}$, where character '1' may be viewed as a pulse or signal. An r.e. e and the language L_e it denotes are:

$$e = (1\,1\,1)^* \qquad L_e = \left\{ 1^n \mid \quad n \mod 3 = 0 \right\}$$

Language L_e contains the character sequences multiple of three. Notice that when dropping parentheses the language changes, due to the precedence of star over concatenation:

$$e_1 = 1\,1\,1^* = 1\,1\,(1)^* \qquad\qquad L_{e_1} = \left\{ 1^n \mid \quad n \geq 2 \right\} \qquad \square$$

Example 2.16 (*language of integers*) Let the alphabet be $\Sigma = \{+, -, d\}$, where character d denotes any decimal digit 0, 1, …, 9. The expression e:

$$e = (+ \cup - \cup \varepsilon)\; d\, d^* \equiv (+ \mid - \mid \varepsilon)\; d\, d^*$$

produces the language $L_e = \{+, -, \varepsilon\}\,\{d\}\,\{d\}^*$ of the integers with or without sign, such as $+353$, -5, 969, and $+001$. $\square$

Table 2.1 Language L_r denoted by a regular expression r

#	Regular expression r	Language L_r denoted by r
1	ε	$\{\varepsilon\}$
2	a	$\{a\}$ where $a \in \Sigma$
3	$s \cup t$ or more often $s \mid t$	$L_s \cup L_t$
4	$s \cdot t$ or more often $s\,t$	$L_s \cdot L_t$ or more often $L_s\,L_t$
5	s^*, s^+	L_s^*, L_s^+

Actually, the correspondence between an r.e. and its denoted language is so direct that it is customary to refer to the language L_e by the r.e. e itself.

We introduce the established terminology for two language families. A language is *regular* if it is denoted by a regular expression. The empty language $\emptyset$ is considered a regular language as well, despite there is not an r.e. that defines it. The collection of all regular languages is called the *family REG* of *regular languages*.

Another simple family of languages is the collection of all the *finite languages*, and it is called *FIN*. Thus, a language is in the family *FIN* if its cardinality is finite, as for instance the language of 32-bit binary numbers.

By comparing the families *REG* and *FIN*, it is easy to see that every finite language is regular, i.e., $FIN \subseteq REG$. In fact, a finite language is the union of finitely many strings $x_1, x_2, \ldots, x_k$, each one being the concatenation of finitely many characters, i.e., $x_i = a_1\, a_2\, \ldots\, a_{n_i}$. Then an r.e. producing such finite language is simply the union of k terms x_i, each one concatenating n_i characters. Since family *REG* includes nonfinite languages, too, the inclusion between the two families is proper, i.e., $FIN \subset REG$.

More language families will be later introduced and compared with *REG*.

2.3.2 Derivation and Language

We formalize the mechanism through which a given r.e. e produces the denoted language. For the moment, we suppose that r.e. e is fully parenthesized (except for the atomic terms), and we introduce the notion of *subexpression* (s.e.) in the next example:

$$e_0 = \Big(\overbrace{\big((a \cup (b\,b))^*\big)}^{\text{subexpression } e_1 \text{ of } e_0} \overbrace{\big(\big(c^+\big) \cup \underbrace{(a \cup (b\,b))}_{\text{subexpr. } s \text{ of } e_2} \big)}^{\text{subexpression } e_2 \text{ of } e_0} \Big)$$

This r.e. e_0 is structured as the concatenation of two parts e_1 and e_2, to be called subexpressions (of e_0). In general, an s.e., f, of an r.e. e is a well-parenthesized substring immediately occurring inside the outermost parentheses. This means that there is not another well-parenthesized substring of e that contains f. In the example, the substring labelled s is not an s.e. of e_0, but it is an s.e. of e_2.

When an r.e. is not fully parenthesized, in order to identify its subexpressions one has to insert (or to imagine) the missing parentheses, in agreement with the operator precedence. We recall that three or more terms combined by union do not need to be pairwise parenthesized, because union is an associative operation. The same remark applies to three or more concatenated terms.

A union or iteration (star and cross) operator offers different choices for producing strings. By making a choice, one obtains an r.e. that defines a less general language, which is included in the original one. We say that an r.e. is a *choice* of another one in the following three cases:

1. r.e. e_k (with $1 \leq k \leq m$ and $m \geq 2$) is a choice of the union:

$$(e_1 \cup \cdots \cup e_k \cup \cdots \cup e_m)$$

2. r.e. $e^m = \underbrace{e \cdot \cdots \cdot e}_{m \text{ times}}$ (with $m \geq 1$) is a choice of the star e^* or cross e^+
3. the empty string ε is a choice of the star e^*

Let e' be an r.e. By substituting some choice for e', a new r.e. e'' can be derived from e'. The corresponding relation, called *derivation*, between two regular expressions e' and e'' is defined next.

Definition 2.17 (*derivation*[13]) We say that an r.e. e' *derives* an r.e. e'', written $e' \Rightarrow e''$, if one of the two conditions below holds true:

1. r.e. e'' is a choice of e'
2. r.e. e' is the concatenation of $m \geq 2$ s.e., as follows:

$$e' = e'_1 \ldots e'_k \ldots e'_m$$

and r.e. e'' is obtained from e' by substituting an s.e., say e'_k, with a choice of e'_k, say e''_k, as shown below:

$$\exists\, k,\ 1 \leq k \leq m,\ \text{such that } e''_k \text{ is a choice of } e'_k \ \wedge\ e'' = e'_1 \ldots e''_k \ldots e'_m$$

Such a derivation $\Rightarrow$ is called *immediate* as it makes exactly one choice. If two or more immediate derivations are applied in series, thus making as many choices altogether, we have a multi-step derivation. We say that an r.e. e_0 derives an r.e. e_n in $n \geq 1$ steps, written $e_0 \overset{n}{\Rightarrow} e_n$, if the following immediate derivations apply:

$$e_0 \Rightarrow e_1 \qquad e_1 \Rightarrow e_2 \qquad \ldots \qquad e_{n-1} \Rightarrow e_n$$

The notation $e' \overset{+}{\Rightarrow} e''$ says that an r.e. e' derives an r.e. e'' in $n \geq 1$ steps, without specifying n. If the number of steps is $n = 0$, then the identity $e_0 = e_n$ holds and says that the derivation relation is reflexive. We also write $e' \overset{*}{\Rightarrow} e''$ if it holds $e' \overset{+}{\Rightarrow} e''$ or $e' = e''$. □

Example 2.18 (*immediate and multi-step derivation*) Immediate derivations:

$$a^* \cup b^+ \Rightarrow a^* \qquad a^* \cup b^+ \Rightarrow b^+$$
$$\left(a^* \cup bb\right)^* \Rightarrow \left(a^* \cup bb\right)\left(a^* \cup bb\right)$$

[13] Also called *implication*.

Notice that the substrings of the r.e. considered must be chosen in order from external to internal, if one wants to produce all possible derivations. For instance, it would be unwise, starting from $e' = (a^* \cup b\,b)^*$, to choose $\left(a^2 \cup b\,b\right)^*$, because a^* is not an s.e. of e'. Although the value 2 is a correct choice for the star, such a premature choice would rule out the derivation of a valid sentence such as $a^2\,b\,b\,a^3$.

Multi-step derivations:

$$a^* \cup b^+ \Rightarrow a^* \Rightarrow \varepsilon \text{ that is } a^* \cup b^+ \overset{2}{\Rightarrow} \varepsilon \quad \text{or also } a^* \cup b^+ \overset{+}{\Rightarrow} \varepsilon$$

$$a^* \cup b^+ \Rightarrow b^+ \Rightarrow b\,b\,b \quad \text{or also } a^* \cup b^+ \overset{+}{\Rightarrow} b\,b\,b$$ □

Some expressions produced by derivation from an expression r contain the meta-symbols of union, star, and cross. Others just contain terminal characters or the empty string, and possibly redundant parentheses, which can be canceled. The latter expressions compose the language denoted by the r.e.

The *language defined by a regular expression r* is:

$$L_r = \left\{ x \in \Sigma^* \mid \; r \overset{*}{\Rightarrow} x \right\}$$

Two r.e. are *equivalent* if they define the same language.

The coming example shows that different derivation orders may produce the same sentence.

Example 2.19 (*derivation orders*) Compare the derivations 1 and 2 below:

steps	1-*st*	2-*nd*	3-*rd*
1: $a^*\,(b \cup c \cup d)\,f^+$	$\Rightarrow a\,a\,a\,(b \cup c \cup d)\,f^+$	$\Rightarrow a\,a\,a\,c\,f^+$	$\Rightarrow a\,a\,a\,c\,f$
2: $a^*\,(b \cup c \cup d)\,f^+$	$\Rightarrow a^*\,c\,f^+$	$\Rightarrow a\,a\,a\,c\,f^+$	$\Rightarrow a\,a\,a\,c\,f$

At line 1, the first step takes the leftmost s.e. (a^*) and chooses $a^3 = a\,a\,a$, whereas at line 2 it takes another s.e. ($b \cup c \cup d$) and chooses c. Since such choices are mutually independent, they can be applied in any order. Thus, by the second step, we obtain the same r.e. $a\,a\,a\,c\,f^+$. The third step takes s.e. f^+, chooses f, and produces the sentence $a\,a\,a\,c\,f$. Since the last step is independent of the other two, it could be performed before, after, or between them. □

2.3.2.1 Ambiguity of Regular Expressions

The next example conceptually differs from the preceding one with respect to how different derivations produce the same sentence.

Example 2.20 (*ambiguous regular expression*) The language over alphabet $\{a, b\}$ such that every sentence contains one or more letters a is defined by:

$$(a \cup b)^*\,a\,(a \cup b)^*$$

where the compulsory presence of an a is evident. Clearly, every sentence containing two or more occurrences of a can be obtained by multiple derivations, which differ with respect to the character identified as the compulsory one. For instance, sentence $a\,a$ offers two possibilities:

$$(a \cup b)^*\ a\ (a \cup b)^* \Rightarrow (a \cup b)\ a\ (a \cup b)^* \Rightarrow a\,a\ (a \cup b)^* \Rightarrow a\,a\,\varepsilon = a\,a$$
$$(a \cup b)^*\ a\ (a \cup b)^* \Rightarrow \varepsilon\,a\ (a \cup b)^* \Rightarrow \varepsilon\,a\ (a \cup b) \Rightarrow \varepsilon\,a\,a = a\,a$$

□

A sentence, and the r.e. that derives it, is said to be *ambiguous* if, and only if, it can be obtained via two structurally different derivations. Thus, sentence $a\,a$ is ambiguous, while sentence $b\,a$ is not ambiguous, because only one set of choices is possible, corresponding to the derivation:

$$(a \cup b)^*\ a\ (a \cup b)^* \Rightarrow (a \cup b)\ a\ (a \cup b)^* \Rightarrow b\,a\ (a \cup b)^* \Rightarrow b\,a\,\varepsilon = b\,a$$

Ambiguous definitions are a source of trouble in many settings, although they may be useful in certain applications. In general, such definitions should be avoided although they may have the advantage of concision over unambiguous definitions. We would like to have an algorithm for checking if an r.e. is ambiguous, but for that we have to wait till Chap. 3.

For the time being, we just state a simple sufficient condition for a sentence and its r.e. to be ambiguous. To this end, given an r.e. f, we number its letters and we obtain a *numbered* r.e., which we denote by f':

$$f' = (a_1 \cup b_2)^*\ a_3\ (a_4 \cup b_5)^*$$

Notice that the numbered r.e. f' defines a language over the numbered alphabet $\{a_1, b_2, a_3, a_4, b_5\}$, in general larger than the original one.

An r.e. f is *ambiguous* if the language defined by the numbered r.e. f' associated to f contains two distinct strings x and y that become identical when the numbers are erased. For instance, the strings $a_1\,a_3$ and $a_3\,a_4$ of language f' prove the ambiguity of the string $a\,a$ of language f.

The same criterion can be applied to an r.e. e that includes the metasymbol ε. For instance, consider the r.e. $e = (a \mid \varepsilon)^+$. Then the numbered r.e. is $e' = (a_1 \mid \varepsilon_2)^+$, from which the two strings $a_1\,\varepsilon_2$ and $\varepsilon_2\,a_1$ can be derived that are mapped to the same ambiguous string a.

Notice that the sufficient condition does not cover cases like the following. The r.e. $\left(a^+ \mid b\right)^+ b$, numbered $\left(a_1^+ \mid b_2\right)^+ b_3$, ambiguously derives the string $a\,a\,b$ with the unique numbering $a_1\,a_1\,b_3$, yet with two distinct derivations:

$$\left(a^+ \mid b\right)^+ b \Rightarrow \left(a^+ \mid b\right)\left(a^+ \mid b\right) b \overset{+}{\Rightarrow} a^+\,a^+\,b \overset{+}{\Rightarrow} a\,a\,b$$

and

$$\left(a^+ \mid b\right)^+ b \Rightarrow \left(a^+ \mid b\right) b \Rightarrow a^+\,b \Rightarrow a\,a\,b$$

Similarly, the r.e $(a^* \mid b^*)\, c$ has the ambiguous derivations:

$$\left(a^* \mid b^*\right) c \Rightarrow a^* c \Rightarrow \varepsilon\, c = c \quad \text{and} \quad \left(a^* \mid b^*\right) c \Rightarrow b^* c \Rightarrow \varepsilon\, c = c$$

which are not detected by our condition.

The concept of ambiguity will be thoroughly studied for grammars.

2.3.3 Other Operators

When regular expressions are used in practice, it is convenient to add to the *basic operators* of *union*, *concatenation*, and *star*, the derived operators of *power* and *cross*. Moreover, for better expressivity other derived operators may be practical:

repetition from $k \geq 0$ to $n > k$ times: $[\,a\,]_k^n = a^k \cup a^{k+1} \cup \cdots \cup a^n$
option: $[\,a\,] = [\,a\,]_0^1 = a^0 \cup a^1 = \varepsilon \cup a$
interval of an ordered set: the short notation $(0 \ldots 9)$ represents any digit in the ordered set $\{0,\ 1,\ \ldots,\ 9\}$; similarly, the notations $(a \ldots z)$ and $(A \ldots Z)$ respectively represent any lowercase/uppercase letter

Sometimes other set operations are also used: *intersection*, *set difference*, and *complement*; the regular expressions using such operators are called *extended*, although the name is not standard and one has to specify the operators allowed case by case.

Example 2.21 (*extended r.e. with intersection*) This operator provides a straightforward formulation when a string, to be valid, must obey two or more conditions. To illustrate, let the alphabet be $\{a,\ b\}$ and assume a valid string must (*i*) contain the substring $b\,b$ and (*ii*) have an even length. Condition (*i*) is imposed by r.e.:

$$(a \mid b)^*\, b\,b\, (a \mid b)^*$$

condition (*ii*) by r.e.:

$$\left((a \mid b)^2\right)^*$$

and the language is defined by the r.e. extended with intersection:

$$\left((a \mid b)^*\, b\,b\, (a \mid b)^*\right) \cap \left((a \mid b)^2\right)^*$$

The same language can be defined by a basic r.e. without intersection, but the formula is more complicated. It says that the substring $b\,b$ can be surrounded by two strings, both of even length or both of odd length:

$$\left((a \mid b)^2\right)^*\, b\,b\, \left((a \mid b)^2\right)^* \mid (a \mid b)\, \left((a \mid b)^2\right)^*\, b\,b\, (a \mid b)\, \left((a \mid b)^2\right)^* \qquad \square$$

Furthermore, it is sometimes simpler to define the sentences of a language *ex negativo*, by stating a property they should not have.

Example 2.22 (*extended r.e. with complement*) Let L be the set of strings over alphabet $\{a, b\}$ not containing $a\,a$ as substring. Its complement is:

$$\neg L = \left\{ x \in (a \mid b)^* \mid \ x \text{ contains substring } a\,a \right\}$$

easily defined by r.e. $(a \mid b)^*\ a\,a\ (a \mid b)^*$, whence an extended r.e. for L is:

$$L = \neg \left((a \mid b)^*\ a\,a\ (a \mid b)^* \right)$$

We also show a definition of language L by a basic r.e.:

$$L = (a\,b \mid b)^*\ (a \mid \varepsilon)$$

which most readers are likely to find less readable. □

Actually, it is no coincidence that the languages in Examples 2.21 and 2.22 admit also an r.e. without intersection or complement. A theoretical result, to be presented in Chap. 3, states that the language produced by any r.e. extended with complement and intersection is regular, therefore by definition can be produced by a nonextended r.e. as well.

2.3.4 Closure Properties of Family *REG*

Let *op* be an operator to be applied to one or two languages, to produce another language. A language family is *closed under operator op* if the language, obtained by applying *op* to any languages of the family, belongs to the same family.

Property 2.23 (closure of REG) *The family* REG *of regular languages is closed under the operators of concatenation, union, and star; therefore, it is closed also under any derived operator, such as cross.* □

This closure property descends from the very definitions of r.e. and of family *REG* (p. 20). In spite of its theoretical connotation, Property 2.23 is very relevant in practice: two regular languages can be combined by using the above operations, at no risk of losing the nice features of family *REG*. This will have an important practical consequence, to permit the compositional design of the algorithms used to check if an input string is valid for a language. Furthermore, we anticipate that the *REG* family is closed under intersection, complement, and reversal, too, which will be proved later.

From the above property, we derive an alternative definition of family *REG*.

Property 2.24 (characterization of REG) *The family of regular languages,* REG*, is* the smallest *language family such that: (*i*) it contains all finite languages and (*ii*) it is closed by concatenation, union, and star.* □

Proof By contradiction, assume there is a smaller family F, i.e., $F \subset REG$, that meets conditions (i) and (ii). Take any language $L_e \in REG$ defined by an r.e. e. Language L_e is obtained by starting from unitary languages, and orderly applying the operators present in e. From the assumption, it follows that L_e belongs also to family F. Then F contains any regular language and thus contradicts the strict inclusion $F \subset REG$. □

We anticipate that other language families exist which are closed under the same operators of Property 2.23. Chief among them is the family, *CF*, of context-free languages, to be introduced soon. From Property 2.24, it follows a (strict) containment relation between these two families, i.e., $REG \subset CF$.

2.4 Linguistic Abstraction: Abstract and Concrete Lists

If one recalls the programming languages he is familiar with, he may observe that, although superficially different in their use of keywords and separators, they are often quite similar at a deeper level. By shifting focus from concrete to abstract syntax, we can reduce the bewildering variety of language constructs to a few essential structures. The verb 'to abstract' means:

> to consider a concept without thinking of a specific example, WordNet 2.1

By abstracting away from the actual characters that represent a language construct, we perform a linguistic abstraction. This is a language transformation that replaces the terminal characters of the concrete language with others taken from an abstract alphabet. Abstract characters should be simpler and suitable to represent similar constructs from different artificial languages.[14]

By this approach, the abstract syntax structures of existing artificial languages are easily described as composition of few elementary paradigms, by means of standard language operations: union, iteration, and substitution (later defined). Starting from the abstract language, a concrete or real language is obtained by the reverse transformation, metaphorically also called 'coating with syntax sugar.'

Factoring a language into its abstract and concrete syntax pays off in several ways. When studying different languages, it affords much conceptual economy. When designing compilers, abstraction helps for portability across different languages, if

[14]The idea of language abstraction is inspired by the research line in linguistics that aims at discovering the underlying similarities between human languages, disregarding morphological and syntactic differences.

the compiler functions are designed to process abstract, instead of concrete, language constructs. Thus, parts of, say, a C compiler can be reused for similar languages, like Java.

Since the number of abstract paradigms in use is surprisingly small, we will be able to present most of them in this chapter, starting from the ones conveniently specified by regular expressions, i.e., the lists.

An abstract *list* contains an unbounded number of elements e of the same type. It is defined by r.e. e^+, or by r.e. e^* if an empty list is permitted. For the time being, a list element is viewed as a terminal character. In later refinements, however, an element may be a string from another formal language; for instance, think of a list of numbers.

2.4.1 Lists with Separators and Opening/Closing Marks

In many real cases, adjacent elements must be divided by a string called *separator*, denoted s in abstract syntax. Thus, in a list of numbers, a separator is needed to delimit the end of a number and the beginning of the next one.

A *list with separators* is defined by r.e. $e\ (s\,e)^*$, saying that the first element e can be followed by zero or more pairs $s\,e$. The equivalent definition $(e\,s)^*\ e$ differs by giving evidence to the last element.

In many concrete cases, there is another requirement, intended for legibility or computer processing: to make the start and end of the list easily recognizable by respectively prefixing and suffixing some special signs: in the abstract, the initial character or *opening mark* i, and the final character or *closing mark* f.

Lists with separators and opening/closing marks are defined as:

$$i\,e\ (s\,e)^*\,f$$

Example 2.25 (*some concrete lists*) Lists are everywhere in languages, as shown by typical examples.

instruction block as in

$$\mathsf{begin}\ \ \mathsf{instr}_1\ ;\ \mathsf{instr}_2\ ;\ \ldots ;\ \mathsf{instr}_n\ \mathsf{end}$$

where instr may stand for assignment, goto, if-statement, write-statement, etc.; corresponding abstract and concrete terms are:

abstract alphabet	*concrete alphabet*
i	begin
e	instr
s	;
f	end

procedure parameters as in

procedure WRITE (	par_1	,	par_2	,	...	par_n	)
i	e	s	e	...	...	e	f

Furthermore, should an empty parameter list be legal, as in a declaration like procedure WRITE ( ), the r.e. becomes $i \left[e \ (s\, e)^* \right] f$

array definition as in

array MATRIX [	int_1	,	int_2	,	...	int_n	]
i	e	s	e	...	...	e	f

where each int is an interval such as 10 ... 50 □

2.4.2 Language Substitution

The above examples have illustrated the mapping from concrete to abstract symbols. Moreover, language designers find it useful to work by stepwise refinement, as done in any branch of engineering when a complex system is divided into its components, atomic or otherwise. To this end, we introduce the new language operation of substitution, which replaces a terminal character of a language called the *source*, with a sentence of another language called the *target*. As always, the source alphabet is Σ and the source language is $L \subseteq \Sigma^*$. Consider a sentence of L containing one or more occurrences of a source character b:

$$x = a_1\, a_2\, \ldots\, a_n \in L \qquad \text{where } a_i = b \text{ for some } 1 \leq i \leq n$$

Let Δ be another alphabet, called target, and $L_b \subseteq \Delta^*$ be the *image language* of b. The *substitution* of language L_b for b in string x produces a set of strings:

$$\{\, y \mid \ y = y_1\, y_2\, \ldots\, y_n \ \wedge \ \textbf{if } a_i \neq b \textbf{ then } y_i = a_i \textbf{ else } y_i \in L_b \,\}$$

which are a language over the alphabet $(\ \Sigma \setminus \{\, b \,\}\ ) \cup \Delta$. Notice that all the characters other than b are left unchanged.

By the usual approach, we can extend the substitution operation from one string to the whole source language, by applying it to every source sentence.

Example 2.26 (*Example 2.25 continued*) Resuming the case of a parameter list, the abstract syntax is:

$$i\, e\ (s\, e)^*\, f$$

and the substitutions to be applied are tabulated below:

abstract char.	*image language*
i	$L_i =$ procedure ⟨ procedure identifier ⟩ '('
e	$L_e =$ ⟨ parameter identifier ⟩
s	$L_s =$ ','
f	$L_f =$ ')'

Thus, the opening mark i is replaced with a string of language L_i, where the procedure identifier has to agree with the reference manual of the programming language. Clearly, the target languages of such substitutions depend on the 'syntax sugar' of the concrete language intended for. Notice that the four above substitutions are independent of one another and can be applied in any order. □

Example 2.27 (*identifiers with dash*) In certain programming languages, long mnemonic identifiers can be constructed by appending alphanumeric strings separated by a dash: thus, string LOOP3-OF-35 is a legal identifier. More precisely, the leftmost word, LOOP3, must initiate with a letter, the others may also initiate with a digit; and adjacent dashes, as well as a trailing dash, are forbidden.

As a first approximation, each language sentence is a nonempty list of elements e, which are alphanumeric words, separated by a dash:

$$e\ (\text{`_'}\ e)^*$$

However, the first word must be different from the others and may be taken to be the opening mark i of a possibly empty list:

$$i\ (\text{`-'}\ e)^*$$

By substituting to i the language $(A \ldots Z)\ (A \ldots Z \mid 0 \ldots 9)^*$, and to e the language $(A \ldots Z \mid 0 \ldots 9)^+$, the final r.e. is obtained. □

This is an overly simple instance of syntax design by abstraction and stepwise refinement, a method to be further developed now and after the introduction of grammars. Other language transformations are studied in Chap. 5.

2.4.3 Hierarchical or Precedence Lists

A frequently recurrent construct is a list such that each element is in turn a list of a different type. The first list is attached to level 1, the second to level 2, and so on. However, the present abstract paradigm, called hierarchical list, is restricted to lists having a bounded number of levels. The case of unbounded levels is studied later by using grammars, under the name of nested structures.

A hierarchical list is also called a *list with precedence*, because the list at level $l \geq 2$ bounds its elements more strongly than the list at level $l - 1$. In other words,

the elements at higher level, i.e., with higher precedence (or more priority), must be assembled into a list, and each such list becomes an element at the next lower level, i.e., with lower priority.

Of course, each level may have its own opening/closing marks and separator. Such delimiters are usually distinct level by level, in order to avoid confusion.

The structure of a hierarchical list with $k \geq 2$ levels is:

hierarchy of lists	*level*	*precedence/priority*
$list_1 = i_1\ list_2\ (\ s_1\ list_2\ )^*\ f_1$	1	lowest/min
$list_2 = i_2\ list_3\ (\ s_2\ list_3\ )^*\ f_2$	2	
$\ldots = \ldots$		
$list_k = i_k\ e_k\ (\ s_k\ e_k\ )^*\ f_k$	k	highest/max

In this schema, only the highest level may contain atomic elements, e_k. But a common variant permits, at any level l ($1 \leq l < k$), atomic elements e_l to occur side by side with lists of level $l + 1$. A few concrete examples follow.

Example 2.28 (*two hierarchical lists*) Structures in programming languages:

block of print instructions as in

$$\mathsf{begin}\ \mathsf{instr}_1\ ;\ \mathsf{instr}_2\ ;\ \ldots\ \mathsf{instr}_n\ \mathsf{end}$$

where instr is a print instruction, e.g., $\mathsf{WRITE}\ (\mathsf{var}_1, \mathsf{var}_2, \ldots, \mathsf{var}_n)$, i.e., a list (from Example 2.25); there are two levels ($k = 2$):

level 1 (low)	list of instructions instr separated by semicolon ';', opened by begin and closed by end
level 2 (high)	list of variables var separated by comma ',', with $i_2 =$ WRITE' (' and $f_2 =$ ') '

arithmetic expression without parentheses the precedences of arithmetic operators determine how many levels the hierarchy has; for instance, the operators '$\times$', '$\div$' and '$+$', '$-$' are layered on two levels ($k = 2$) and the expression:

$$\overbrace{\underbrace{3}_{\mathsf{n}}}^{\mathsf{term}} + \overbrace{\underbrace{5}_{\mathsf{n}} \times \underbrace{7}_{\mathsf{n}} \times 4 \ \ldots}^{\mathsf{term}} - \overbrace{8 \times 2 \div 5 \ \ldots\ldots\ldots\ldots}^{\mathsf{term}} + \overbrace{8}^{\mathsf{term}} + \overbrace{3}^{\mathsf{term}}$$

is a two-level list, with neither opening nor closing marks; at level 1, we find a list of terms $\mathsf{term} = list_2$, separated by signs '$+$' and '$-$', i.e., by low precedence operators; at level 2 we see a list of numbers $\mathsf{n} = e_2$, separated by signs '$\times$' and '$\div$', i.e., by high precedence operators; one may go further and introduce a level 3 with an exponentiation sign '$**$' as separator □

Of course, hierarchical structures are omnipresent in natural languages as well. Think of a list of nouns:

father, mother, son and daughter

Here we may observe a difference with respect to the abstract paradigm: the penultimate element uses a distinct separator and, possibly in order to warn the listener of an utterance that the list is approaching the end. Furthermore, the items in the list may be enriched by second-level qualifiers, such as a list of adjectives.

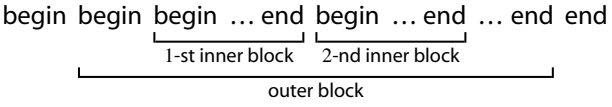
In all sorts of documents and written media, hierarchical lists are extremely common. For instance, a book is a list of chapters, separated by white pages, between a front and back cover. A chapter is a list of sections, a section is a list of paragraphs, and so on.

2.5 Context-Free Generative Grammars

We start the study of the family of context-free languages, which plays the central role in compilation. Initially invented by linguists for natural languages in the 1950s, context-free grammars have proved themselves extremely useful in computer science applications: all existing technical languages have been defined using such grammars. Moreover, since the early 1960s, efficient algorithms have been found to analyze, recognize, and translate sentences of context-free languages. This chapter presents the relevant properties of context-free languages, illustrates their application by many typical cases, and compares and combines them with regular languages. At last, we position the context-free model in the classical hierarchy of grammars and computational models due to Chomsky, and we briefly examine some context-sensitive models.

2.5.1 Limits of Regular Languages

Regular expressions are very practical for describing lists and related paradigms but fall short of the capacity needed to define other frequently occurring constructs. A relevant case are the block structures (or nested parentheses) present in many technical languages, schematized by:



Example 2.29 (*simple block structure*) Let the alphabet be shortened to $\{ b, e \}$ and consider a quite limited case of nested structure, such that all opening marks precede all closing marks. Clearly, opening/closing marks must have identical count:

$$L_1 = \left\{ b^n e^n \mid \quad n \geq 1 \right\}$$

We argue that language L_1 cannot be defined by a regular expression, but we have to defer the formal proof to a later section. In fact, since each string must have all letters b left of any letter e, either we write an overly general r.e. such as $b^+ e^+$, which unfortunately defines illegal strings like $b^3 e^5$, or we write a too restricted r.e. that exhaustively lists a finite sample of strings up to a bounded length. On the other hand, if we comply with the condition that the count of the two letters is the same by writing $(b e)^+$, illegal strings like $b e b e$ creep in. □

For defining this and other useful languages, regular or not, we move to the formal model of *generative* grammars.

2.5.2 Introduction to Context-Free Grammars

A generative *grammar* or *syntax*[15] is a set of simple rules that can be repeatedly applied in order to generate all and only the valid strings.

Example 2.30 (*palindromes*) The language L, over the alphabet $\Sigma = \{a, b\}$, is first defined, using the reversal operation, as:

$$L = \left\{ u\, u^R \mid \ u \in \Sigma^* \right\} = \{\varepsilon,\ a\,a,\ b\,b,\ a\,b\,b\,a,\ b\,a\,a\,b,\ \ldots,\ a\,b\,b\,b\,b\,a,\ \ldots\}$$

It contains strings having specular symmetry, called palindromes, of even-length. The following grammar G contains three rules:

$$pal \to \varepsilon \qquad pal \to a\,pal\,a \qquad pal \to b\,pal\,b$$

The arrow '$\to$' is a metasymbol, exclusively used to separate the left part of a rule from the right part.

To derive the strings, just replace the symbol *pal*, called *nonterminal*, with the right part of a rule, for instance

$$pal \Rightarrow a\,pal\,a \Rightarrow a\,b\,pal\,b\,a \Rightarrow a\,b\,b\,pal\,b\,b\,a \Rightarrow \ldots$$

The derivation process can be chained and terminates when the last string obtained no longer contains a nonterminal symbol. At that moment, the generation of the sentence is concluded. We complete the derivation:

$$a\,b\,b\,pal\,b\,b\,a \Rightarrow a\,b\,b\,\varepsilon\,b\,b\,a = a\,b\,b\,b\,b\,a$$

Incidentally, the language of palindromes is not regular.

[15] Sometimes the term *grammar* has a broader connotation than *syntax*, as when some rules for computing the meaning of sentences are added to the rules for enumerating them. When necessary, the intended meaning of the term will be made clear.

Next, we enrich the example into a list of palindromes separated by commas, exemplified by sentence '$a\,b\,b\,a,\ b\,b\,a\,a\,b\,b,\ a\,a$'. The grammar adds two list-generating rules to the previous ones:

$$\begin{array}{ll} list \to pal\ \text{`,'}\ list & pal \to \varepsilon \\ list \to pal & pal \to a\ pal\ a \\ & pal \to b\ pal\ b \end{array}$$

The top left rule says: the concatenation of palindrome, comma, and list produces a (longer) list; notice that the comma is between quotes, to stress that it is a terminal symbol like a and b. The other rule says that a list can be made of one palindrome.

Now there are two nonterminal symbols: *list* and *pal*. The former is termed *axiom* because it defines the intended language. The latter defines certain component substrings, also called *constituents* of the language, the palindromes. □

Example 2.31 (*metalanguage of regular expressions*) A regular expression that defines a language over a fixed terminal alphabet, say $\Sigma = \{a, b\}$, is a formula, i.e., a string over the alphabet $\Sigma_{\text{r.e.}} = \{a, b, \text{`}\cup\text{'}, \text{`}*\text{'}, \text{`}\varepsilon\text{'}, \text{`(', `)'}\}$, where the quotes around the r.e. metasymbols indicate their use as grammar terminals (concatenation '$\cdot$' could be added as well). Such formulas in turn are the sentences of the language $L_{\text{r.e.}}$ defined by the next grammar. Looking back to the definition of r.e. on p. 18, we write the rules of grammar $G_{\text{r.e.}}$:

$$\begin{array}{ll} 1\colon expr \to \text{`}\varepsilon\text{'} & 4\colon expr \to \text{`('}\ expr\ \text{`}\cup\text{'}\ expr\ \text{`)'} \\ 2\colon expr \to a & 5\colon expr \to \text{`('}\ expr\ expr\ \text{`)'} \\ 3\colon expr \to b & 6\colon expr \to \text{`('}\ expr\ \text{`)'}\text{`}*\text{'} \end{array}$$

where numbering is only for reference. A derivation is (quotes are omitted):

$$\begin{aligned} expr &\underset{4}{\Rightarrow} (expr \cup expr) \underset{5}{\Rightarrow} ((expr\ expr) \cup expr) \underset{2}{\Rightarrow} ((a\ expr) \cup expr) \\ &\underset{6}{\Rightarrow} \big(\big(a\ (expr)^*\big) \cup expr\big) \underset{4}{\Rightarrow} \big(\big(a\ ((expr \cup expr))^*\big) \cup expr\big) \\ &\underset{2}{\Rightarrow} \big(\big(a\ ((a \cup expr))^*\big) \cup expr\big) \underset{3}{\Rightarrow} \big(\big(a\ ((a \cup b))^*\big) \cup expr\big) \\ &\underset{3}{\Rightarrow} \big(\big(a\ ((a \cup b))^*\big) \cup b\big) = e \end{aligned}$$

Since the last string can be interpreted as an r.e. e, it denotes a second language L_e over alphabet Σ, which is the set of the strings starting with letter a, plus string b:

$$L_e = \{a,\ b,\ a\,a,\ a\,b,\ a\,a\,a,\ a\,b\,a,\ \ldots\}$$ □

A word of caution: this example displays two language levels, since the syntax defines certain strings to be understood as definitions of other languages. To avoid

terminological confusion, we say that the syntax $G_{\text{r.e.}}$ stays at the *metalinguistic* level, that is, over the linguistic level, or, equivalently, that the syntax is a *metagrammar*.

To set the two levels apart, it helps to consider the alphabets: at meta-level the alphabet is $\Sigma_{\text{r.e.}} = \{a,\ b,\ `\cup',\ `*',\ `\varepsilon',\ `(',\ `)'\}$, whereas the final language has alphabet $\Sigma = \{a,\ b\}$, devoid of metasymbols.

An analogy with human language may also clarify the issue. A grammar of Russian can be written, say, in English. Then it contains both Cyrillic and Latin characters. Here English is the metalanguage and Russian the final language, which only contains Cyrillic characters. Another example of language versus metalanguage is provided by *XML*, the metanotation used to define a variety of Web document types such as *HTML*.

Definition 2.32 (*context-free grammar*) A *context-free* (*CF*) (or type 2 or *BNF*[16]) grammar G is defined by four entities:

V	*nonterminal alphabet*, a set of symbols termed *nonterminals* or *metasymbols*
Σ	*terminal alphabet*, a set of symbols termed *terminals*, disjoint from the preceding set
P	a set of syntactic *rules* (or *productions*)
$S \in V$	a particular nonterminal termed *axiom*

A rule of set P is an ordered pair $X \to \alpha$, with $X \in V$ and $\alpha \in (V \cup \Sigma)^*$. Two or more rules ($n \geq 2$):

$$X \to \alpha_1 \qquad X \to \alpha_2 \qquad \ldots \qquad X \to \alpha_n$$

with the same left part X can be concisely grouped in:

$$X \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n \qquad \text{or} \qquad X \to \alpha_1 \cup \alpha_2 \cup \cdots \cup \alpha_n$$

We say that the strings $\alpha_1, \alpha_2, \ldots, \alpha_n$ are the *alternatives* of X. □

2.5.3 Conventional Grammar Representations

To prevent confusion, the metasymbols '$\to$', '|', '$\cup$', and ε should not be used for terminal or nonterminal symbols (Example 2.31 is a motivated exception). Moreover, the terminal and nonterminal alphabets must be disjoint, i.e., $\Sigma \cap V = \emptyset$.

[16] Type 2 comes from Chomsky classification (p. 105). Backus Normal Form, or also Backus–Naur Form, comes from the names of John Backus and Peter Naur, who pioneered the use of such grammars for defining programming languages.

Table 2.2 Different typographic styles for writing grammar rules

Nonterminal	Terminal	Example
A compound word between angle brackets, e.g., ⟨ if sentence ⟩	A word written as it is, without any special marks	⟨ if sentence ⟩ → if ⟨ cond ⟩ then ⟨ sentence ⟩ else ⟨ sentence ⟩
A word written as it is, without any special marks; it may not contain blank spaces; e.g., sentence or sentence_list	A word written in bold, in italic or quoted, e.g., **then**, *then* or 'then'	if_sentence → **if** cond **then** sentence **else** sentence or if_sentence → *if* cond *then* sentence *else* sentence or if_sentence → 'if' cond 'then' sentence 'else' sentence
An uppercase Latin letter	A word written as it is, without any special marks	$F \rightarrow$ if C then D else D

In professional and scientific practice, different styles are used to represent terminals and nonterminals, as specified in Table 2.2. In the first style, the grammar of Example 2.30 becomes:

$$\begin{cases} \langle \text{ sentence } \rangle & \rightarrow & \varepsilon \\ \langle \text{ sentence } \rangle & \rightarrow & a \ \langle \text{ sentence } \rangle \ a \\ \langle \text{ sentence } \rangle & \rightarrow & b \ \langle \text{ sentence } \rangle \ b \end{cases}$$

Alternative rules may be grouped together:

$$\langle \text{ sentence } \rangle \rightarrow \varepsilon \mid a \ \langle \text{ sentence } \rangle \ a \mid b \ \langle \text{ sentence } \rangle \ b$$

If a technical grammar is large, of the order of a few hundred rules, it should be written with care, to facilitate searching for specific definitions, making changes and cross referencing. The nonterminals should be identified by self-explanatory names, and the rules should be divided into sections and be numbered for reference. On the other hand, in very simple examples the third style of Table 2.2 is more suitable, that is, to have disjoint short symbols for terminals and nonterminals.

In this book, for simplicity we often adopt the following style:

- lowercase Latin letters near the beginning of the alphabet $\{ a, b, \ldots \}$ for terminal characters
- uppercase Latin letters $\{ A, B, \ldots, Z \}$ for nonterminal symbols
- lowercase Latin letters near the end of the alphabet $\{ \ldots, r, s, \ldots, z \}$ for strings over the alphabet Σ, i.e., including only terminals
- lowercase Greek letters $\{ \alpha, \beta, \ldots, \omega \}$ for the strings over the combined alphabet $V \cup \Sigma$

Table 2.3 Classification of grammar rule forms

Class	Description	Model
Terminal	Either *RP* contains only terminals or is the empty string	$\to u \mid \varepsilon$
Empty or *null*	*RP* is the empty string	$\to \varepsilon$
Initial or *axiomatic*	*LP* is the grammar axiom	$S \to$
Recursive	*LP* occurs in *RP*	$A \to \alpha A \beta$
Self-nesting or self-embedding	*LP* occurs in *RP* between nonempty strings	$A \to \alpha A \beta$
Left-recursive	*LP* is the prefix of *RP*	$A \to A \beta$
Right-recursive	*LP* is the suffix of *RP*	$A \to \beta A$
Left-right-recursive or *two-side recur.*	The two cases above together	$A \to A \beta A$
Copy, *renaming*, or *categorization*	*RP* consists of one nonterminal	$A \to B$
Identity	*LP* and *RP* are identical	$A \to A$
Linear	*RP* contains at most one nonterminal (the rest is terminal)	$\to u B v \mid w$
Left-linear or *type* 3	As the linear case above, but the nonterminal is prefix	$\to B v \mid w$
Right-linear or *type* 3	As the linear case above, but the nonterminal is suffix	$\to u B \mid w$
Homogeneous normal	*RP* consists of either $n \geq 2$ nonterminals or one terminal	$\to A_1 \ldots A_n \mid a$
Chomsky normal or *homogeneous of degree* 2	*RP* consists of either two nonterminals or one terminal	$\to B C \mid a$
Greibach normal or *real-time*	*RP* is one terminal possibly followed by nonterminals	$\to a \sigma \mid b$
Operator form	*RP* consists of two nonterminals separated by one terminal, which is called *operator* (more generally, *RP* is devoid of adjacent nonterminals)	$\to A a B$

Rule Types

In grammar studies, the rules may be classified depending on their form, with the aim of making the study of language properties more immediate. For future reference, in Table 2.3 we list some common rule forms along with their technical names. Each rule form is next schematized, with symbols adhering to the following stipulations: a and b are terminals; u, v, and w denote strings of terminals and may be empty; A, B, and C are nonterminals; α and β denote strings containing a mix of terminals and nonterminals, and may be empty; lastly, σ denotes a string of nonterminals.

The classification of Table 2.3 is based on the form of the right part *RP* of a rule, except the recursive rules, which may also consider the left part *LP*. We omit all the rule parts that are irrelevant for the classification. By the Chomsky classification,

the left-linear and right-linear forms are also known as *type* 3 *grammars*. Most rule forms listed in Table 2.3 will occur in the book, and the remaining ones are listed for general reference.

We will see that some of the grammar forms can be forced on any given grammar and leave the language unchanged. Such forms are called *normal*.

2.5.4 Derivation and Language Generation

We reconsider and formalize the notion of string derivation. Let $\beta = \delta A \eta$ be a string containing a nonterminal A, where δ and η are any strings, possibly empty. Let $A \to \alpha$ be a rule of grammar G and let $\gamma = \delta \alpha \eta$ be the string obtained by replacing the nonterminal A in β with the rule right part α.

The relation between two such strings is called *derivation*. We say that string β *derives* string γ for grammar G, and we write

$$\beta \underset{G}{\Rightarrow} \gamma$$

or simply $\beta \Rightarrow \gamma$ when the grammar name is understood. Rule $A \to \alpha$ is applied in such a derivation and string α *reduces* to nonterminal A.

Now consider a chain of derivations of length $n \geq 0$:

$$\beta_0 \Rightarrow \beta_1 \Rightarrow \cdots \Rightarrow \beta_n$$

which can be shortened to:

$$\beta_0 \overset{n}{\Rightarrow} \beta_n$$

If $n = 0$, for every string β we posit $\beta \overset{0}{\Rightarrow} \beta$, that is, the derivation relation is reflexive.

To express derivations of any length, we write:

$$\beta_0 \overset{*}{\Rightarrow} \beta_n \qquad \text{or} \qquad \beta_0 \overset{+}{\Rightarrow} \beta_n$$

if the length of the chain is $n \geq 0$ or $n \geq 1$, respectively.

The *language generated* or *defined* by a grammar G, *starting from nonterminal* A, is the set of the terminal strings that derive from nonterminal A in one or more steps, in formula:

$$L_A(G) = \left\{ x \in \Sigma^* \mid \ A \overset{+}{\Rightarrow} x \right\}$$

If the nonterminal is the axiom S, we have the *language generated by* G:

$$L(G) = L_S(G) = \left\{ x \in \Sigma^* \mid \ S \overset{+}{\Rightarrow} x \right\}$$

Sometimes, we need to consider the derivations that produce strings still containing nonterminals. A *string form* generated by a grammar G, starting from nonterminal $A \in V$, is a string $\alpha \in (V \cup \Sigma)^*$ such that it holds $A \overset{*}{\Rightarrow} \alpha$. In particular, if nonter-

minal A is the axiom, the string form is termed *sentential form*. Clearly, a sentence is a sentential form devoid of nonterminals.

Example 2.33 (*book structure*) The grammar G_l below defines the structure of a book. The book contains a front page f and a nonempty series of chapters, which is derived from nonterminal A. Each chapter starts with a title t and contains a nonempty series of lines l, which is derived from nonterminal B. Grammar G_l:

$$G_l \begin{cases} S \rightarrow f\,A \\ A \rightarrow A\,t\,B \mid t\,B \\ B \rightarrow l\,B \mid l \end{cases}$$

We list a few derivations. From nonterminal A, it derives the string form $t\,B\,t\,B$ and the string $t\,l\,l\,t\,l \in L_A\,(G_l)$. From the axiom S, it derives the sentential forms $f\,A\,t\,l\,B$ and $f\,t\,B\,t\,B$, and the sentence $f\,t\,l\,t\,l\,l\,l$.

The language generated starting from nonterminal B is $L_B\,(G_l) = l^+$. The language $L\,(G_l)$ generated by grammar G_l is also defined by r.e. $f\,\left(t\,l^+\right)^+$, and this shows that the language is in the family *REG*. In fact, language $L\,(G_l)$ is a case of an abstract hierarchical list. □

A language is *context-free* if there exists a context-free grammar that generates it. The empty set (or language) $\emptyset$ is context-free, too. The family of all the context-free languages (including $\emptyset$) is denoted by *CF*. Two grammars G and G' are *equivalent* if they generate the same language, i.e., $L\,(G) = L\,(G')$.

Example 2.34 (*equivalent grammars*) The next grammar G_{l2} is clearly equivalent to the grammar G_l of Example 2.33:

$$G_{l2} \begin{cases} S \rightarrow f\,X \\ X \rightarrow X\,t\,Y \mid t\,Y \\ Y \rightarrow l\,Y \mid l \end{cases}$$

since the only change affects the way nonterminals are identified. Also the following grammar G_{l3}:

$$G_{l3} \begin{cases} S \rightarrow f\,A \\ A \rightarrow A\,t\,B \mid t\,B \\ B \rightarrow B\,l \mid l \end{cases}$$

is equivalent to G_l. The only difference from G_1 is in the third row, which defines nonterminal B by a left-recursive rule, instead of the right-recursive rule of G_l. Clearly, the two collections of derivations of any length $n \geq 1$:

$$B \underset{G_l}{\overset{n}{\Longrightarrow}} l^n \qquad \text{and} \qquad B \underset{G_{l3}}{\overset{n}{\Longrightarrow}} l^n$$

generate the same language $L_B\,(G_l) = L_B\,(G_{l3}) = l^+$. □

2.5.5 Erroneous Grammars and Useless Rules

When writing a grammar, attention should be paid to make sure that all the nonterminals are well-defined and each of them effectively helps to produce some sentence. Without that, some rules may turn out to be unproductive.

A grammar G is called *clean* (or *reduced*) under the following conditions:

1. every nonterminal A is *reachable* from the axiom, that is, there exists a derivation $S \stackrel{*}{\Rightarrow} \alpha A \beta$
2. every nonterminal A is *well-defined*, that is, it generates a nonempty language, i.e., $L_A(G) \neq \emptyset$

It is often straightforward to check by inspection whether a grammar is clean. The following algorithm formalizes the checks.

2.5.5.1 Grammar Cleaning

The grammar cleaning algorithm operates in two phases. First, it pinpoints the undefined nonterminals and eliminates the rules that contain such nonterminals; second, it identifies the unreachable nonterminals and eliminates the rules containing them.

phase 1 Compute the set $DEF \subseteq V$ of the well-defined nonterminals. The set DEF is initialized with the nonterminals that occur in the terminal rules, the rules having a terminal string as their right part:

$$DEF := \left\{ A \mid \quad (A \to u) \in P \qquad \text{with } u \in \Sigma^* \right\}$$

Then this transformation is repeatedly applied until convergence is reached:

$$DEF := DEF \cup \{ B \mid \quad (B \to D_1 \ldots D_n) \in P \ \wedge \ \forall i \, D_i \in (\Sigma \cup DEF) \}$$

Notice that each symbol D_i $(1 \leq i \leq n)$ is a terminal in Σ or a nonterminal already present in DEF. At each iteration, two outcomes are possible:

- a new nonterminal is found that occurs as left part of a rule having as right part a string of terminals or well-defined nonterminals
- else the termination condition is reached

The nonterminals that belong to the complement set $V \setminus DEF$ are undefined and are eliminated by the algorithm together with any rule where they occur.

phase 2 A nonterminal is reachable from the axiom if, and only if, there exists a path in the following directed graph (called *reachability graph*), which represents

a binary relation (called *produce*) between nonterminals:

$$A \xrightarrow{produce} B$$

Such a relation says that a nonterminal A *produces* a nonterminal B if, and only if, there exists a rule $A \to \alpha B \beta$, where α and β are any strings.
Clearly, a nonterminal C is reachable from the axiom S if, and only if, in the graph there is a path directed from S to C. The unreachable nonterminals are the complement with respect to the nonterminal alphabet V. The algorithm eliminates them together with the rules where they occur, because they do not generate any sentence.

Notice that if a grammar generates the empty language $\emptyset$, all rules and nonterminals, including the axiom, are deleted by the algorithm. Therefore, no clean grammar exists to generate the empty language.

Quite often the requirement below is added to the cleanness conditions:

- the grammar must not permit any *circular derivation* $A \stackrel{+}{\Rightarrow} A$

In fact, such derivations are inessential because, if a string x is obtained by means of a circular derivation, e.g., $A \Rightarrow A \Rightarrow x$, it can also be obtained by means of the shorter noncircular derivation $A \Rightarrow x$. Moreover, circular derivations cause *ambiguity*, a negative phenomenon discussed in Sect. 2.5.10.

An identity rule (see Table 2.3) permits a one-step circular derivation. Thus, such a rule is inessential (and ambiguous) and should be canceled from the grammar. In this book, we assume grammars are always clean and noncircular.

Example 2.35 (*unclean grammars*)

- the grammar $\{ S \to a A S b,\ A \to b \}$ is unclean and does not generate any sentence, that is, it generates the empty set $\emptyset$
- grammar $\{ S \to a,\ A \to b \}$ has an unreachable nonterminal A; the same (finite) language $\{ a \}$ is generated by the clean grammar $\{ S \to a \}$
- grammar $\{ S \to a A S b \mid A,\ A \to S \mid c \}$ has a circular derivation $S \Rightarrow A \Rightarrow S$; the noncircular grammar $\{ S \to a S S b \mid c \}$ is equivalent
- notice that circularity may also come from the presence of an empty rule, as for instance in the following grammar fragment:

$$\begin{cases} X \to X\ Y \mid \ldots \\ Y \to \varepsilon \mid \ldots \end{cases} \qquad \text{circular derivation: } X \Rightarrow X\ Y \Rightarrow X$$

□

Finally, we observe that a grammar, although clean, may still contain redundant rules, as in the next example.

Example 2.36 (*grammar with redundant rules*)

$$\begin{cases} 1\colon\ S \to a\,A\,S\,b \\ 2\colon\ S \to a\,B\,S\,b \\ 3\colon\ S \to \varepsilon \end{cases} \qquad \begin{cases} 4\colon\ A \to c \\ 5\colon\ B \to c \end{cases}$$

One of the rule pairs (1, 4) and (2, 5), which generate exactly the same sentences, should be canceled. □

2.5.6 Recursion and Language Infinity

An essential property of most technical languages is to be infinite. We study how this property follows from the grammar rule forms. In order to generate an unbounded number of strings, the grammar has to derive strings of unbounded length. To this end, recursive rules are necessary, as next argued.

An n-step derivation of the form $A \overset{n}{\Rightarrow} x\,A\,y$ $(n \geq 1)$ is called *recursive*, or *immediately recursive* if $n = 1$. Similarly, the nonterminal A is called recursive. If the strings x or y are empty, the recursion is termed *left* or *right*, respectively.

Property 2.37 (language infinity) *Let grammar G be clean and devoid of circular derivations. Then language $L(G)$ is infinite if, and only if, grammar G has a recursive derivation.* □

Proof Clearly, without recursion any derivation of grammar G has a bounded length; therefore, language $L(G)$ would be finite.

Conversely, assume grammar G offers a recursive derivation $A \overset{n}{\Rightarrow} x\,A\,y$ $(n \geq 1)$, where, because of the hypothesis of noncircularity, not both strings x and y may be empty. Then there exists a derivation $A \overset{+}{\Rightarrow} x^m\,A\,y^m$ for every $m \geq 1$. By the cleanness hypothesis of grammar G, nonterminal A can be reached from the axiom through a derivation $S \overset{*}{\Rightarrow} u\,A\,v$, and by the same reason also nonterminal A derives at least one terminal string $A \overset{+}{\Rightarrow} w$. When we combine these derivations, we obtain a derivation for every $m \geq 1$:

$$S \overset{*}{\Rightarrow} u\,A\,v \overset{+}{\Rightarrow} u\,x^m\,A\,y^m\,v \overset{+}{\Rightarrow} u\,x^m\,w\,y^m\,v \qquad \text{with } m \geq 1$$

and such derivations in all generate an infinite language. □

In order to see whether a grammar has recursions, we reuse the binary relation *produce* on p. 40: a grammar does not have any recursions if, and only if, the relation graph does not contain any circuits.

We illustrate Property 2.37 by means of two grammars that generate a finite language and an infinite one (arithmetic expressions), respectively.

Example 2.38 (*finite language*) Consider the grammar reported below with its derivations:

$$\begin{cases} S \to a\,B\,c \\ B \to a\,b \mid C\,a \\ C \to c \end{cases} \qquad \begin{array}{l} \text{derivations} \\ S \Rightarrow a\,B\,c \Rightarrow a\,a\,b\,c \\ S \Rightarrow a\,B\,c \Rightarrow a\,C\,a\,c \Rightarrow a\,c\,a\,c \end{array}$$

This grammar does not have any recursion and allows just two derivations, which define the finite language $\{\,a\,a\,b\,c,\ a\,c\,a\,c\,\}$. □

The next example is a most common paradigm of so many artificial languages. It will be replicated and transformed over and over in the book.

Example 2.39 (*arithmetic expressions*) The grammar G below:

$$G = \big(\,\{E,\,T,\,F\},\,\{i,\,+,\,\times,\,\text{`)'},\,\text{`(`}\},\,P,\,E\,\big)$$

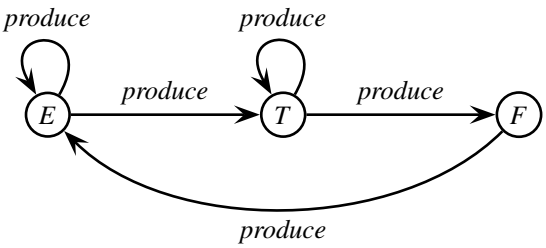
has this rule set P:

$$P \begin{cases} E \to E + T \mid T \\ T \to T \times F \mid F \\ F \to \text{`(`}\,E\,\text{`)'} \mid i \end{cases}$$

The language of G:

$$L(G) = \{\,i,\ i+i+i,\ i \times i,\ (i+i) \times i,\ \ldots\}$$

is the set of the arithmetic expressions over letter i, with the signs of sum '+' and product '×', and parentheses '( )'. Nonterminal F (factor) is recursive but not immediately. Nonterminals T (term) and E (expression) are immediately recursive, both to the left. Such properties are evident from the circuits in the (reachability) graph of the *produce* relation of G:



Since grammar G is clean and noncircular, language $L(G)$ is infinite. □

2.5.7 Syntax Tree and Canonical Derivation

The derivation process can be visualized as a syntax tree, for better legibility. A *tree* is a directed ordered graph that does not contain any circuit, such that every node pair is connected by exactly one directed path. An arc $N_1 \to N_2$ defines a *father* (or *parent*) *node–sibling* (or *child*) *node* relation, customarily drawn from top to bottom as in a genealogical tree. The siblings of a node are ordered from left to right. The *degree* (also called *arity*) of a node is the number of its siblings. A tree contains only one node without parent, termed *root*.

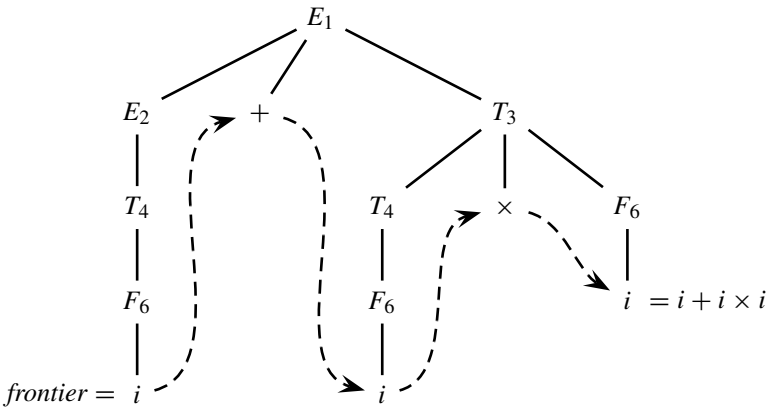
Consider a node N: the *subtree* of N is the tree that has node N as root and contains all the siblings of N, all of their siblings, etc., that is, all the *descendants* of N. The nodes without siblings are termed *leaves* or *terminal nodes*. The sequence of all the leaves, read from left to right, is the *frontier* of the tree. A *syntax tree* has the axiom as root and a sentence as frontier.

To construct a tree, consider a derivation. For each rule $A_0 \to A_1 A_2 \ldots A_r$ $(r \geq 1)$ used in the derivation, draw a small tree (or elementary tree) that has A_0 as root and $A_1 A_2 \ldots A_r$ as siblings, which may be terminals or nonterminals. If the rule is $A_0 \to \varepsilon$, draw one sibling labeled with ε. Such elementary trees are then pasted together, by uniting each nonterminal sibling, say A_i, with the root node that has the same label A_i, which is used to expand A_i in the subsequent derivation step.

Example 2.40 (*syntax tree*) The arithmetic expression grammar (Example 2.39) is reproduced in Fig. 2.1, and the rules are numbered for reference in the syntax tree construction. The derivation below:

$$\begin{aligned} E &\underset{1}{\Rightarrow} E+T \underset{2}{\Rightarrow} T+T \underset{4}{\Rightarrow} F+T \underset{6}{\Rightarrow} i+T \underset{3}{\Rightarrow} i+T\times F \\ &\underset{4}{\Rightarrow} i+F\times F \underset{6}{\Rightarrow} i+i\times F \underset{6}{\Rightarrow} i+i\times i \end{aligned} \tag{2.3}$$

corresponds to the following syntax tree:



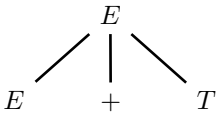
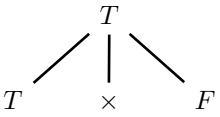
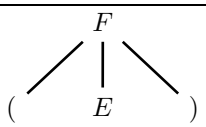
grammar rule	*elementary tree*
1: $E \to E + T$	
2: $E \to T$	
3: $T \to T \times F$	
4: $T \to F$	
5: $F \to \text{`(' } E \text{ `)'}$	
6: $F \to i$	

Fig. 2.1 Rules of arithmetic expressions (Example 2.39) and corresponding elementary trees

where the labels of the rules applied are displayed. The frontier, evidenced by the dashed line, yields the sentence $i + i \times i$.

Notice that the same tree represents other equivalent derivations, like the next one:

$$E \underset{1}{\Rightarrow} E + T \underset{3}{\Rightarrow} E + T \times F \underset{6}{\Rightarrow} E + T \times i \underset{4}{\Rightarrow} E + F \times i \qquad (2.4)$$
$$\underset{6}{\Rightarrow} E + i \times i \underset{2}{\Rightarrow} T + i \times i \underset{4}{\Rightarrow} F + i \times i \underset{6}{\Rightarrow} i + i \times i$$

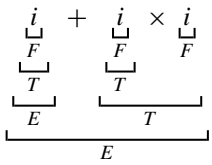
and many others, which differ from one another in the rule application order. Derivations (2.3) and (2.4) are termed *left* and *right*, respectively. □

A syntax tree of a sentence x can be encoded in a text, by enclosing each subtree between brackets (or parentheses), which may be subscripted with the subtree root

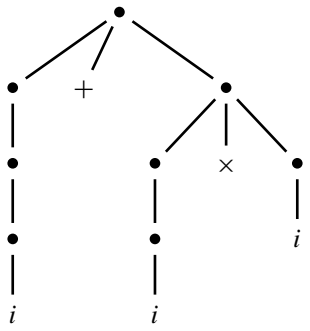
symbol.[17] Thus, the previous tree is encoded by the *parenthesized expression*:

$$\Big[\ [\,[\,[\,i\,]_F\,]_T\,\Big]_E + \Big[\ [\,[\,i\,]_F\,]_T \times [\,i\,]_F\ \Big]_T\Big]_E$$

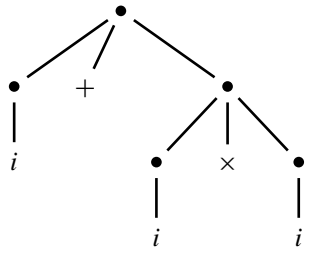
or by the graphical variant:



The representation can be simplified by dropping the nonterminal labels, thus obtaining a *skeleton tree* (left):



skeleton tree



condensed skeleton tree

or the corresponding parenthesized string (without subscripts):

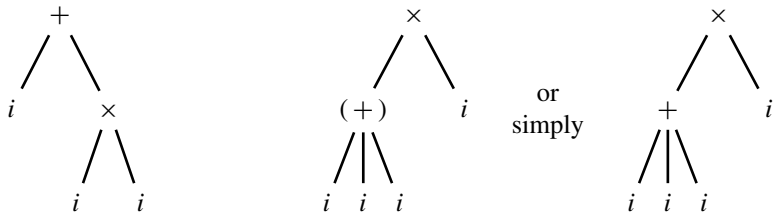
$$\Big[\ [\,[\,[\,i\,]\,]\,]\ +\ \Big[\ [\,[\,i\,]\,]\ \times\ [\,i\,]\ \Big]\Big]$$

A further simplification of the skeleton tree consists of shortening the nonbifurcating paths, thus resulting in the *condensed skeleton tree* (right). The nodes fused together represent the grammar copy rules. The corresponding parenthesized sentence is (without subscripts):

$$\Big[\ [\,i\,] + \Big[\ [\,i\,] \times [\,i\,]\ \Big]\Big]$$

Example 2.41 (*abstract syntax tree*) Within compilers, an arithmetic expression is sometimes represented by a simplified tree, called *abstract syntax tree*, wherein all the grammatical details, e.g., nonterminals, are omitted and only the operator structure is retained. Three possible representations are shown below:

[17] It is necessary to assume that the brackets are not in the terminal alphabet.



$i + i \times i$ $\qquad\qquad$ $(i + i + i) \times i$

The tree nodes are labeled by operators, and their parent–sibling relationship directly models the expression structure. The subexpression parentheses are optional in the abstract tree, whereas they are necessary for correctly computing the value when the expression is in string form (above right).

Later we will see that also regular expressions can be usefully represented by abstract syntax trees. □

Some approaches tend to view a grammar as a device for assigning structure to sentences. From this standpoint, a grammar defines a set of syntax trees, that is, a tree language instead of a string language.[18]

2.5.7.1 Left and Right Derivations

A derivation of $p \geq 1$ steps:

$$\beta_0 \Rightarrow \beta_1 \Rightarrow \cdots \Rightarrow \beta_p$$

where

$$\beta_i = \delta_i A_i \eta_i \qquad \text{and} \qquad \beta_{i+1} = \delta_i \alpha_i \eta_i \qquad \text{with } 0 \leq i \leq p-1$$

is called *left* (also *leftmost*) or *right* (also *rightmost*), if it holds $\delta_i \in \Sigma^*$ or $\eta_i \in \Sigma^*$, respectively, for every $0 \leq i \leq p-1$.

In words, at each step a left derivation or a right one expands the rightmost nonterminal or the leftmost one, respectively. A letter l or r may be subscripted to the arrow sign '$\Rightarrow$', to make explicit the derivation order.

Notice there exist other derivations that are neither left nor right, because the nonterminal symbol expanded is not always either leftmost or rightmost, or because at some step it is leftmost and at some other step it is rightmost.

[18] For an introduction to the theory of tree languages, their grammars, and automata, see [3,4].

Returning to Example 2.40 on p. 43, derivation (2.3) is left and is denoted by $E \overset{+}{\underset{l}{\Rightarrow}} i + i + i \times i$, derivation (2.4) is right, whereas the derivation (2.5) below:

$$\begin{aligned} E &\underset{l,\,r}{\Longrightarrow} E + T \underset{r}{\Rightarrow} E + T \times F \underset{l}{\Rightarrow} T + T \times F \Rightarrow T + F \times F \\ &\underset{r}{\Rightarrow} T + F \times i \underset{l}{\Rightarrow} F + F \times i \underset{r}{\Rightarrow} F + i \times i \underset{l,\,r}{\Longrightarrow} i + i \times i \end{aligned} \tag{2.5}$$

is neither left nor right. The three derivations have the same syntax tree. The example actually illustrates an essential property of context-free grammars.

Property 2.42 (left and right derivations) *Every sentence of a context-free grammar can be generated by a left derivation and by a right one.* □

Therefore, it does not do any harm to use just left (or right) derivations in the definition (on p. 37) of the language generated by a grammar.

On the other hand, more complex grammar types, as the context-sensitive grammars, do not share this nice property, which is quite important for obtaining efficient algorithms for string recognition and parsing, as we will see.

2.5.8 Parenthesis Languages

Many artificial languages include parenthesized or nested structures, made of matching pairs of *opening/closing marks*. Any such occurrence may in turn contain other matching pairs. The marks are abstract elements that have different concrete representations in distinct settings. Thus, the block structures of Pascal are enclosed within begin … end, while in the C language curly brackets '{ }' are frequently used.

A massive use of parenthesis structures characterizes the mark-up language XML, which offers the possibility of inventing new matching pairs. An example is the pair ⟨ title ⟩ . . . ⟨ / title ⟩, used to delimit the document title. Similarly, in the notation of LaTeX, a mathematical formula is enclosed between the marks `\begin{equation} ... \end{equation}`.

When a parenthesis structure may contain another one of the same kind, it is called *self-nesting*. Self-nesting is potentially unbounded in artificial languages, whereas in natural languages its use is moderate, because it causes a difficulty of comprehension by breaking the flow of discourse. Next comes an example of complex German sentence,[19] with as many as three nested relative clauses:

der Mann der die Frau die das Kind das die Katze füttert sieht liebt schläft

[19] The man who loves the woman (who sees the child (who feeds the cat)) sleeps.

$$
\begin{cases}
S \rightarrow \langle \text{OBJECT} \rangle \\
\langle \text{OBJECT} \rangle \rightarrow \text{`\{ \}'} \mid \text{`\{'} \ \langle \text{MEMBERS} \rangle \ \text{`\}'} \\
\langle \text{MEMBERS} \rangle \rightarrow \langle \text{PAIR} \rangle \mid \langle \text{PAIR} \rangle \ \text{,} \ \langle \text{MEMBERS} \rangle \\
\langle \text{PAIR} \rangle \rightarrow \langle \text{STRING} \rangle \ \text{:} \ \langle \text{VALUE} \rangle \\
\langle \text{VALUE} \rangle \rightarrow \langle \text{STRING} \rangle \mid \langle \text{OBJECT} \rangle \mid \langle \text{ARRAY} \rangle \mid \text{num} \mid \text{bool} \\
\langle \text{STRING} \rangle \rightarrow \text{`` ''} \mid \text{``} \langle \text{CHARS} \rangle \text{''} \\
\langle \text{ARRAY} \rangle \rightarrow \text{`[ ]'} \mid \text{`['} \ \langle \text{ELEMENTS} \rangle \ \text{`]'} \\
\langle \text{ELEMENTS} \rangle \rightarrow \langle \text{VALUE} \rangle \mid \langle \text{VALUE} \rangle \ \text{,} \ \langle \text{ELEMENTS} \rangle \\
\langle \text{CHARS} \rangle \rightarrow \langle \text{CHAR} \rangle \mid \langle \text{CHAR} \rangle \ \langle \text{CHARS} \rangle \\
\langle \text{CHAR} \rangle \rightarrow \text{char}
\end{cases}
$$

Fig. 2.2 Official *JSON* grammar: uppercase and lowercase words denote nonterminals and terminals, respectively, and quotes mark metasymbols here used as terminals

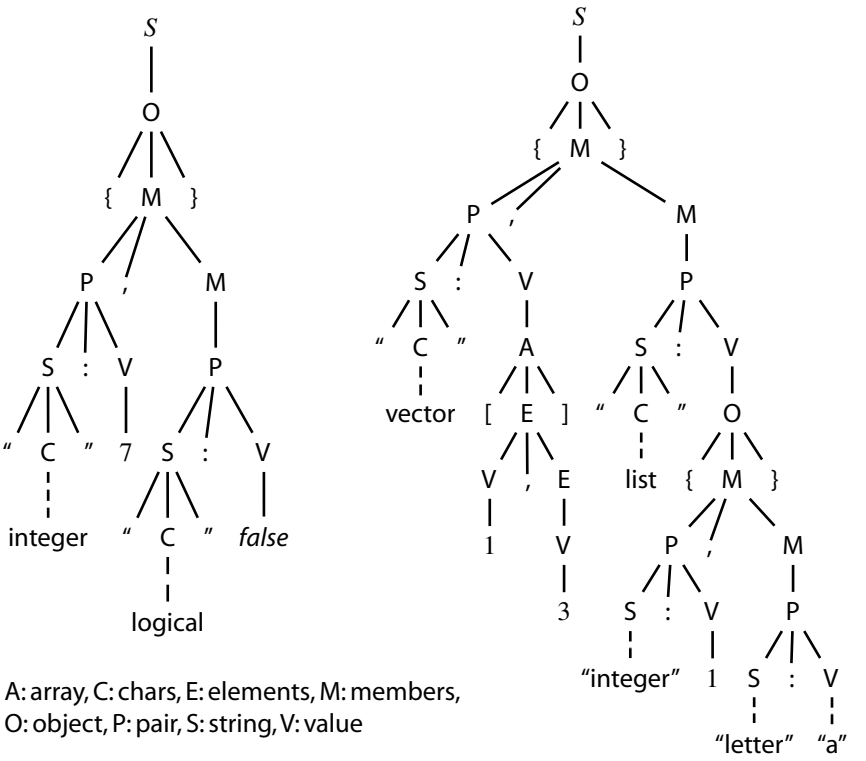
Returning to technical applications, in Fig. 2.2 we reproduce the syntax of *JSON*, a recent language that features recursive nesting of two different parenthesized constructs:

Example 2.43 (*JSON grammar*) The *JavaScript Object Notation* (*JSON*) is a lightweight data-interchange format, which is easy for humans to read and write, and efficient for machines to parse and generate. The grammar of *JSON* is shown in Fig. 2.2. Symbol num is a numeric constant, and symbol bool stands for *true*, *false*, or *null*. Symbol char denotes any 8-bit character, except those occurring as terminals in the grammar rules.

The syntactic paradigm of *JSON* is that of the nested lists of arbitrary depth, enclosed in curly (list of pairs) or square (array list) brackets, with elements separated by comma. Four sample *JSON* sentences are:

```
{"integer":7,"logical":false}
{"vector":[1,3],"list":{"integer":1,"letter":"a"}}
{"matrix2x2":[[1,3],[5,8]]}
{"timetable":{"airport":["JFK","LAX"],"dept.h":[13,20]}}
```

Clearly, *JSON* is meant to represent so-called *semi-structured data*, i.e., large data sets of mixed type. Here are the (partially condensed) syntax trees of the two top sentences, with node names shortened to their initials:



Some subtrees are condensed (dashed edges), e.g., chars and string. □

By abstracting from any concrete representation and content of parenthesized constructs, the parenthesis paradigm is known as *Dyck language*. The terminal alphabet contains one or more pairs of opening/closing marks. An example is the alphabet $\Sigma = \{\text{`(', `)', `[', `]'}\}$ and the parenthesis sentence '[ ] (([ ] ( )))'.

Dyck sentences are characterized by the following *cancellation rule* that checks whether parentheses are well nested: given a string, repeatedly substitute the empty string ε for a pair of adjacent matching parentheses:

$$(\,) \Rightarrow \varepsilon \qquad \text{and} \qquad [\,] \Rightarrow \varepsilon$$

and obtain another string. Repeat until the transformation does not apply any longer. The original string is correct if, and only if, the last one is empty.

Example 2.44 (*Dyck language*) To aid the eye, we encode the left (open) parentheses as a, b, …, and the right (closed) ones as a', b', …. With the alphabet

$\Sigma = \{ a, a', b, b' \}$, the Dyck language is generated by the grammar below (left):

$$\begin{cases} S \to a\, S\, a'\, S \\ S \to b\, S\, b'\, S \\ S \to \varepsilon \end{cases}$$

valid Dyck string (but invalid for language L_1)

$$\underbrace{a\ a\ \underbrace{a\ a'}\ a'}_{1^{\text{st}}\text{ nest}}\ \underbrace{a\ a\ \underbrace{a\ a'}\ a'\ a'}_{2^{\text{nd}}\text{ nest chained to }1^{\text{st}}}\ a'$$

Notice that we need nonlinear rules (the first two) to generate the Dyck language. To see why, compare with the language L_1 of Example 2.29 on p. 31. Language L_1, by recoding its alphabet $\{ b, e \}$ as $\{ a, a' \}$, is strictly included in the Dyck language, since it disallows any string with two or more nests concatenated, as there are in the string above (right). Such sentences have a branching syntax tree that requires nonlinear rules for its derivation. □

Another way of constraining a grammar to produce nested constructs is to force each rule to be parenthesized.

Definition 2.45 (*parenthesized grammar*) Let $G = (V, \Sigma, P, S)$ be a grammar with an alphabet Σ without parentheses. The parenthesized grammar G_p has an extended alphabet $\Sigma \cup \{ \text{`(', `)'} \}$ and these rules:

$$A \to \text{`('}\ \alpha\ \text{`)'} \qquad \text{where } A \to \alpha \text{ is a rule of } G$$

The grammar is *distinctly parenthesized* if every rule has the form:

$$A \to \text{`}_A\text{('}\ \alpha\ \text{`)}_A\text{'} \qquad B \to \text{`}_B\text{('}\ \beta\ \text{`)}_B\text{'} \qquad \ldots$$

where `$_A$(' and `)$_A$', etc, are parentheses subscripted (on the left or right for better legibility) with the nonterminal name A, etc. □

Each sentence produced by such grammars exhibits a parenthesized structure.

Example 2.46 (*parenthesis grammar*) The parenthesized version of the grammar for palindrome lists (p. 33) is:

$$\begin{cases} list \to \text{`('}\ pal\ \text{`,'}\ list\ \text{`)'} \\ list \to \text{`('}\ pal\ \text{`)'} \end{cases} \qquad \begin{cases} pal \to \text{`('}\ \text{`)'} \\ pal \to \text{`('}\ a\ pal\ a\ \text{`)'} \\ pal \to \text{`('}\ b\ pal\ b\ \text{`)'} \end{cases}$$

Now, the original sentence $a\,a$ is parenthesized as `( ( a ( ) a ) )' or is distinctly parenthesized as `$_{list}$($_{pal}$(a $_{pal}$( )$_{pal}$ a)$_{pal}$)$_{list}$'. □

A notable effect of the presence of parentheses is to allow a simpler checking of string correctness, to be discussed in Chap. 4.

2.5.9 Regular Composition of Context-Free Languages

If the basic operations of regular languages, i.e., union, concatenation, and star, are applied to context-free languages, the result remains a member of the *CF* family, to be shown next.

Let $G_1 = (\Sigma_1, V_1, P_1, S_1)$ and $G_2 = (\Sigma_2, V_2, P_2, S_2)$ be grammars that define languages L_1 and L_2, respectively. We may safely assume that their non-terminal sets are disjoint, i.e., $V_1 \cap V_2 = \emptyset$. Moreover, we stipulate that symbol S, to be used as axiom of the grammar under construction, is not used by either grammar, i.e., $S \notin (V_1 \cup V_2)$.

union The grammar G of language $L_1 \cup L_2$ contains all the rules of G_1 and G_2, plus the initial rules $S \to S_1 \mid S_2$. In formula it is:

$$G = (\Sigma_1 \cup \Sigma_2, \{S\} \cup V_1 \cup V_2, \{S \to S_1 \mid S_2\} \cup P_1 \cup P_2, S)$$

concatenation The grammar G of language $L_1 L_2$ contains all the rules of G_1 and G_2, plus the initial rule $S \to S_1 S_2$. It is:

$$G = (\Sigma_1 \cup \Sigma_2, \{S\} \cup V_1 \cup V_2, \{S \to S_1 S_2\} \cup P_1 \cup P_2, S)$$

star The grammar G of language L_1^* contains all the rules of G_1 and the initial rules $S \to S S_1 \mid \varepsilon$.

cross From the identity $L^+ = L \cdot L^*$, the grammar of language L^+ can be written by applying the concatenation construction to L and L^*, but it is better to produce the grammar directly. The grammar G of language L_1^+ contains all the rules of G_1 and the initial rules $S \to S S_1 \mid S_1$.

From all this, we have:

Property 2.47 (closure of CF) *The family* CF *of context-free languages is closed with respect to union, concatenation, star, and cross.* □

Example 2.48 (*union of languages*) The language L below:

$$L = \underbrace{\left\{ a^i b^i c^* \mid \ i \geq 0 \right\}}_{\text{component } L_1} \cup \underbrace{\left\{ a^* b^i c^i \mid \ i \geq 0 \right\}}_{\text{component } L_2} = L_1 \cup L_2$$

has sentences of the form $a^i b^j c^k$ $(i, j, k \geq 0)$ with $i = j$ or $j = k$, such as

$$a^5 b^5 c^2 \qquad a^5 b^5 c^5 \qquad b^5 c^5$$

The grammars G_1 and G_2 for the component languages L_1 and L_2 are straightforward:

$$G_1 \begin{cases} S_1 \to X\,C \\ X \to a\,X\,b \mid \varepsilon \\ C \to c\,C \mid \varepsilon \end{cases} \qquad G_2 \begin{cases} S_2 \to A\,Y \\ Y \to b\,Y\,c \mid \varepsilon \\ A \to a\,A \mid \varepsilon \end{cases}$$

We just add to the rules of G_1 and G_2, the alternatives $S \to S_1 \mid S_2$, to trigger a derivation with either grammar G_1 or G_2. □

A word of caution: if the nonterminal sets overlap, this construction produces a grammar that generates a language typically larger than the union. To see why, replace grammar G_2 with the trivially equivalent grammar G_2':

$$G_2' \begin{cases} S_2' \to A\,X \\ X \to b\,X\,c \mid \varepsilon \\ A \to a\,A \mid \varepsilon \end{cases}$$

Then the putative union grammar, i.e., $\{ S \to S_1 \mid S_2' \} \cup P_1 \cup P_2'$, would also allow hybrid derivations that use rules from both grammars, and for instance, it would generate string $a\,b\,c\,b\,c$, which is not in the union language.

Notably, Property 2.47 holds for both families *REG* and *CF*, but only the former is closed under intersection and complement, to be seen later.

Grammar of Mirror Language

When one examines the effect of string reversal on the sentences of a *CF* language, one immediately sees that the family *CF* is closed with respect to reversal (the same as family *REG*). Given a grammar, the rules that generate the mirror language are obtained by reversing the right part of every rule.

2.5.10 Ambiguity in Grammars

In natural language, the common linguistic phenomenon of ambiguity shows up when a sentence has two or more meanings. Ambiguity is of two kinds, semantic or syntactic. Semantic ambiguity occurs in the clause a hot spring, where the noun denotes either a coil or a season. A case of syntactic (or structural) ambiguity is the clause half baked chicken, which has different meanings depending on the structure assigned:

$$\big[\,[\,\text{half baked}\,]\,\text{chicken}\,\big] \quad \text{or} \quad \big[\,\text{half}\,[\,\text{baked chicken}\,]\,\big]$$

Although ambiguity may cause misunderstanding in human communication, negative consequences are counteracted by the availability of nonlinguistic clues for choosing the intended interpretation.

Artificial languages too can be ambiguous, but the phenomenon is less deep than in human languages. In most situations, ambiguity is a defect to be removed or counteracted.

A *sentence x* defined by grammar G is syntactically *ambiguous* if it is generated with two different syntax trees. Then the *grammar* too is called *ambiguous*.

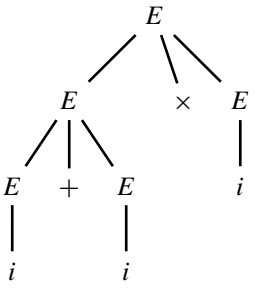
Example 2.49 (*ambiguity of a sentence*) Consider again the arithmetic expression language of Example 2.39 on p. 42, but define it with a different grammar G', equivalent to the previous one:

$$G' : E \to E + E \mid E \times E \mid \text{`(' } E \text{`)'} \mid i$$

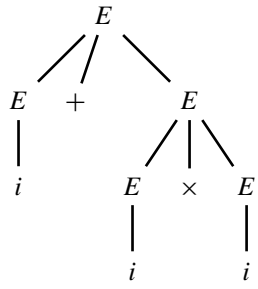
The left derivations:

$$E \Rightarrow E \times E \Rightarrow E + E \times E \Rightarrow i + E \times E \Rightarrow i + i \times E \Rightarrow i + i \times i \quad (2.6)$$
$$E \Rightarrow E + E \Rightarrow i + E \Rightarrow i + E \times E \Rightarrow i + i \times E \Rightarrow i + i \times i \quad (2.7)$$

generate the same sentence, but with different trees:



tree of derivation (2.6)



tree of derivation (2.7)

Therefore, sentence $i + i \times i$ is ambiguous, as well as grammar G'.

Now pay attention to the meaning of the two readings of the same expression. The left tree interprets the sentence as $(i + i) \times i$, the right one as $i + (i \times i)$. The latter interpretation is likely to be preferable, as it agrees with the traditional operator precedence. Another ambiguous sentence is $i + i + i$, which has two trees that differ in the subexpression association order: from left to right, or the other way. Thus, grammar G' is ambiguous.

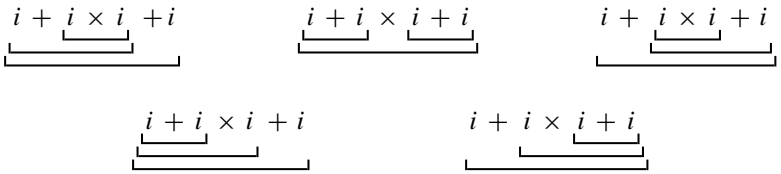
A major defect of this grammar is that it does not force the expected traditional precedence of multiplication over addition. □

On the other hand, for the grammar G of Example 2.39 on p. 42, each sentence has only one left derivation, therefore all the sentences generated by G are unambiguous, and grammar G is unambiguous as well.

It may be noticed that the new grammar G' is smaller than the old one G. This manifests a frequent property of ambiguous grammars, which is their conciseness with respect to the equivalent unambiguous ones. In special situations, when one wants the simplest grammar possible for a language, ambiguity may be tolerated, but, in general, conciseness cannot be bought at the cost of equivocation.

The *degree of ambiguity* of a sentence x of a language $L(G)$ is the number of distinct syntax trees deriving the sentence. For a grammar, the degree of ambiguity is the maximum degree of any ambiguous sentence. The next derivation shows that such a degree may be unbounded.

Example 2.50 (*Example 2.49 continued*) Sentences $i + i + i$ and $i + i \times i + i$ have ambiguity degree 2 and 5, resp. Here are the five skeleton trees of the latter:



Clearly, longer sentences cause the ambiguity degree to grow unbounded. □

An important practical problem is to check a given grammar for ambiguity. This is an example of a seemingly simple problem, for which no general algorithm exists: the problem is undecidable.[20] This means that any general procedure for checking a grammar for ambiguity may be forced to examine longer and longer sentences, without ever reaching the certainty of the answer. On the other hand, for a specific grammar, with some ingenuity one can often prove nonambiguity by applying some form of inductive reasoning.

In practice, this is unnecessary, and usually two approaches suffice. First, a small number of rather short sentences, which cover all grammar rules, are tested, by constructing their syntax trees and checking that they are unique. If the sentences pass the test, one has to proceed to check whether the grammar complies with certain conditions characterizing the so-called deterministic context-free languages, to be fully studied in Chap. 4. Such conditions are sufficient, although not necessary, to ensure nonambiguity.

Even better is to prevent the problem when a grammar is designed, by avoiding some common pitfalls to be explained next.

2.5.11 Catalogue of Ambiguous Forms and Remedies

Following the definition, an ambiguous sentence displays two or more structures, each one possibly associated with a sensible interpretation. Though ambiguity cases

[20] A proof can be found in [9].

are abundant in natural languages, communication clarity is not seriously impaired because the sentences are uttered or written in a living context made of gestures, intonation, presuppositions, etc, which helps select the interpretation intended by the author. On the contrary, most existing compilers cannot tolerate ambiguity, because they are not capable to select the correct meaning out of the many that would be possible for an ambiguous sentence in a programming language, with the negative consequence of an unpredictable behavior of the program.

Now we classify the most common ambiguity types, and we show how to remove them by modifying the grammar, or in some cases the language.

2.5.11.1 Ambiguity from Bilateral Recursion

A nonterminal symbol A is bilaterally (two-side) recursive if it is both left and right-recursive, i.e., $A \overset{+}{\Rightarrow} A\,\gamma$ and $A \overset{+}{\Rightarrow} \beta\,A$. We distinguish the cases when the two derivations are produced by the same rule or by different rules.

Example 2.51 (*bilateral recursion from same rule*) The grammar G_1 below:

$$G_1 : E \to E + E \mid i$$

generates string $i + i + i$ with two different left derivations:

$$E \Rightarrow E + E \Rightarrow E + E + E \Rightarrow i + E + E \Rightarrow i + i + E \Rightarrow i + i + i$$
$$E \Rightarrow E + E \Rightarrow i + E \Rightarrow i + E + E \Rightarrow i + i + E \Rightarrow i + i + i$$

Ambiguity comes from the absence of a fixed generation order for the string, i.e., from the left or right. When looking at the intended meaning of the string as an arithmetic formula, this grammar does not specify the application order of arithmetic operations.

In order to remove ambiguity, observe that this language is a list with separators, i.e., $L\,(G_1) = i\,(+\,i\,)^*$. This is a paradigm we are able to define with a right-recursive rule $E \to i + E \mid i$ or with a left-recursive one $E \to E + i \mid i$. Both rules are unambiguous. □

Example 2.52 (*left and right recursions in different rules*) A second case of bilateral recursive ambiguity is grammar G_2:

$$G_2 : A \to a\,A \mid A\,b \mid c$$

This language too is regular: $L(G_2) = a^*\,c\,b^*$. It is the concatenation of two lists a^* and b^*, with a letter c interposed. Ambiguity disappears if the two lists are derived by separate rules, thus suggesting the grammar (left):

$$\begin{cases} S \to A\,c\,B \\ A \to a\,A \mid \varepsilon \\ B \to b\,B \mid \varepsilon \end{cases} \qquad \begin{cases} S \to a\,S \mid X \\ X \to X\,b \mid c \end{cases}$$

An alternative remedy is to decide that the first list should be generated before the second one (or conversely), as done above (right). □

We remark that a double recursion on the same nonterminal does not cause ambiguity by itself, provided that the two recursions are not left and right. Observe the grammar:

$$S \to +SS \mid \times SS \mid i$$

that defines the so-called prefix polish expressions with the signs of addition and multiplication (further studied in Chap. 5), such as the expression $++ii \times ii$. Notwithstanding two rules are doubly recursive, the grammar is not ambiguous, and as a matter of fact one recursion is right but the other is not left.

2.5.11.2 Ambiguity from Union

If two languages $L_1 = L(G_1)$ and $L_2 = L(G_2)$ share some sentence, that is, their intersection is not empty, the grammar G of the union language, constructed as explained on p. 51, generates such sentence with two different trees, respectively using the rules of G_1 or G_2, and is therefore ambiguous.

Example 2.53 (*union of overlapping languages*) In language and compiler design, there are various causes for overlap:

1. When one wants to single out a special pattern that requires specific processing, within a general class of phrases. Consider the additive arithmetic expressions with constants C and variables i. A grammar is:

$$\begin{cases} E \to E + C \mid E + i \mid C \mid i \\ C \to 0 \mid 1D \mid \cdots \mid 9D \\ D \to 0D \mid \cdots \mid 9D \mid \varepsilon \end{cases}$$

   Now, assume the compiler has to single out such expressions as $i + 1$ or $1 + i$, because they have to be translated to machine code, by using increment instead of addition. To this end, we add the rules:

$$E \to i + 1 \mid 1 + i$$

   Unfortunately, the new grammar is ambiguous, since a sentence like $1 + i$ is generated by the original rules, too.
2. When the same operator is overloaded, that is, used with different meanings in different constructs. In the Pascal language, a sign '+' denotes both addition in:

$$E \to E + T \mid T \qquad T \to V \qquad V \to \ldots$$

and set union in:

$$E_{ins} \to E_{ins} + T_{ins} \mid T_{ins} \qquad\qquad T_{ins} \to V$$

To remove such ambiguities, we need a severe grammar surgery: either the two ambiguous constructs are made disjoint or they are fused together. Disjoining the constructs is unfeasible in the previous examples, because string '1' cannot be removed from the set of integer constants derived from nonterminal C. To enforce a special treatment of value '1', if one accepts a syntactic change to the language, it suffices to add the operator *inc* (for increment by 1) and replace rule $E \to i + 1 \mid 1 + i$ with rule $E \to inc\, i$.

In the latter example, ambiguity is semantic and is caused by the double meaning (polysemy) of the operator '+'. A remedy is to collapse together the rules for arithmetic and set expressions, generated from nonterminals E and E_{ins}, respectively, thus giving up a syntax-based separation. The semantic analyzer will take care of it. Alternatively, if modifications are permissible, one may replace the sign '+' by a character '∪' in the set expressions. □

In the example below, removing the overlapping constructs actually succeeds.

Example 2.54 (*disambiguation of grammars* [10])

1. The grammar G below (left):

$$G \begin{cases} S \to b\,S \mid c\,S \mid D \\ D \to b\,D \mid c\,D \mid \varepsilon \end{cases} \qquad\qquad \begin{array}{l} S \overset{+}{\Rightarrow} b\,b\,c\,S \Rightarrow b\,b\,c\,D \Rightarrow b\,b\,c \\ S \Rightarrow D \overset{+}{\Rightarrow} b\,b\,c\,D \Rightarrow b\,b\,c \end{array}$$

   is ambiguous since $L(G) = \{b,\ c\}^* = L_D(G)$. In fact, the two derivations above (right) produce the same result. By deleting the rules of D, which are redundant, we have $S \to b\,S \mid c\,S \mid \varepsilon$, no longer ambiguous.
2. The grammar below (left):

$$\begin{cases} S \to B \mid D \\ B \to b\,B\,c \mid \varepsilon \\ D \to d\,D\,e \mid \varepsilon \end{cases} \qquad\qquad \begin{cases} S \to B \mid D \mid \varepsilon \\ B \to b\,B\,c \mid b\,c \\ D \to d\,D\,e \mid d\,e \end{cases}$$

   where nonterminals B and D respectively generate strings $b^n\,c^n$ and $d^n\,e^n$, with $n \geq 0$, has just one ambiguous sentence: the empty string ε. A remedy is to generate ε directly from the axiom, as done above (right). □

2.5.11.3 Ambiguity from Concatenation

Concatenating languages may cause ambiguity, if a suffix of a sentence of language one is also a prefix of a sentence of language two.

Recall that the concatenation grammar G of $L_1 L_2$ (p. 51) contains the rule $S \to S_1 S_2$ in addition to the rules (by hypothesis not ambiguous) of G_1 and G_2. Ambiguity arises in G if the following sentences exist in the languages:

$$u' \in L_1 \qquad u'\, v \in L_1 \qquad v\, z'' \in L_2 \qquad z'' \in L_2 \qquad \text{with } v \neq \varepsilon$$

Then the string $u'\, v\, z''$ of language $L_1 \cdot L_2$ is ambiguous, via the derivations:

$$S \Rightarrow S_1\, S_2 \overset{+}{\Rightarrow} u'\, S_2 \overset{+}{\Rightarrow} u'\, v\, z'' \qquad\qquad S \Rightarrow S_1\, S_2 \overset{+}{\Rightarrow} u'\, v\, S_2 \overset{+}{\Rightarrow} u'\, v\, z''$$

Example 2.55 (*concatenation of Dyck languages*) Consider the concatenation $L = L_1\, L_2$ of the Dyck languages (p. 49) L_1 and L_2 over alphabets $\{a,\ a',\ b,\ b'\}$ and $\{b,\ b',\ c,\ c'\}$, respectively. A sentence of L is $a\, a'\, b\, b'\, c\, c'$. The standard grammar G of L is:

$$G \begin{cases} S \ \to\ S_1\, S_2 \\ S_1 \ \to\ a\, S_1\, a'\, S_1 \ \mid\ b\, S_1\, b'\, S_1 \ \mid\ \varepsilon \\ S_2 \ \to\ b\, S_2\, b'\, S_2 \ \mid\ c\, S_2\, c'\, S_2 \ \mid\ \varepsilon \end{cases}$$

For grammar G, sentence $a\, a'\, b\, b'\, c\, c'$ is derived in two ways:

$$\underbrace{a\, a'\, b\, b'}_{S_1}\ \underbrace{c\, c'}_{S_2} \qquad\qquad \underbrace{a\, a'}_{S_1}\ \underbrace{b\, b'\, c\, c'}_{S_2}$$

□

To remove such an ambiguity, one should block moving a string from the suffix of language one to the prefix of language two (and conversely).

If the designer is free to modify the language, a simple[21] remedy is to interpose a new terminal as a separator between the two languages. In our example, with character '♯' as separator, the (concatenation) language $L_1\, \sharp\, L_2$ is easily defined without ambiguity by a grammar with initial rule $S \to S_1\, \sharp\, S_2$.

2.5.11.4 Unique Decoding

A nice illustration of concatenation ambiguity comes from the study of codes in information theory. An information source is a process that produces a message, i.e., a sequence of symbols from a finite set $\Gamma = \{A,\ B,\ \ldots,\ Z\}$. Each such symbol is then encoded into a string over a terminal alphabet Σ, typically binary, and a coding function maps each symbol into a short terminal string, termed its code.

[21] A more complex unambiguous grammar equivalent to G exists, having the property that every string not containing any letter c, such as string $b\, b'$, is in language L_1 but not in L_2. On the other hand, notice that in so doing string $b\, c\, c'\, b'$ is assigned to language L_2.

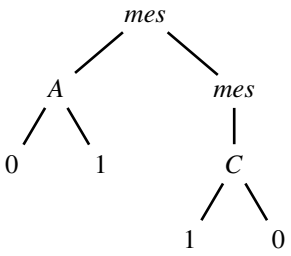
For instance, consider the following source symbols and their mapping into a binary code over alphabet $\Sigma = \{0, 1\}$:

$$\Gamma = \{ \overbrace{A}^{01}, \overbrace{C}^{10}, \overbrace{E}^{11}, \overbrace{R}^{001} \}$$

Message *ARRECA* is encoded as 0100100111001. By decoding this string, the original text is the only one that is obtained. Message coding is expressed by grammar G_1:

$$G_1 \begin{cases} mes \to A\,mes \mid C\,mes \mid E\,mes \mid R\,mes \mid A \mid C \mid E \mid R \\ A \to 01 \qquad C \to 10 \qquad E \to 11 \qquad R \to 001 \end{cases}$$

Grammar G_1 generates a message such as *AC* by concatenating the corresponding codes, as displayed in the syntax tree:



As grammar G_1 is clearly unambiguous, every encoded message, i.e., every sentence of language $L(G_1)$, has one and only one syntax tree corresponding to the decoded message.

On the contrary, the next bad choice of codes:

$$\Gamma = \{ \overbrace{A}^{00}, \overbrace{C}^{01}, \overbrace{E}^{10}, \overbrace{R}^{010} \}$$

renders ambiguous the grammar:

$$\begin{cases} mes \to A\,mes \mid C\,mes \mid E\,mes \mid R\,mes \mid A \mid C \mid E \mid R \\ A \to 00 \qquad C \to 01 \qquad E \to 10 \qquad R \to 010 \end{cases}$$

Consequently, message 000100101001 00 is decipherable in two ways: *ARRECA* and *ACAEECA*. The defect is twofold. (*i*) The identity:

$$\underbrace{01}_{\text{init}} \cdot 00 \cdot 10 = \underbrace{010}_{\text{init}} \cdot 010$$

holds for two sequences of concatenated codes. (*ii*) The initial code 0 1 of the left sequence is prefix of the initial code 0 1 0 of the right one.

Code theory studies conditions such as the negation of (*ii*) that make a code set uniquely decipherable.

2.5.11.5 Other Ambiguous Situations

The next case is similar to the ambiguity of regular expressions (p. 22).

Example 2.56 (*analogy with r.e.*) Consider the left grammar:

$$\begin{cases} 1\colon\ S \to D\,c\,D \\ 2\colon\ D \to b\,D \mid c\,D \mid \varepsilon \end{cases} \qquad \begin{cases} 1\colon\ S \to B\,c\,D \\ 2\colon\ B \to b\,B \mid \varepsilon \\ 3\colon\ D \to b\,D \mid c\,D \mid \varepsilon \end{cases}$$

Nonterminal D generates language $\{b,\ c\}^*$. Rule 1 says that a sentence contains at least one letter c. The same structure is defined by the ambiguous regular expression $\{b,\ c\}^*\ c\ \{b,\ c\}^*$: every sentence with two or more letters c is ambiguous as the distinguished occurrence of c is not fixed. This defect can be repaired by imposing that the distinguished c is, say, the leftmost one, as the right grammar above shows. Notice in fact that nonterminal B may not derive a string containing letter c. □

Example 2.57 (*setting an order on rules*) In the left grammar:

$$\begin{cases} 1\colon\ S \to b\,S\,c \\ 2\colon\ S \to b\,b\,S\,c \\ 3\colon\ S \to \varepsilon \end{cases} \qquad \begin{cases} 1\colon\ S \to b\,S\,c \mid D \\ 2\colon\ D \to b\,b\,D\,c \mid \varepsilon \end{cases}$$

rules 1 and 2 may be applied in any order, thus producing the ambiguous derivations:

$$\begin{aligned} S &\Rightarrow b\,b\,S\,c \Rightarrow b\,b\,b\,S\,c\,c \Rightarrow b\,b\,b\,c\,c \\ S &\Rightarrow b\,S\,c \Rightarrow b\,b\,b\,S\,c\,c \Rightarrow b\,b\,b\,c\,c \end{aligned}$$

Remedy: impose that rule 1 is applied before rule 2 in any derivation, as the right grammar above shows. □

2.5.11.6 Ambiguity of Conditional Statements

A notorious case of ambiguity in the programming languages with conditional instructions occurred in the first version of language Algol 60,[22] a milestone for

[22] The official version [11] removed the ambiguity.

the application of *CF* grammars. Consider the grammar:

$$S \rightarrow \textbf{if}\, b\, \textbf{then}\, S\, \textbf{else}\, S \mid \textbf{if}\, b\, \textbf{then}\, S \mid a$$

where letters b and a stand for a Boolean condition and any nonconditional construct, respectively; for brevity both are left undefined. The first and second alternatives produce a *two-leg* and a *one-leg* conditional, respectively.

Ambiguity arises when two sentences are nested, with the outermost being a two-way conditional. For instance, examine the two readings:

$$\underbrace{\textbf{if}\, b\, \textbf{then}\, \underbrace{\textbf{if}\, b\, \textbf{then}\, a\, \textbf{else}\, a}_{\text{2-way conditional}}}_{\text{1-way conditional}} \qquad \underbrace{\textbf{if}\, b\, \textbf{then}\, \underbrace{\textbf{if}\, b\, \textbf{then}\, a}_{\text{1-way conditional}}\, \textbf{else}\, a}_{\text{2-way conditional}}$$

caused by the 'dangling' **else**. It is possible to eliminate the ambiguity at the cost of complicating the grammar. Assume we decide to choose the left skeleton tree, which binds the **else** to the immediately preceding **if**. The corresponding grammar is:

$$\begin{cases} S \rightarrow S_E \mid S_T \\ S_E \rightarrow \textbf{if}\, b\, \textbf{then}\, S_E\, \textbf{else}\, S_E \mid a \\ S_T \rightarrow \textbf{if}\, b\, \textbf{then}\, S_E\, \textbf{else}\, S_T \mid \textbf{if}\, b\, \textbf{then}\, S \end{cases}$$

Observe that the syntax class S has been split into two classes. Class S_E defines a two-leg conditional such that its nested conditionals are in the same class S_E. The other class S_T defines a one-leg conditional, or a two-leg one such that the first and second nested conditionals are of class S_E and S_T, respectively. This excludes the two combinations:

$$\textbf{if}\, b\, \textbf{then}\, S_T\, \textbf{else}\, S_T \qquad\qquad \textbf{if}\, b\, \textbf{then}\, S_T\, \textbf{else}\, S_E$$

The facts that only class S_E may precede **else** and that only class S_T defines a one-leg conditional, disallow the derivation of the right skeleton tree.

If the language syntax can be modified, a simpler solution exists: many language designers have introduced a closing mark for delimiting conditional constructs. See the next use of the closing mark **end_if**:

$$S \rightarrow \textbf{if}\, b\, \textbf{then}\, S\, \textbf{else}\, S\, \textbf{end_if} \mid \textbf{if}\, b\, \textbf{then}\, S\, \textbf{end_if} \mid a$$

Indeed, this is a sort of parenthesizing (p. 50) of the original grammar.

2.5.11.7 Inherent Ambiguity of Language

In all the preceding examples, we have found that a language is defined by equivalent grammars, some ambiguous some not. Yet this is not always the case. A language is called *inherently ambiguous* if every grammar that defines it is ambiguous. Surprisingly enough, inherently ambiguous languages exist!

Example 2.58 (*unavoidable ambiguity from union*) Recall the language L of Example 2.48 on p. 51:

$$L = \left\{ a^i\, b^j\, c^k \mid \;\; i, j, k \geq 0 \;\wedge\; (i = j \;\vee\; j = k) \right\}$$

Language L can be equivalently defined by means of the union:

$$L = L_1 \cup L_2 = \left\{ a^i\, b^i\, c^* \mid \;\; i \geq 0 \right\} \cup \left\{ a^*\, b^i\, c^i \mid \;\; i \geq 0 \right\}$$

of two nondisjoint languages L_1 and L_2.

We intuitively argue that any grammar G of language L is necessarily ambiguous. The grammar G on p. 51 unites the rules of the component grammars G_1 and G_2 and is obviously ambiguous for every sentence $x \in \{ \varepsilon,\, a\,b\,c,\, \ldots,\, a^i\, b^i\, c^i,\, \ldots \}$ shared by languages L_1 and L_2. Any such sentence is produced by G_1 by means of rules checking that $|\,x\,|_a = |\,x\,|_b$. This check is possible only for a syntax structure of the type:

$$\underbrace{a \;\ldots\; a\; \underbrace{a\, b}\; b \;\ldots\; b\;\; c\, c \;\ldots\; c}$$

Now the same string x, viewed as a sentence of language L_2, must be generated by grammar G_2 with a structure of the type:

$$\underbrace{a \;\ldots\; a\, a\;\; b \;\ldots\; b\; \underbrace{b\, c}\; c \;\ldots\; c}$$

in order to perform the equality check $|\,x\,|_b = |\,x\,|_c$. No matter which grammar variation we make, the two exponent equality checks are unavoidable and the grammar remains ambiguous for such sentences. □

In reality, inherent language ambiguity is rare and it scarcely affects technical languages, if ever.

2.5.12 Weak and Structural Equivalence

It is not enough for a grammar to generate correct sentences, as it should also assign to each of them a suitable structure, in agreement with the intended meaning. This *structural adequacy* requirement has already been invoked at times, for instance when discussing operator precedence in hierarchical lists.

We ought to reexamine the notion of grammar in light of structural adequacy. Recall the definition on p. 38: two grammars G and G' are equivalent if they define the same language, i.e., $L(G) = L(G')$. Such a definition, to be qualified now as *weak equivalence*, poorly fits with the real possibility of substituting one grammar for the other in technical language processors such as compilers. The reason is that the two grammars are not guaranteed to assign the same meaningful structure to every sentence.

We need a more stringent definition of equivalence, which is only relevant for unambiguous grammars. Grammars G and G' are *strongly* or *structurally equivalent*, if (i) $L(G) = L(G')$ and (ii) in addition grammars G and G' assign to each sentence two syntax trees, which may be considered *structurally similar*.

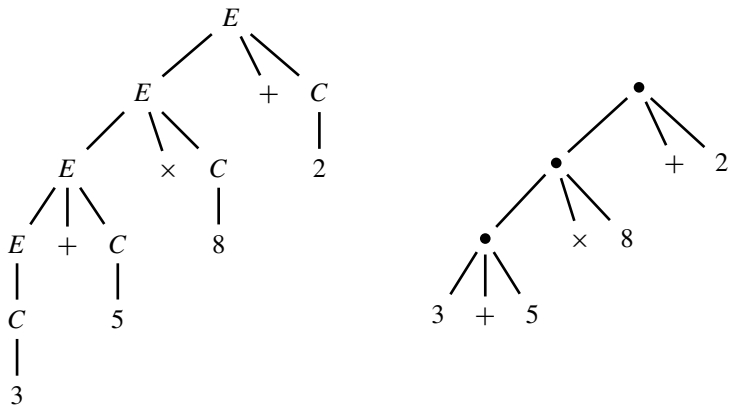
Condition (ii) should be formulated in accordance with the intended application. A plausible formulation is: two syntax trees are structurally similar if the corresponding condensed skeleton trees (p. 45) are equal.

Strong equivalence implies weak equivalence, but the former is a decidable property, unlike the latter.[23]

Example 2.59 (*structural adequacy of arithmetic expressions*) The difference between strong and weak equivalence is manifested by the case of arithmetic expressions, such as $3 + 5 \times 8 + 2$, first viewed as a list of digits separated by addition and multiplication signs. Here is a first grammar G_1:

$$G_1 \begin{cases} E \rightarrow E + C \mid E \times C \mid C \\ C \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{cases}$$

The syntax tree of the previous sentence is:



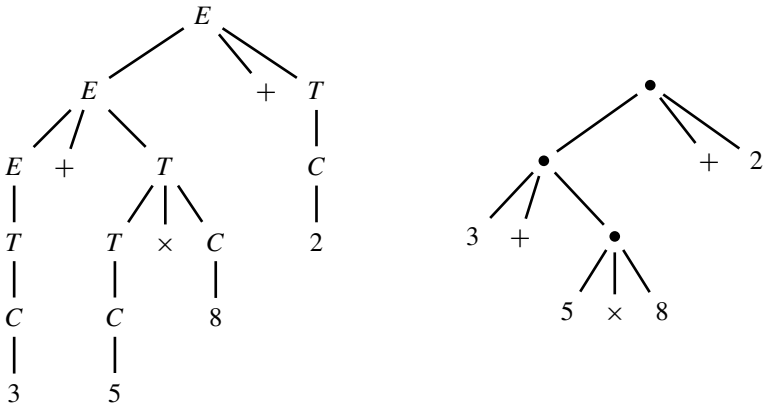
[23] The decision algorithm, e.g., in [12], is similar to the one for checking the equivalence of finite automata, to be presented in the next chapter.

In the condensed skeleton (right), nonterminals and copy rules are dropped.

Here is a second grammar G_2:

$$G_2 \begin{cases} E \rightarrow E + T \mid T \\ T \rightarrow T \times C \mid C \\ C \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{cases}$$

Grammars G_1 and G_2 are weakly equivalent. When we observe the syntax tree of the same sentence:



we see that the condensed skeleton differs from the previous one: it contains a subtree with frontier 5×8, associated with a multiplication. Therefore, grammars G_1 and G_2 are not structurally equivalent.

Is either one of the grammars preferable? Concerning ambiguity, both of them are all right. But only grammar G_2 is structurally adequate, if one considers also meaning. In fact, sentence $3 + 5 \times 8 + 2$ denotes a computation to be executed in the traditional order $3 + (5 \times 8) + 2 = (3 + 40) + 2 = 43 + 2 = 45$: this is the *semantic interpretation*. The parentheses that specify the evaluation order $\big((3 + (5 \times 8)) + 2\big)$ can be mapped on the subtrees of the skeleton tree produced by G_2.

Instead, grammar G_1 produces the parenthesizing $\big(((3 + 5) \times 8) + 2\big) = 66$, which is inadequate, because by giving precedence to the first addition over multiplication, it returns a wrong semantic interpretation, i.e., the value 66. Incidentally, grammar G_2 is more complex because enforcing operator precedence requires more nonterminals and rules.

It is crucial for a grammar intended for driving a compiler to be structurally adequate, as we will see in Chap. 5 on syntax-directed translation.

A case of structural equivalence is illustrated by the next grammar G_3[24]:

$$\begin{cases} E \rightarrow E+T \mid T+T \mid C+T \mid E+C \mid T+C \mid C+C \mid T \times C \mid C \times C \mid C \\ T \rightarrow T \times C \mid C \times C \mid C \\ C \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{cases} \quad \textit{grammar } G_3$$

Now the condensed skeleton trees of any arithmetic expression coincide for grammars G_2 and G_3. They are structurally equivalent. □

Generalized Structural Equivalence

Sometimes a similarity criterion looser than the strict identity of condensed skeleton trees is more suitable. This consists in requesting that the two corresponding trees should be easily mapped one onto the other by some simple transformation. The idea can be differently materialized: one possibility is to have a bijective correspondence between the subtrees of one tree and the subtrees of the other tree. For instance, the two grammars:

$$S \rightarrow S\,a \mid a \qquad\qquad S \rightarrow a\,S \mid a$$

are just weakly equivalent in generating language $L = a^+$, since the condensed skeleton trees differ, as in the example of sentence $a\,a$:



However, the two grammars may be considered structurally equivalent in a generalized sense, as each left-linear tree of the former grammar corresponds to a right-linear tree of the latter. The intuition that the two grammars are similar is satisfied because their trees are specularly identical, i.e., they coincide by turning left-recursive rules into right-recursive.

2.5.13 Grammar Transformations and Normal Forms

We are going to study a range of transformations that are useful to obtain grammars having certain desired properties, without affecting the language. Normal forms are restricted rule patterns, yet still allowing any context-free language to be defined. Such forms are widely used in theoretical papers, to simplify formal statements and

[24]Incidentally, grammar G_3 has more rules because it does not exploit copy rules to express the inclusion of syntax classes. Categorization and the ensuing taxonomies reduce description complexity in any area of knowledge.

theorem proofs. Otherwise, in applied work, normal form grammars are usually not a good choice, because they are often larger and less readable; moreover, most normal forms are not strongly equivalent to the original grammar. However, some normal forms have important practical uses for language analysis in Chap. 4. In particular, the nonleft-recursive form is needed by top-down deterministic parsers and the operator form is exploited by parallel parsing algorithms.

We start the survey of transformations from the simple ones. Let a grammar $G = (V, \Sigma, P, S)$ be given.

2.5.13.1 Nonterminal Expansion

A general-purpose transformation preserving language is *nonterminal expansion*, which consists of replacing a nonterminal with its alternatives.

Replace rule $A \to \alpha B \gamma$ with rules:

$$A \to \alpha \beta_1 \gamma \mid \alpha \beta_2 \gamma \mid \cdots \mid \alpha \beta_n \gamma \qquad n \geq 1$$

where $B \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$ are all the alternatives of B. Clearly, the language does not change, since the two-step derivation $A \Rightarrow \alpha B \gamma \Rightarrow \alpha \beta_i \gamma$ becomes the immediate derivation $A \Rightarrow \alpha \beta_i \gamma$, to the same effect.

2.5.13.2 Axiom Elimination from Right Parts

At no loss of generality, every right part of a rule may exclude the presence of the axiom, i.e., it may be a string over alphabet $\Sigma \cup (V \setminus \{S\})$. To this end, just introduce a new axiom S_0 and the rule $S_0 \to S$.

2.5.13.3 Nullable Nonterminals and Elimination of Empty Rules

A nonterminal A is *nullable* if it can derive the empty string, i.e., there exists derivation $A \stackrel{+}{\Rightarrow} \varepsilon$. Consider the set named $Null \subseteq V$ of nullable nonterminals. Set $Null$ is computed by the following logical clauses, to be applied in any order until the set ceases to grow, i.e., a fixed point is reached:

$$A \in Null \quad \text{if} \begin{cases} (A \to \varepsilon) \in P \\ (A \to A_1 A_2 \ldots A_n) \in P \text{ with } A_i \in V \setminus \{A\} \\ \text{and for every } 1 \leq i \leq n \text{ it holds } A_i \in Null \end{cases}$$

Row one finds the nonterminals that are immediately nullable, and row two finds those that derive a string of nullable nonterminals.

Table 2.4 Elimination of copy rules

Nullable	Original grammar	Nonnullable grammar (to be cleaned)
No	$S \to SAB \mid AC$	$S \to SAB \mid SA \mid SB \mid S \mid A\,C \mid C$
Yes	$A \to a\,A \mid \varepsilon$	$A \to a\,A \mid a$
Yes	$B \to b\,B \mid \varepsilon$	$B \to b\,B \mid b$
No	$C \to c\,C \mid c$	$C \to c\,C \mid c$

Example 2.60 (*computing nullable nonterminals*) Examine this grammar:

$$\begin{cases} S \to S\,A\,B \mid A\,C \\ A \to a\,A \mid \varepsilon \\ B \to b\,B \mid \varepsilon \\ C \to c\,C \mid c \end{cases}$$

Result: $Null = \{A,\ B\}$. Notice that if rule $S \to A\,B$ were added to the grammar, the result would include $S \in Null$. □

The *normal form without nullable nonterminals*, in short *nonnullable*, is defined by the condition that no nonterminal is nullable, but the axiom. Clearly, the axiom is nullable if, and only if, the empty string is in the language.

To construct the nonnullable form, first compute set *Null*, then do so:

- for each rule $(A \to A_1\,A_2\,\ldots\,A_n) \in P$, with $A_i \in V \cup \Sigma$, add as alternatives the strings obtained from the right part, by deleting in all possible ways any nullable nonterminal A_i
- remove the empty rules $A \to \varepsilon$, for every nonterminal $A \neq S$

If the grammar thus obtained is unclean or circular, it should be cleaned with the known algorithms (p. 39).

Example 2.61 (*Example 2.60 continued*) In Table 2.4, column one lists nullability. The other columns list the original rules and those produced by the transformation, to be cleaned by removing the circular alternative $S \to S$. □

2.5.13.4 Copies or Subcategorization Rules and Their Elimination

A *copy* (or *subcategorization*) rule has the form $A \to B$, where $B \in V$ is a nonterminal symbol. Any such rule is tantamount to the relation $L_B\,(G) \subseteq L_A\,(G)$, which means that the syntax class B is included in the class A.

For a concrete example, related to programming languages, these rules:

$$iterative_phrase \to while_phrase \mid for_phrase \mid repeat_phrase$$

introduce three subcategories of iterative phrase: while, for, and repeat.

Copy rules can be eliminated, yet many more alternatives have to be introduced and grammar legibility usually deteriorates. Notice that copy elimination reduces the height of the syntax trees by shortening the derivations.

Now we present an algorithm for eliminating copy rules, not for practical utility, but because in Chap. 3 the same transformation is applied to eliminate the spontaneous moves of a finite automaton. For simplicity, we assume that the given grammar G is in nonnullable normal form and that the axiom S does not occur in any rule right part.

For a grammar G and nonterminal A, we define the set $Copy\,(A) \subseteq V$, containing A and the nonterminals that are immediate or transitive copies of A:

$$Copy\,(A) = \left\{ B \in V \mid \ \exists \text{ a derivation } A \stackrel{*}{\Rightarrow} B \right\} \tag{2.8}$$

To compute the set *Copy*, apply the following logical clauses until a fixed point is reached:

$$\begin{array}{ll} A \in Copy\,(A) & \text{initialization} \\ C \in Copy\,(A) & \text{if } B \in Copy\,(A) \ \wedge \ (B \to C) \in P \end{array}$$

Then construct the rule set P' of a new grammar G', equivalent to G and copy-free, as follows:

$$P' := P \setminus \{ A \to B \mid \ A, B \in V \} \qquad \text{cancelation of copy rules}$$

$$P' := P' \cup \left\{ A \to \alpha \,\middle|\, \begin{array}{l} \alpha \in \left((\Sigma \cup V)^* \setminus V \right) \wedge \\ (B \to \alpha) \in P \ \wedge \ B \in Copy\,(A) \end{array} \right\}$$

The effect is that a nonimmediate (multiple-step) derivation $A \stackrel{+}{\Rightarrow} B \Rightarrow \alpha$ of G shrinks to the immediate (one-step) derivation $A \Rightarrow \alpha$ of G'. Notice that the transformation retains all the original noncopy rules.

We note that if, contrary to our simplifying hypothesis, grammar G contains nullable nonterminals, the definition of set *Copy* at formula (2.8) and its computation must consider also the derivations of the form:

$$A \stackrel{+}{\Rightarrow} B\,C \stackrel{+}{\Rightarrow} B$$

where nonterminal C is nullable.

Example 2.62 (*copy-free rules for arithmetic expressions*) By applying the copy elimination algorithm to grammar G_2 (Example 2.59):

$$G_2 \begin{cases} E \rightarrow E + T \mid T \\ T \rightarrow T \times C \mid C \\ C \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{cases}$$

we obtain these copy sets:

$$Copy\,(E) = \{\, E,\ T,\ C \,\} \qquad Copy\,(T) = \{\, T,\ C \,\} \qquad Copy\,(C) = \{\, C \,\}$$

The copy-free grammar G_3 equivalent to G_2 is:

$$G_3 \begin{cases} E \rightarrow E + T \mid T \times C \mid 0 \mid 1 \mid 2 \mid \cdots \mid 9 \\ T \rightarrow T \times C \mid 0 \mid 1 \mid 2 \mid \cdots \mid 9 \\ C \rightarrow 0 \mid 1 \mid 2 \mid \cdots \mid 9 \end{cases}$$

□

It is worth repeating that copy rules are very convenient for reusing certain rule blocks, which correspond to syntactic subcategories. A grammar with copy rules is more concise than one without them, and it highlights the generalization and specialization of language constructs. For these reasons, the reference grammars of technical languages often include copy rules.

2.5.13.5 Repeated Right Parts and Invertible Grammars

In Example 2.62 on p. 69, the last grammar G_3 contains two rules with a repeated right part: $E \rightarrow T \times C$ and $T \rightarrow T \times C$. On the other hand, a grammar such as G_2, which does not include any rules with repeated right parts (*RRP*), is qualified as *invertible*. Before showing that every grammar admits an equivalent invertible form, we discuss the role of *RRP* in grammars.

A convenient use of *RRP* is when two nonterminals C_1 and C_2 include in their languages a common part R and also two idiosyncratic parts, respectively denoted A_1 and A_2, schematically: $C_1 \rightarrow R \mid A_1$ and $C_2 \rightarrow R \mid A_2$. Since part R can be the root of arbitrarily complex constructs, the *RRP* form may save considerable duplication in the grammar rules.

A concrete example occurs in many programming languages: two constructs exist for the assignment statement, one of general use and one restricted to variable initialization, say to assign a constant value. Clearly, the latter offers part of the possibilities of the former, and some saving of grammar rules may result from having a nonterminal R for the common subset.

On the other hand, the presence of *RRP* rules moderately complicates syntax analysis, when the algorithm is of the simplest bottom-up type, in particular the operator precedence analyzer to be presented in Chap. 4.

We show how to convert a grammar $G = (V, \Sigma, P, S)$ into an equivalent invertible grammar $G' = \left(V', \Sigma, P', S'\right)$.[25]

For simplicity, we start from a grammar G that is nonnullable, i.e., free from empty rules. In the algorithm, for uniformity the nonterminals of G (including its axiom) are denoted by subscripted letters B, while the nonterminals of G' are the axiom S' and the other nonterminals denoted by subscripted letters A. The idea is that each nonterminal A_i of G', except its axiom S', uniquely corresponds to a nonempty subset of nonterminals of G, i.e., $[B_1, \ldots, B_r]$, with $r \geq 1$, to be also denoted as $set(A_i)$.

We have to focus on the similarity of two rules such that their right parts, although not necessarily identical, only differ in the nonterminal names, as for instance the rules $B_3 \rightarrow u\, B_2\, v\, B_4\, w$ and $B_1 \rightarrow u\, B_3\, v\, B_2\, w$, where u, v, and w are possibly empty terminal strings. We say that such rules have the same *right part pattern*, which we may indicate by the string $u - v - w$; clearly, a special case of such similarity is when two rules have *RRP*.

Construction of Grammar G'

Initially, sets V' and P' are empty. Then the algorithm creates new nonterminals and rules, until the last step does not produce any new rule. Then the algorithm creates the axiomatic rules and terminates:

1. for each maximal set[26] of terminal rules of G that have a *RRP* u:

$$\begin{cases} B_1 \rightarrow u \\ B_2 \rightarrow u \\ \ldots \\ B_m \rightarrow u \end{cases} \quad \text{where } u \in \Sigma^+ \text{ and each } B_l \in V$$

   (any of the nonterminals B_l may also be the axiom S) add to V' the nonterminal $[B_1, B_2, \ldots, B_m]$, if not already present, and add to set P' the rule:

$$[B_1, B_2, \ldots, B_m] \rightarrow u$$

2. for each right part pattern ρ of G, of the form:

$$\rho = x_0 - x_1 - \cdots - x_j - \cdots - x_r, \text{ where } r \geq 1 \text{ and each } x_j \in \Sigma^*$$

   find in set P all the rules that have the same right part pattern ρ:

$$P_\rho = \begin{cases} B_1 \rightarrow x_0\, B_{1,1}\, x_1 \ \ldots\ B_{1,j}\, x_j \ \ldots\ B_{1,r}\, x_r \\ \ldots \\ B_m \rightarrow x_0\, B_{m,1}\, x_1 \ \ldots\ B_{m,j}\, x_j \ \ldots\ B_{m,r}\, x_r \end{cases}$$

[25] We refer to the book by Harrison [13] for more details and a correctness proof.

[26] This means that any proper subset included in set P_ρ is not considered.

Let $P_\rho \subseteq P$ be the set of such rules. Examine every sequence σ of r nonterminals in V' (not necessarily distinct), where number r is the same as in the pattern ρ:

$$\sigma = \left\langle A_{\sigma_1},\ \ldots,\ A_{\sigma_j},\ \ldots,\ A_{\sigma_r} \right\rangle \qquad \text{with } A_{\sigma_j} \in V'$$

Notice that A_{σ_j} is a nonterminal already present in V'. Therefore, there are $|\ V'\ |^r$ distinct sequences σ.
Then, select from set P_ρ every maximal subset of rules, named $P_{\rho,\sigma}$, defined as follows:

$$P_{\rho,\sigma} = \begin{cases} B_{h_1} \ \rightarrow\ x_0\, B_{h_1,1}\, x_1\ \ldots\ B_{h_1,j}\, x_j\ \ldots\ B_{h_1,r}\, x_r \\ \ldots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{with } 1 \leq t \leq m \\ B_{h_t} \ \rightarrow\ x_0\, B_{h_t,1}\, x_1\ \ldots\ B_{h_t,j}\, x_j\ \ldots\ B_{h_t,r}\, x_r \end{cases}$$

where, for each 'column' j with $1 \leq j \leq r$, the inclusion holds:

$$\left\{\ B_{h_1,j},\ B_{h_2,j},\ \ldots,\ B_{h_t,j}\ \right\} \subseteq set\,(A_{\sigma_j}) \tag{2.9}$$

Notice that all the indexes h_1, ..., h_t are distinct and that the set inclusion holds: $\{\, h_1,\ \ldots,\ h_t\,\} \subseteq \{\, 1,\ \ldots,\ m\,\}$. From such set of rules, create the nonterminal $[\, B_{h_1},\ B_{h_2},\ \ldots,\ B_{h_t}\,]$, if not already present in V', and add to P' the rule:

$$[\, B_{h_1},\ B_{h_2},\ \ldots,\ B_{h_t}\,] \rightarrow x_0\, A_{\sigma_1}\, x_1\, A_{\sigma_2}\, x_2\ \ldots\ A_{\sigma_r}\, x_r$$

The algorithm repeats the previous step 2 until it reaches a fixed point. Then let the current nonterminal and rule sets be V' and P', respectively.

3. To finish, in V' find every nonterminal, say A_i, that contains the axiom S in $set\,(A_{\sigma_i})$ and create the axiomatic rules:

$$S' \rightarrow A_i \qquad \text{such that } S \in set\,(A_i)$$

Example 2.63 (*elimination of repeated right parts*) We apply the construction to the noninvertible grammar G below:

$$G \begin{cases} S \ \rightarrow\ c\,S\ \mid\ c\,B_1\,d\,B_2 \\ B_1 \ \rightarrow\ a\,a'\ \mid\ b\,b'\ \mid\ a\,B_1\,a'\,B_1\ \mid\ b\,B_1\,b'\,B_1 \\ B_2 \ \rightarrow\ a\,a'\ \mid\ a\,B_2\,a'\,B_2 \end{cases}$$

The right part patterns of G are:

$$c - \varepsilon \qquad c - d - \varepsilon \qquad a\,a' \qquad b\,b' \qquad a - a' - \varepsilon \qquad b - b' - \varepsilon$$

In Table 2.5, we tabulate the nonterminals and rules stepwise created. The invertible grammar G' includes the rules computed by the algorithm:

$$G' \begin{cases} S' \to [S] \\ [S] \to c\,[B_1]\,d\,[B_1,\,B_2] \mid c\,[B_1,\,B_2]\,d\,[B_1,\,B_2] \mid c\,[S] \\ [B_1] \to b\,b' \mid a\,[B_1]\,a'\,[B_1] \mid a\,[B_1]\,a'\,[B_1,\,B_2] \mid \\ \qquad b\,[B_1]\,b'\,[B_1] \mid b\,[B_1]\,b'\,[B_1,\,B_2] \mid \\ \qquad b\,[B_1,\,B_2]\,b'\,[B_1] \mid b\,[B_1,\,B_2]\,b'\,[B_1,\,B_2] \\ [B_1,\,B_2] \to a\,a' \mid a\,[B_1,\,B_2]\,a'\,[B_1,\,B_2] \end{cases}$$

Grammar G' is more combinatorial than the original grammar G. □

2.5.13.6 Chomsky and Binary Normal Form

In the Chomsky normal form, there are only two rule types:

homogeneous binary	$A \to B\,C$	where $B,\ C \in V$
terminal with singleton right part	$A \to a$	where $a \in \Sigma$

Moreover, if the empty string is in the language, there is the axiomatic rule $S \to \varepsilon$, but the axiom S is not allowed in any rule right part. With such constraints, any internal node of a syntax tree may have either two nonterminal siblings or one terminal sibling.

Given a grammar, under the simplifying hypothesis of being in nonnullable form, we explain how to obtain a Chomsky normal form. Each rule $A_0 \to A_1\,A_2\,\ldots\,A_n$ of length $n > 2$ is converted into a rule of length 2, by singling out the first symbol A_1 and the remaining suffix $A_2\,\ldots\,A_n$. Then a new ancillary nonterminal is created, named $\langle\,A_2\,\ldots\,A_n\,\rangle$, and the new rule:

$$\langle\,A_2\,\ldots\,A_n\,\rangle \to A_2\,\ldots\,A_n$$

Now the original rule is replaced by:

$$A_0 \to A_1\ \langle\,A_2\,\ldots\,A_n\,\rangle$$

After this transformation, the rule right parts of length 2 may still contain terminals, which must be transformed into nonterminals. If a rule has the form $A \to a\,B$, with $a \in \Sigma$, it must be replaced by the following rules, where $\langle\,a\,\rangle$ is a new ancillary nonterminal:

$$A \to \langle\,a\,\rangle\ B \qquad\qquad \langle\,a\,\rangle \to a$$

Proceed similarly in the case the right parts are $a\,b$ or $B\,b$, with $b \in \Sigma$, by adding the new ancillary nonterminal $\langle\,b\,\rangle$.

Continue applying the same series of transformations to the grammar thus obtained, until all the rules are in the form requested.

Table 2.5 Invertible grammar rules created for Example 2.63

step 1	
maximal set of terminal rules with right part aa': $\{ B_1 \to a\,a',\ B_2 \to a\,a' \}$	rule added to P': $[B_1, B_2] \to a\,a'$
maximal set of terminal rules with right part bb': $\{ B_1 \to b\,b' \}$	rule added to P': $[B_1] \to b\,b'$
step 2	
$V' = \{ [B_1],\ [B_1,\ B_2] \}$ right part pattern $\rho_1 = c - \varepsilon$ $P_{\rho_1} = \{ S \to c\,S \}$	the possible sequences σ are listed along the rules created: $\langle [B_1] \rangle$ none; $\langle [B_1, B_2] \rangle$ none no rule is created because the inclusion condition (2.9) is false for every σ
$V' = \{ [B_1],\ [B_1,\ B_2] \}$ right part pattern $\rho_2 = c - d - \varepsilon$ $P_{\rho_2} = \{ S \to c\,B_1\,d\,B_2 \}$	$\langle [B_1]\,[B_1] \rangle$ none $\langle [B_1]\,[B_1, B_2] \rangle$ $[S] \to c[B_1]d[B_1, B_2]$ $\langle [B_1, B_2]\,[B_1] \rangle$ none $\langle [B_1, B_2]\,[B_1, B_2] \rangle$ $[S] \to c[B_1, B_2]d[B_1, B_2]$
$V' = \{ [B_1],\ [B_1,\ B_2],\ [S] \}$, right part pattern $\rho_2 = c - d - \varepsilon$ $P_{\rho_2} = \{ S \to c\,B_1\,d\,B_2 \}$	now V' has been updated with $[S]$, but it is unproductive to reconsider the pattern $\rho_2 = c - d - \varepsilon$, since no rule with such pattern contains S in its right part
$V' = \{ [B_1],\ [B_1,\ B_2],\ [S] \}$ right part pattern $\rho_1 = c - \varepsilon$ $P_{\rho_1} = \{ S \to c\,S \}$	$\langle [S] \rangle$ $[S] \to c[S]$
$V' = \{ [B_1],\ [B_1,\ B_2],\ [S] \}$ right part pattern $\rho_3 = a - a' - \varepsilon$ $P_{\rho_3} = \left\{ \begin{array}{l} B_1 \to a\,B_1\,a'\,B_1 \\ B_2 \to a\,B_2\,a'\,B_2 \end{array} \right\}$	$\langle [B_1]\,[B_1] \rangle$ $[B_1] \to a[B_1]a'[B_1]$ $\langle [B_1]\,[B_1, B_2] \rangle$ $[B_1] \to a[B_1]a'[B_1, B_2]$ $\langle [B_1, B_2]\,[B_1, B_2] \rangle$ $[B_1, B_2] \to a[B_1, B_2]a'[B_1, B_2]$ $\langle [S]\,[X] \rangle$ $X \in V'$ none $\langle [X]\,[S] \rangle$ $X \in V'$ none
$V' = \{ [B_1],\ [B_1,\ B_2],\ [S] \}$ right part pattern $\rho_4 = b - b' - \varepsilon$ $P_{\rho_4} = \{ B_1 \to b\,B_1\,b'\,B_1 \}$	$\langle [B_1]\,[B_1] \rangle$ $[B_1] \to b[B_1]b'[B_1]$ $\langle [B_1]\,[B_1, B_2] \rangle$ $[B_1] \to b[B_1]b'[B_1, B_2]$ $\langle [B_1, B_2]\,[B_1] \rangle$ $[B_1] \to b[B_1, B_2]b'[B_1]$ $\langle [B_1, B_2]\,[B_1, B_2] \rangle$ $[B_1] \to b[B_1, B_2]b'[B_1, B_2]$ $\langle [S]\,[X] \rangle$ $X \in V'$ none $\langle [X]\,[S] \rangle$ $X \in V'$ none
step 3	
creates the axiomatic rule	$S' \to [S]$

Example 2.64 (*conversion into Chomsky normal form*) The grammar below:

$$\left\{ \begin{array}{lcl} S & \to & d\,A \mid c\,B \\ A & \to & d\,A\,A \mid c\,S \mid c \\ B & \to & c\,B\,B \mid d\,S \mid d \end{array} \right.$$

becomes the following equivalent grammar, in Chomsky normal form:

$$\begin{cases} S \to \langle d \rangle A \mid \langle c \rangle B \\ A \to \langle d \rangle \langle AA \rangle \mid \langle c \rangle S \mid c \\ B \to \langle c \rangle \langle BB \rangle \mid \langle d \rangle S \mid d \end{cases} \qquad \begin{cases} \langle d \rangle \to d \\ \langle c \rangle \to c \\ \langle AA \rangle \to AA \\ \langle BB \rangle \to BB \end{cases} \quad \square$$

This form is used in mathematical essays, but rarely in practical work. We observe that an almost identical transformation would permit to obtain a grammar where every rule right part contains *at most* two nonterminals. Such form will be used in the next section on operator grammars.

2.5.13.7 Operator Grammar

A grammar is in *operator form* if, for each rule right part, there is at least one terminal symbol between any two nonterminal symbols, in formula:

$$\text{for every rule } A \to \alpha \text{ it holds } \alpha \cap \big((\Sigma \cup V)^* \, V \, V \, (\Sigma \cup V)^* \big) = \emptyset$$

Since such grammar form is used in some efficient language parsing algorithms, we explain how to obtain it, starting from a generic grammar. In reality, most technical languages include a majority of rules that already comply with the operator form, and a few rules that need to be transformed. For instance, the usual grammar of arithmetic expressions is already in operator form:

$$\begin{cases} E \to E + T \mid T \\ T \to T \times F \mid F \\ F \to a \mid \text{`(' } E \text{`)'} \end{cases}$$

whereas in the next grammar of the *prefix* Boolean expressions, where every operator precedes its operand(s), the first rule is not in operator form:

$$\begin{cases} E \to BEE \mid \neg E \mid a \\ B \to \wedge \mid \vee \end{cases}$$

Given a grammar G, we explain how to obtain an equivalent grammar H in operator form.[27]

[27] Following [14], where a correctness proof of the transformation can be found.

1. For each existing nonterminal X and for each terminal a, we create a new nonterminal, denoted X_a, with the stipulation that nonterminal X_a generates a string u if, and only if, nonterminal X generates the string $u\,a$. Therefore, for the languages respectively generated by nonterminals X and X_a by using the rules of grammar G and H, the identity holds:

$$L_G(X) = L_H(X_a)\,a \cup L_H(X_b)\,b \cup \cdots \cup \{\varepsilon\}$$

   where the union spans all the terminals and the empty string is omitted if it is not in $L_G(X)$. Clearly, if in $L_G(X)$ no string ends by, say, b, the language $L_H(X_b)$ is empty; therefore, nonterminal X_b is useless.
2. For every rule $A \to \alpha$ of G, for every nonterminal X occurring in the right part α, and for all the terminals a, b, …, replace X by the set of strings:

$$X_a\,a \qquad X_b\,b \qquad \ldots \qquad \varepsilon \tag{2.10}$$

   where the empty string ε is only needed if X is nullable.
3. To finish the construction, for every rule $A \to \beta\,a$ computed at (2.10), we create the rule $A_a \to \beta$.

As in similar cases, the construction may produce also useless nonterminals.

Example 2.65 (*conversion into operator normal form—from* [14]) The grammar G with rules $S \to a\,S\,S \mid b$ is not in operator form. The nonterminals created at step 1 are S_a and S_b. The rules created by (2.10) are:

$$S \to a\,S_a\,a\,S_a\,a \mid a\,S_a\,a\,S_b\,b \mid a\,S_b\,b\,S_a\,a \mid a\,S_b\,b\,S_b\,b \mid b$$

The empty string is not in $L_G(S)$. Then from step 3, we obtain the rules:

$$S_a \to a\,S_a\,a\,S_a \mid a\,S_b\,b\,S_a \qquad\qquad S_b \to a\,S_a\,a\,S_b \mid a\,S_b\,b\,S_b \mid \varepsilon$$

At last, since nonterminal S_a generates nothing, we delete rules $S_a \to a\,S_a\,a\,S_a \mid a\,S_b\,b\,S_a$ and $S_b \to a\,S_a\,a\,S_b$, and we obtain grammar H:

$$H \begin{cases} S \to a\,S_b\,b\,S_b\,b \mid b \\ S_b \to a\,S_b\,b\,S_b \mid \varepsilon \end{cases}$$

□

The operator form, combined with a sort of precedence relation between terminal symbols, yields the class of operator precedence grammars, to be studied in Chap. 4 for their nice application to serial and parallel parsing algorithms.

2.5.13.8 Conversion of Recursion from Left to Right

Another normal form, termed *nonleft-recursive*, is characterized by the absence of left-recursive rules or derivations (l-recursions). It is indispensable for the top-down parsers to be studied in Chap. 4. We explain how to transform l-recursion into right recursion.

Transformation of Immediate l-Recursion

The more common and easier case is when the l-recursion to be eliminated is immediate. Consider all l-recursive alternatives of a nonterminal A:

$$A \to A\,\beta_1 \mid A\,\beta_2 \mid \cdots \mid A\,\beta_h \qquad h \geq 1$$

where no string β_i is empty, and let the remaining alternatives of A, which are needed to terminate the recursion, be:

$$A \to \gamma_1 \mid \gamma_2 \mid \cdots \mid \gamma_k \qquad k \geq 1$$

Create a new ancillary nonterminal A' and replace the previous rules with the next ones:

$$\begin{cases} A \to \gamma_1 A' \mid \gamma_2 A' \mid \cdots \mid \gamma_k A' \mid \gamma_1 \mid \gamma_2 \mid \cdots \mid \gamma_k \\ A' \to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_h A' \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_h \end{cases}$$

Now every original derivation involving l-recursive steps, as for instance

$$\underbrace{A \Rightarrow A\,\beta_2 \Rightarrow A\,\beta_3\,\beta_2}_{\text{l-recursive}} \Rightarrow \gamma_1\,\beta_3\,\beta_2$$

is replaced by the equivalent derivation:

$$A \Rightarrow \gamma_1\,\underbrace{A' \Rightarrow \gamma_1\,\beta_3\,A'}_{\text{right-recursive}} \Rightarrow \gamma_1\,\beta_3\,\beta_2$$

Example 2.66 (*conversion of immediate l-recursion into right recursion*) In the usual grammar of arithmetic expressions:

$$\begin{cases} E \to E + T \mid T \\ T \to T \times F \mid F \\ F \to \text{`(' } E \text{ `)'} \mid i \end{cases}$$

nonterminals E and T are immediately l-recursive. By applying the transformation, the right-recursive grammar below is obtained:

$$\begin{cases} E \to T\,E' \\ E' \to +T\,E' \mid +T \end{cases} \qquad \begin{cases} T \to F\,T' \mid F \\ T' \to \times F\,T' \mid \times F \end{cases} \qquad F \to \text{`(' } E \text{ `)'} \mid i$$

Actually, in this case, but not always, a simpler solution is possible, to specularly reverse the l-recursive rules, thus obtaining:

$$\begin{cases} E \rightarrow T + E \mid T \\ T \rightarrow F \times T \mid F \\ F \rightarrow \text{`(' } E \text{ `)'} \mid i \end{cases}$$

□

Transformation of Nonimmediate Left Recursion

The next algorithm is used to eliminate nonimmediate l-recursion. We present it under the simplifying assumptions that grammar G is homogeneous, nonnullable and with singleton terminal rules; in other words, the rules are like those of the Chomsky normal form, but more than two nonterminals are permitted in a rule right part. The reader should be able to generalize the algorithm to any context-free grammar.

The algorithm has two nested loops. The internal loop applies nonterminal expansion to turn nonimmediate l-recursion into immediate l-recursion, and so it shortens the derivation length. The external loop turns immediate l-recursion into right recursion, and to do so it creates ancillary nonterminals.

Let $V = \{A_1, A_2, \ldots, A_m\}$ be the nonterminal alphabet and let A_1 be the axiom. For orderly scanning, we view the nonterminals as an (arbitrarily) ordered set, from 1 to m.

Input: a grammar with left recursion (l-recursion)
Output: an equivalent grammar without l-recursion

for $i := 1$ to m **do**
  for $j := 1$ to $i - 1$ **do**
    replace each rule of type $A_i \rightarrow A_j\, \alpha$ $(i > j)$ by the rules:
      $A_i \rightarrow \gamma_1\, \alpha \mid \gamma_2\, \alpha \mid \ldots \mid \gamma_k\, \alpha$
    where $A_j \rightarrow \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_k$ are the alternatives of A_j
    `// note this may create immediate l-recursions`
  eliminate, through the previous algorithm, all the immediate
  l-recursions that may have appeared as an alternative of A_i,
  thus creating the ancillary nonterminal A'_i

The idea is to modify the rules in such a way that, in the resulting grammar, if the right part of a rule $A_i \rightarrow A_j \ldots$ starts with a nonterminal A_j, then it is $i < j$, i.e., the left part nonterminal A_i precedes A_j in the ordering. This ensures the algorithm terminates.[28]

[28] A proof of termination and correctness can be found in [9] or in [15].

Table 2.6 Turning recursion from left to right

i	j	Algorithm action	Transformation of grammar G_3
1	Skipped	Eliminate the immediate left recursion of nonterminal A_1 (there is none)	Unchanged
2	1	Replace the rule $A_2 \to A_1\, d$ with those obtained by expanding nonterminal A_1	$\begin{cases} A_1 \to A_2\, a \mid b \\ A_2 \to A_2\, c \mid A_2\, a\, d \mid b\, d \mid e \end{cases}$
2	Terminated	Eliminate two immediate left recursions of nonterminal A_2 and finally obtain grammar G_3'	$\begin{cases} A_1 \to A_2\, a \mid b \\ A_2 \to b\, d\, A_2' \mid e\, A_2' \mid b\, d \mid e \\ A_2' \to c\, A_2' \mid a\, d\, A_2' \mid c \mid a\, d \end{cases}$

Example 2.67 (*conversion of nonimmediate l-recursion into right recursion*) We apply the algorithm to the grammar G_3 below:

$$G_3 \begin{cases} A_1 \to A_2\, a \mid b \\ A_2 \to A_2\, c \mid A_1\, d \mid e \end{cases}$$

which has the l-recursion $A_1 \Rightarrow A_2\, a \Rightarrow A_1\, d\, a$. In Table 2.6, we list the steps that produce grammar G_3', which is free from l-recursion. □

It would be straightforward to modify the algorithm to turn right recursion into left recursion, which is a conversion sometimes applied to speed up the bottom-up parsing algorithms of Chap. 4.

2.5.13.9 Greibach and Real-Time Normal Form

In the *real-time* normal form, every rule starts with a terminal:

$$A \to a\, \alpha \qquad \text{where } a \in \Sigma \text{ and } \alpha \in (\, \Sigma \cup V\,)^*$$

A special case of real-time form is the *Greibach* normal form:

$$A \to a\, \alpha \qquad \text{where } a \in \Sigma \text{ and } \alpha \in V^*$$

Every rule starts with a terminal, followed by zero or more nonterminals. To be exact, both forms exclude the empty string from the language.

The designation 'real-time' will be later understood as a property of the pushdown automaton that recognizes the language. At each step, the automaton reads and consumes an input character, i.e., a terminal; therefore, the total number of steps equals the length of the string to be recognized.

For simplicity, we assume that the given grammar is nonnullable, and we explain how to obtain the above forms. For the real-time form: first eliminate all the left

recursions, and then, by elementary transformations, expand any nonterminal that occurs in the first position of a rule right part, until a terminal prefix is produced. Subsequently, continue for the Greibach form: if in any position other than the first, a terminal occurs, replace it by an ancillary nonterminal, and add the terminal rule that derives the terminal.

Example 2.68 (*conversion into Greibach normal form*) The grammar below:

$$\begin{cases} A_1 \rightarrow A_2\, a \\ A_2 \rightarrow A_1\, c \mid b A_1 \mid d \end{cases}$$

is converted into Greibach form in the following three steps:

1. Eliminate l-recursions by the step:

$$A_1 \rightarrow A_2\, a \qquad\qquad A_2 \rightarrow A_2\, a\, c \mid b A_1 \mid d$$

and then:

$$A_1 \rightarrow A_2\, a \qquad\qquad \begin{cases} A_2 \rightarrow b A_1 A_2' \mid d A_2' \mid d \mid b A_1 \\ A_2' \rightarrow a\, c A_2' \mid a\, c \end{cases}$$

2. Expand the nonterminals in first position until a terminal prefix surfaces:

$$\begin{cases} A_1 \rightarrow b A_1 A_2'\, a \mid d A_2'\, a \mid d\, a \mid b A_1\, a \\ A_2 \rightarrow b A_1 A_2' \mid d A_2' \mid d \mid b A_1 \\ A_2' \rightarrow a\, c A_2' \mid a\, c \end{cases}$$

3. Substitute the ancillary nonterminals for any terminal in a position other than the first:

$$\begin{cases} A_1 \rightarrow b A_1 A_2' \langle a \rangle \mid d A_2' \langle a \rangle \mid d \langle a \rangle \mid b A_1 \langle a \rangle \\ A_2 \rightarrow b A_1 A_2' \mid d A_2' \mid d \mid b A_1 \\ A_2' \rightarrow a \langle c \rangle A_2' \mid a \langle c \rangle \\ \langle a \rangle \rightarrow a \\ \langle c \rangle \rightarrow c \end{cases}$$

If the transformation were stopped before the last step, the grammar would be in real-time form, but not in Greibach form. □

2.5.13.10 Final Remarks on Normal Forms

Among the many ways that CF grammars may be classified is that of 'shape of the rules', e.g., Chomsky normal form, operator normal form, real-time and Greibach normal form, linear, and right-linear. We have shown that some of these kinds, e.g., the first four above, can be used to define any CF language.

Other grammar forms with the same property have been studied in the literature, such as the *double Greibach normal form*, which is a special case of real-time form and has rules of the following shapes:

$$\begin{array}{ll} A \to a\,\alpha\,b & \text{where } a,\ b \in \Sigma \text{ and } \alpha \in V^+ \\ A \to u & \text{where } u \in \Sigma^+ \end{array}$$

Clearly, the shape of the rules makes it impossible to have left-recursive and right-recursive derivations, i.e., all the recursive derivations are self-nesting (see Table 2.3 on p. 36). We have seen on p. 76 that left-recursive derivations can be replaced by right-recursive ones. Now we show how to eliminate also right-recursive derivations, by converting a grammar into the double Greibach form.

Example 2.69 (*conversion into double Greibach normal form*) The language $\{\, a^*\, a^n\, b^n \mid \ n \geq 1 \,\}$, defined by the grammar below (left) through using right-recursive and self-nesting rules:

$$\left\{ \begin{array}{lcl} S & \to & a\,S \mid X \\ X & \to & a\,X\,b \mid a\,b \end{array} \right. \qquad \left\{ \begin{array}{lcl} S & \to & a\,S\,b \mid a\,b \mid a\,Y\,b \\ Y & \to & a\,Y\,a \mid a\,a \mid a \end{array} \right. \tag{2.11}$$

is also defined by the grammar (right) without any use of left/right-recursive rules. □

How would we go, if we wanted to prove that any *CF* grammar can be converted into the double Greibach normal form? To answer once and for all this and similar questions for various rule shapes, a more general form of grammar rule, called *position restricted*, has been studied.[29] A grammar is *position restricted of type* $(\, m_1,\ \ldots,\ m_{n+1} \,)$, where $n \geq 2$ and each m_i is a nonnegative integer, if each rule is either a terminal rule $A \to u$, with u a nonempty terminal string, or has the following form:

$$\begin{array}{l} A \to v_1\, A_1\, v_2\ \ldots\ v_n\, A_n\, v_{n+1} \\ \text{where } |\, v_1 \,| = m_1, \ldots, |\, v_{n+1} \,| = m_{n+1} \text{ and } \forall i\ A_i \in V,\ v_i \in \Sigma^* \end{array} \tag{2.12}$$

For instance, any rule in Chomsky normal form $A \to A_1\, A_2$ is position restricted of type $(\, m_1 = 0,\ m_2 = 0,\ m_3 = 0 \,)$.

[29] We refer to Blattner and Ginsburg [16] for a complete presentation.

A theoretical result of some generality and interest says that, for every context-free grammar and for every tuple $(m_1, \ldots, m_{n+1})$ with $n \geq 2$, there exists an equivalent position restricted grammar of type $(m_1, \ldots, m_{n+1})$.

We observe that the result above says more than the already stated properties about the operator/Greibach/double-Greibach grammar normal form. In particular, it says that each one of such grammar forms can be further restricted to have only two nonterminal symbols in the right part, provided that we allow also terminal rules $A \to u$ where the length of u is not necessarily one. Thus, the following different restricted forms enjoy the property of being normal forms (letters b and c are terminal characters):

position restricted type	*rule*	*name*
$m_1 = 0,\ m_2 = 1,\ m_3 = 0$	$A \to A_1\, b\, A_2$	binary infix operator form
$m_1 = 1,\ m_2 = 0,\ m_3 = 0$	$A \to b\, A_1\, A_2$	binary Greibach form
$m_1 = 0,\ m_2 = 0,\ m_3 = 1$	$A \to A_1\, A_2\, b$	reversed binary Greibach form
$m_1 = 1,\ m_2 = 0,\ m_3 = 1$	$A \to b\, A_1\, A_2\, c$	binary double Greibach form

To illustrate, we show the (ugly) grammar G in binary Greibach form equivalent to the grammars (2.11) above.

$$G \begin{cases} S \to a\,b \mid a\,a\,b \mid a\,A\,Z \\ A \to a \mid a\,a \mid a\,A\,A \\ Z \to a\,Z\,B \mid a\,b \\ B \to b \end{cases} \qquad \begin{aligned} L_A(G) &= \left\{ a^k \mid \ k \geq 1 \right\} \\ L_Z(G) &= \{ a^n\, b^n \mid \ n \geq 1 \} \end{aligned}$$

Obviously, if in a certain normal form we permit to have also some rules with extra shapes, the resulting grammar type remains a fortiori a normal form. For instance, we may allow in the binary infix operator form also rules $A \to a\, A_1\, b\, A_2$ of type $(m_1 = 1,\ m_2 = 1,\ m_3 = 0)$ that feature a prefix operator a and an infix operator b. Having a richer set of shapes usually results in a grammar that is more readable, and especially that has a structure better suited to the language semantic.

We add a last remark on the linear rules (just one nonterminal symbol in their right part). Such rules are not considered in the definition of the position restricted form, since such a form requires at least 3-tuples of integers. The reason is that linear rules alone do not suffice to generate all the context-free languages, as we will discuss later.

To finish, although not all the grammar transformations studied will be used in this book, in particular not the Chomsky and Greibach ones, still practicing with them is recommended as an exercise for becoming fluent in grammar design and manipulation, a skill certainly needed in language and compiler engineering.

2.6 Grammars of Regular Languages

Since regular languages are a rather limited class of context-free languages, it is not surprising that their grammars admit severe restrictions, to be next considered. Furthering the study of regular languages, we will also see that longer sentences have unavoidable repetitions, a property that can be exploited to prove that certain context-free languages are not regular. Other contrastive properties of the families *REG* and *CF* will emerge in Chaps. 3 and 4 from the consideration of the amount of memory needed to check whether a string is in a language, which is finite for the former family and unbounded for the latter.

2.6.1 From Regular Expression to Grammar

For a language defined with a regular expression, it is straightforward to write a grammar, by analyzing the expression and mapping its subexpressions onto grammar rules. At the construction heart, the iterative operators (star and cross) are replaced by unilaterally recursive rules.

Algorithm 2.70 (*conversion of regular expression into grammar*) First, we identify and label with a different number each subexpression contained in the given regular expression r. From the very definition of r.e., the possible cases and corresponding grammar rules (with uppercase nonterminals) are listed in Table 2.7. Notice that we allow an empty string ε as a term.

To shorten the grammar, if in any row a term r_i is a terminal or ε, we do not introduce a corresponding nonterminal E_i, but we write it directly in the rule. Notice that rows 3 and 4 offer the choice of left or right-recursive rules.

To apply this conversion scheme, each subexpression label is assigned to a nonterminal as a distinguishing subscript. At the start, the axiom is assigned to the whole r.e. An example suffices to understand the procedure. □

Example 2.71 (*from r.e. to grammar*) The expression e below:

$$e = (\,a\,b\,c\,)^* \cup (\,f\,f\,)^+$$

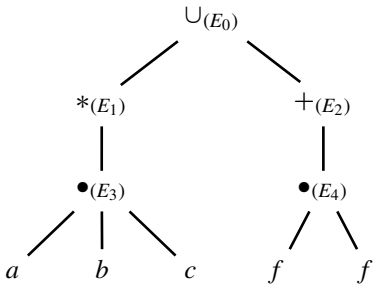
is analyzed into the arbitrarily numbered subexpressions shown in the tree below, which is a sort of abstract syntax tree (see p. 45) of the r.e. with added numbering:

Table 2.7 From regular subexpression to grammar rules

#	Regular (sub)expression	Grammar rule	Detail
1	$r = r_1 \cdot r_2 \cdot \cdots \cdot r_k$	$E \to E_1\, E_2\, \ldots\, E_k$	$k \geq 2$
2	$r = r_1 \cup r_2 \cup \cdots \cup r_k$	$E \to E_1 \mid E_2 \mid \cdots \mid E_k$	$k \geq 2$
3	$r = (\, r_1\,)^*$	$E \to E\, E_1 \mid \varepsilon$ or $E \to E_1\, E \mid \varepsilon$	
4	$r = (\, r_1\,)^+$	$E \to E\, E_1 \mid E_1$ or $E \to E_1\, E \mid E_1$	
5	$r = b$	$E \to b$	$b \in \Sigma$
6	$r = \varepsilon$	$E \to \varepsilon$	

Table 2.8 Mapping the regular expression of Example 2.71 onto grammar rules

Row in Table 2.7	Regular (sub)expression	Grammar rule
2	$E_1 \cup E_2$	$E_0 \to E_1 \mid E_2$
3	E_3^*	$E_1 \to E_1\, E_3 \mid \varepsilon$
4	E_4^+	$E_2 \to E_2\, E_4 \mid E_4$
1	$a\, b\, c$	$E_3 \to a\, b\, c$
1	$f\, f$	$E_4 \to f\, f$



We see that symbol E_0 (corresponding to the whole r.e. e) is the union of subexpressions E_1 and E_2, symbol E_1 is the star of subexpression E_3, etc.

The mapping of Table 2.7 yields the grammar rules in Table 2.8. Axiom E_0 derives the sentential forms E_1 and E_2, nonterminal E_1 generates the form E_3^*, and from that the string form $(\, a\, b\, c\,)^*$. Similarly, nonterminal E_2 generates the sentential form E_4^+ and finally the string form $(\, f\, f\,)^+$. □

We have several important remarks on the properties of grammars obtained from regular expressions through the mapping above.

First, any recursive rule in such a grammar is not self-nesting since it is either left- or right-recursive (lines 3 and 4 of Table 2.7). Moreover, the only recursive derivations in such grammars are immediate and correspond to the rules of lines 3 and

4. Therefore, such grammars never produce self-nesting derivations. We anticipate that the absence of such derivations in a grammar ensures that the language generated is regular.

Second, notice that if a regular expression is ambiguous (see the definition on p. 22), the corresponding grammar is so as well (see Example 2.75 on p. 85). Third, the grammar can be much more concise than the regular expression, because it saves the rules that define repeated subexpressions. To see that, consider the r.e. below:

$$e' = (a\,b\,c)^* \cup \left(f\ (a\,b\,c)^*\,f\right)^+$$

which differs from e of Example 2.71 by the repetition of subexpression $(a\,b\,c)^*$ in the second subexpression. Clearly, the grammar constructed for e' can reuse the rules $E_1 \to E_1\,E_3 \mid \varepsilon$ and $E_3 \to a\,b\,c$ inside the rules for the second term of e': $E_4 \to f\,E_1 f$.

We have seen how to map each r.e. operator onto equivalent grammar rules, to generate the same language. It follows that every regular language is context-free. Moreover, we intuitively know about context-free languages that are not regular, e.g., palindromes and Dyck language, and we shall see a formal proof of that in Example 2.81 on p. 90. Therefore, the following property holds.

Property 2.72 (language family inclusion) *The family* REG *of regular languages is strictly included in the family* CF *of context-free languages, i.e.,* REG $\subset$ CF. □

2.6.2 Linear and Unilinear Grammars

Algorithm 2.70 converts a regular expression into a grammar and substantially preserves the r.e. structure. But for a regular language, it is possible to constrain the grammar to a very simple rule form, called unilinear or type 3. Such a form gives evidence to some fundamental properties and leads to a straightforward construction of the automaton that recognizes the strings of a regular language.

We recall that a grammar is *linear* if every rule has the form:

$$A \to u\,B\,v \qquad \text{where } u,\ v \in \Sigma^* \text{ and } B \in (V \cup \{\varepsilon\}) \tag{2.13}$$

that is, at most one nonterminal occurs in the rule right part.

When we draw a syntax tree, we see that it never branches into two subtrees, but that it has a linear structure made by a stem with leaves directly attached to it. Linear grammars are not powerful enough to generate any context-free language (an example is the Dyck language), but they already exceed the power needed for regular languages. For instance, the following well-known subset of the Dyck language is generated by a linear grammar, though it is not regular (to be proved on p. 94).

Example 2.73 (*nonregular linear language*) The linear grammar $S \to b\,S\,e \mid b\,e$ generates language L_1:

$$L_1 = \left\{ b^n\, e^n \mid \ n \geq 1 \right\} = \{\, b\,e,\ b\,b\,e\,e,\ \ldots \}$$ □

Definition 2.74 (*left-linearity and right-linearity*) The rules of the following forms are called *right-linear* and, symmetrically, *left-linear*:

right-linear rule	$A \to u\,B$	where $u \in \Sigma^*$ and $B \in (\, V \cup \varepsilon\,)$
left-linear rule	$A \to B\,u$	with the same stipulations

Both cases are linear and obtained by deleting on either side one of the two terminal strings that embrace nonterminal B in a linear grammar; see (2.13). A grammar where all the rules are either right-linear or left-linear is termed *unilinear* or of *type* 3.[30] □

For a right-linear grammar, every syntax tree has an oblique stem sloping to the right, or to the left for a left-linear grammar. Moreover, if the grammar is recursive, it is necessarily right-recursive.

Example 2.75 (*unilinear grammar*) The strings containing a substring $a\,a$ and ending with a letter b are defined by the (ambiguous) regular expression:

$$(\,a \mid b\,)^*\ a\,a\ (\,a \mid b\,)^*\ b$$

The language is generated by the unilinear grammars G_r and G_l:

right-linear grammar

$$G_r \begin{cases} S \to a\,S \mid b\,S \mid a\,a\,A \\ A \to a\,A \mid b\,A \mid b \end{cases}$$

left-linear grammar

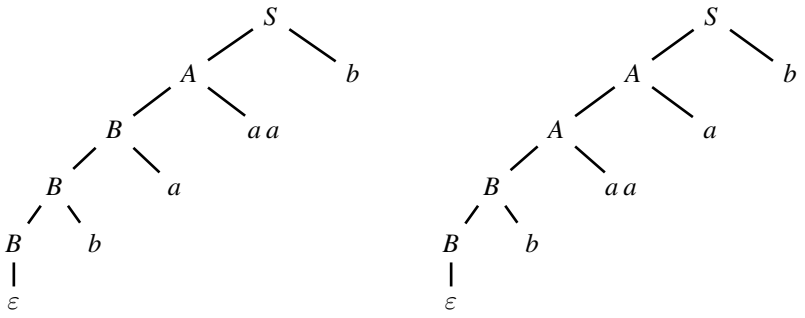
$$G_l \begin{cases} S \to A\,b \\ A \to A\,a \mid A\,b \mid B\,a\,a \\ B \to B\,a \mid B\,b \mid \varepsilon \end{cases}$$

An equivalent nonunilinear grammar is obtained by the algorithm on p. 82:

$$\begin{cases} E_1 \to E_2\,a\,a\,E_2\,b \\ E_2 \to E_2\,a \mid E_2\,b \mid \varepsilon \end{cases}$$

With grammar G_l, the leftward trees of the ambiguous sentence $b\,a\,a\,a\,b$ are:

[30] Within the Chomsky hierarchy (p. 105).



□

Example 2.76 (*parenthesis-free arithmetic expressions*) The language L:

$$L = \{ a,\ a + a,\ a \times a,\ a + a \times a,\ \ldots \}$$

is defined by the right-linear or left-linear grammars G_r or G_l:

$$G_r \begin{cases} S \rightarrow a \\ S \rightarrow a + S \\ S \rightarrow a \times S \end{cases} \qquad G_l \begin{cases} S \rightarrow a \\ S \rightarrow S + a \\ S \rightarrow S \times a \end{cases}$$

By the way, neither grammar is adequate to impose a precedence between arithmetic operations. □

Strictly Unilinear Grammars

The unilinear rule form can be further constrained, with the aim of simplifying the coming discussion on the theoretical properties and the construction of automata for language recognition. A grammar is *strictly unilinear* if every rule contains at most one terminal symbol, i.e., if it has the form:

$$A \rightarrow a\,B \quad (\text{or } A \rightarrow B\,a) \qquad \text{where } a \in (\Sigma \cup \varepsilon) \text{ and } B \in (V \cup \varepsilon)$$

A further simplification is possible: to impose that the only terminal rules are the empty ones. In this case, we may assume that the grammar contains just the following rule types:

$$A \rightarrow a\,B \mid \varepsilon \quad (\text{or } A \rightarrow B\,a \mid \varepsilon) \qquad \text{where } a \in \Sigma \text{ and } B \in V$$

Summarizing the discussion, we may indifferently use a grammar in unilinear form or strictly unilinear form, and we may additionally choose to have as terminal rules only the empty ones.

Example 2.77 (*Example 2.76 continued*) By adding ancillary nonterminals, the right-linear grammar G_r is transformed into an equivalent strictly right-linear grammar G'_r (left) and also into one with null terminal rules G''_r (right):

$$G'_r \begin{cases} S \rightarrow a \mid a\,A \\ A \rightarrow +S \mid \times S \end{cases} \qquad G''_r \begin{cases} S \rightarrow a\,A \\ A \rightarrow +S \mid \times S \mid \varepsilon \end{cases}$$

□

2.6.3 Linear Language Equations

We continue the study of unilinear grammars, and we show that the languages they generate are regular. The proof consists of transforming the unilinear rules into a set of linear equations, which have regular languages as their solution. In Chap. 3, we will see that every regular language can be defined by a unilinear grammar, and this proves the identity of the language families defined by regular expressions and by unilinear grammars.

For simplicity, consider a grammar $G = (V, \Sigma, P, S)$ in strictly right-linear form with null terminal rules (the case of left-linear grammars is analogous). Every rule can be transcribed into a linear equation that has as unknowns the languages generated from each nonterminal. For each nonterminal A, we shorten $L_A(G)$ as L_A:

$$L_A = \left\{ x \in \Sigma^* \mid \; A \stackrel{+}{\Rightarrow} x \right\}$$

In particular, it holds $L(G) \equiv L_S$. A string $x \in \Sigma^*$ is in language L_A if:

- string x is empty, and set P contains rule $A \rightarrow \varepsilon$
- string x is empty, set P contains rule $A \rightarrow B$, and it holds $\varepsilon \in L_B$
- string $x = a\,y$ starts with character a, set P contains rule $A \rightarrow a\,B$, and string $y \in \Sigma^*$ is in the language L_B

Let $n = |V|$ be the number of nonterminals of grammar G. Each nonterminal A_i is defined by a set of alternatives:

$$A_i \rightarrow a\,A_1 \mid b\,A_1 \mid \cdots \mid a\,A_n \mid b\,A_n \mid \cdots \mid A_1 \mid \cdots \mid A_n \mid \varepsilon$$

some possibly missing.[31] We write the corresponding linear equation:

$$L_{A_i} = a\,L_{A_1} \cup b\,L_{A_1} \cup \cdots \cup a\,L_{A_n} \cup b\,L_{A_n} \cup \cdots \cup L_{A_1} \cup \cdots \cup L_{A_n} \cup \varepsilon$$

The last term disappears if the rule does not contain the alternative $A_i \rightarrow \varepsilon$.

[31] In particular, alternative $A_i \rightarrow A_i$ is never present since the grammar is noncircular.

This system of n simultaneous equations in n unknowns (languages generated by nonterminals) can be solved through the well-known Gaussian elimination method, by applying the following formula to break recursion.

Property 2.78 (Arden identity) *The equation in the unknown language X:*

$$X = K\,X \cup L \tag{2.14}$$

where K is a nonempty language and L is any language, has one and only one solution:

$$X = K^* L \tag{2.15}$$

□

It is simple to see that language $K^* L$ is a solution of (2.14), since by substituting it for the unknown in both sides, the equation turns into the identity:

$$K^* L = \left(K\,K^* L\right) \cup L$$

We omit the proof that the only solution is (2.15).

Example 2.79 (*language equations*) The right-linear grammar below:

$$\begin{cases} S \;\rightarrow\; s\,S \mid e\,A \\ A \;\rightarrow\; s\,S \mid \varepsilon \end{cases}$$

defines a list of (possibly missing) elements e, divided by a separator s. It is transcribed into the system:

$$\begin{cases} L_S = s\,L_S \cup e\,L_A \\ L_A = s\,L_S \cup \varepsilon \end{cases}$$

First, substitute the second equation into the first:

$$\begin{cases} L_S = s\,L_S \cup e\,(s\,L_S \cup \varepsilon) \\ L_A = s\,L_S \cup \varepsilon \end{cases}$$

Then apply the distributive property of concatenation with respect to union, to factorize variable L_S as a common suffix:

$$\begin{cases} L_S = s\,L_S \cup e\,s\,L_S \cup e = (s \cup e\,s)\,L_S \cup e \\ L_A = s\,L_S \cup \varepsilon \end{cases}$$

Now apply Arden identity to the first equation, thus obtaining:

$$\begin{cases} L_S = (s \cup e\,s)^*\ e \\ L_A = s\,L_S \cup \varepsilon \end{cases}$$

and finally obtain $L_A = s\ (s \cup e\,s)^*\ e \cup \varepsilon$, by substitution. □

It would be straightforward to write linear equations also for a grammar that is unilinear, but not in the strict sense.

We have thus proved that every unilinearly generated language is regular. A complementary method for computing the regular expression of a language defined by a finite automaton will be described in Chap. 3.

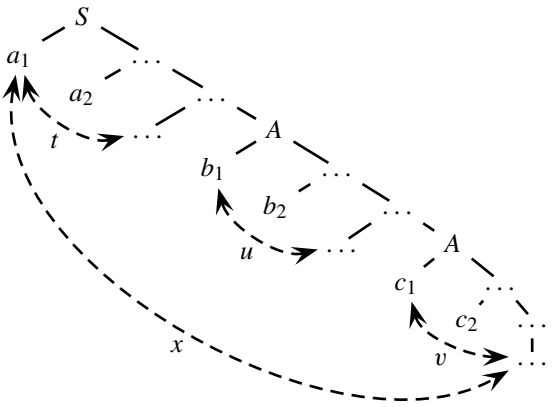
2.7 Comparison of Regular and Context-Free Languages

In this section, we present certain useful properties and techniques that permit to compare more closely regular and context-free languages, and we present a new very expressive model that combines regular expressions and grammar rules.

Next, we introduce a property that permits to understand the scope of regular languages, in order to realize which constructs can be thus defined and which instead require the full power of context-free grammars. Recall first that, in order to generate an infinite language, a grammar has to be recursive (Property 2.37 on p. 41), as only a recursive derivation, such as $A \overset{+}{\Rightarrow} u\,A\,v$, can be iterated for n (unbounded) times and can thus produce string $u^n\,A\,v^n$. This fact leads to observe that any sufficiently long sentence has to include a recursive derivation in its generation, therefore it contains certain substrings that can be repeated for unboundedly many times, thus producing a longer sentence of the language. The observation will be stated more precisely, first for the unilateral grammar, then for the context-free ones.

Property 2.80 (pumping of strings) *Take a unilinear grammar G. For any sufficiently long sentence x, meaning of length greater than some constant that only depends on the grammar, it is possible to find a factorization $x = t\,u\,v$, where substring u is not empty, such that, for every $n \geq 0$, the string $t\,u^n\,v$ is in the language (it is customary to say that the given sentence can be 'pumped' by injecting the substring u for arbitrarily many times).* □

Proof Take a strictly right-linear grammar and let k be the number of nonterminal symbols. Observe the syntax tree of any sentence x of length k or more. Clearly, two nodes exist with the same nonterminal label A:



Consider the factorization into $t = a_1\, a_2\, \ldots$, $u = b_1\, b_2\, \ldots$ and $v = c_1\, c_2\, \ldots\, c_m$. Therefore, there is a recursive derivation:

$$S \stackrel{+}{\Rightarrow} t\,A \stackrel{+}{\Rightarrow} t\,u\,A \stackrel{+}{\Rightarrow} t\,u\,v$$

that can be repeated to generate the strings $t\,u\,u\,v$, $t\,u\,\ldots\,u\,v$, and $t\,v$. □

Property 2.80 is next exploited to demonstrate that a quite simple context-free language is not regular.

Example 2.81 (*language with two equal powers*) Consider the familiar context-free language L_1:

$$L_1 = \left\{\, b^n\, e^n \mid \;\; n \geq 1 \,\right\}$$

and assume by contradiction that it is regular. Take a sentence $x = b^k\, e^k$, with k large enough, and break it into three substrings $x = t\,u\,v$, with substring u not empty. Depending on the positions of the two divisions, the substrings t, u, and v are as in the following scheme:

$$\begin{array}{l}
\underbrace{b\ldots b}_{t}\,\underbrace{b\ldots b}_{u}\,\underbrace{b\ldots b\,e\ldots \quad \ldots \quad \ldots e}_{v} \\
\underbrace{b\ldots b}_{t}\,\underbrace{b\ldots \quad \ldots b\,e\ldots \quad \ldots e}_{u}\,\underbrace{e\ldots e}_{v} \\
\underbrace{b\ldots \quad \ldots \quad \ldots b\,e\ldots e}_{t}\,\underbrace{e\ldots e}_{u}\,\underbrace{e\ldots e}_{v}
\end{array}$$

Pumping, i.e., repeating, the middle substring u will lead to a contradiction in all cases. For row one, if u is repeated twice, the number of letters b exceeds the number of letters e, thus causing the pumped string not to be in the language. For row two, when repeating u twice, the string $t\,u\,u\,v$ contains a pair of substrings $b\,e$ and does not conform to the language structure. Finally for row three, when repeating u, the

number of e exceeds the number of b. In all cases, the pumped strings are not valid and Property 2.80 is contradicted. This completes the proof that the language is not regular. □

Example 2.81 above should have convinced the reader that the regular family is too narrow for modeling some typical constructs of technical languages. Yet, it would be foolish to discard regular expressions, as they perfectly fit for modeling some most common parts of technical languages: on one hand, there are the substrings that make the so-called lexicon, e.g., numerical constants and identifiers, and on the other hand, there are many constructs that are variations over the list paradigm, e.g., lists of procedure parameters or instructions.

Role of Self-nesting Derivations

Since we have ascertained that the *REG* family is strictly included within the *CF* family, we focus on what makes some typical languages not regular, e.g., the two-power language, Dyck or palindromes. A careful observation reveals that their grammars have a common feature. All of them use some recursive derivation that is neither left nor right, but is *self-nesting*:

$$A \overset{+}{\Rightarrow} \alpha A \beta \qquad\qquad \alpha \neq \varepsilon \text{ and } \beta \neq \varepsilon$$

A grammar is not self-nesting if, for all nonterminals A, every derivation $A \overset{+}{\Rightarrow} \alpha A \beta$ has either $\alpha = \varepsilon$ or $\beta = \varepsilon$.

On the contrary, self-nesting derivations cannot be obtained with the grammars that we know to generate regular languages, namely those equivalent to a regular expression (construction of Table 2.7 on p. 83) and the left-linear or right-linear grammars; all of them produce only unilateral recursion.

It is the absence of self-nesting recursion that allowed us to solve linear equations by the Arden identity (p. 88). Disallowing self-nested derivations drastically reduces the generative capacity of context-free grammars, as next stated.

Property 2.82 (regularity of a language) *A language L is regular if, and only if, there exists a nonself-nesting grammar for L.* □

We have already argued that, if L is regular, then it is generated by a nonself-nesting (right-linear) grammar. The proof that, if L is generated by a nonself-nesting grammar, then L is regular, relies on an exhaustive analysis of all the possible derivations and is omitted.[32]

[32] The proof can be found in the book by Harrison [13]. Other interesting properties of nonself-nesting grammars are in [17].

Example 2.83 (*nonself-nesting grammar*) The grammar G below:

$$G \begin{cases} S \rightarrow A\,S \mid b\,A \\ A \rightarrow a\,A \mid \varepsilon \end{cases}$$

does not permit self-nesting derivations, though it is not unilinear. Thus, language $L(G)$ is regular, as we can see by solving the language equations:

$$\begin{cases} L_S = L_A\,L_S \cup b\,L_A \\ L_A = a\,L_A \cup \varepsilon \end{cases} \xrightarrow[\text{applied to } L_A]{\text{Arden identity}} \begin{cases} L_S = L_A\,L_S \cup b\,L_A \\ L_A = a^* \end{cases}$$

and obtain $L_S = (a^*)^*\,b\,a^* = a^*\,b\,a^*$, by substitution and simplification. □

Context-Free Languages Over One-Letter Alphabet

Given a grammar G, by Property 2.82 the absence of self-nesting derivations is a sufficient condition for the regularity of $L(G)$. But the presence of self-nesting derivations does not necessarily cause the language to be nonregular. On the way to illustrate this fact, we take the opportunity to mention a curious property of the context-free languages that have a one-letter (or unary) alphabet.

Property 2.84 (context-free language over a one-letter alphabet) *Every language defined by a context-free grammar over a one-letter alphabet* Σ, $|\,\Sigma\,| = 1$, *is regular.* □

Notice that the sentences x over a one-letter alphabet are in one-to-one correspondence with the integers, via mapping $x \leftrightarrow n$ if and only if $|\,x\,| = n$.

Example 2.85 (*grammar over a one-letter alphabet*) The grammar G below:

$$G\colon S \rightarrow a\,S\,a \mid \varepsilon$$

has the self-nesting derivation $S \Rightarrow a\,S\,a$, but $L(G) = \{\,a^n\,a^n \mid \ n \geq 0\,\} = (a\,a)^*$ is regular. A right-linear grammar equivalent to G is easily obtained:

$$S \rightarrow a\,a\,S \mid \varepsilon$$

by shifting to suffix the nonterminal S that was placed in the middle. □

From Property 2.84, it follows that any unary language that is not regular cannot be either context-free. An example is the set of all prime numbers represented as unary strings: $\{\,a,\ a^3,\ a^5,\ a^7,\ a^{11},\ \ldots\,\}$.

2.7.1 Limits of Context-Free Languages

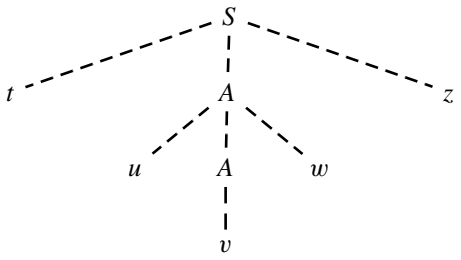
In order to understand what cannot be done with context-free grammars, we study the unavoidable repetitions that are found in the longer sentences of such languages, much as we did for regular languages. We will see that longer sentences necessarily contain two substrings that can be repeated for the same unbounded number of times by applying a self-nesting derivation. This property will be exploited to prove that context-free grammars are unable to generate certain languages where three or more parts are repeated for the same number of times.

Property 2.86 (language with three equal powers) *The language L below:*

$$L = \left\{ a^n b^n c^n \mid \ n \geq 1 \right\}$$

is not context-free. □

Proof By contradiction, assume that a grammar G of L exists and imagine the syntax tree of sentence $x = a^n b^n c^n$ ($n \geq 0$). Focus on the paths from root (axiom S) to leaf: at least one of such paths must have a length increasing with the length of x. Since n is unbounded, such a path necessarily traverses two nodes labeled with the same nonterminal, i.e., A. The situation is schematized in the draft tree (dashed edges may contain other nodes):



where strings t, u, v, w, and z are terminal. The corresponding derivation:

$$S \stackrel{+}{\Rightarrow} t\,A\,z \stackrel{+}{\Rightarrow} t\,u\,A\,w\,z \stackrel{+}{\Rightarrow} t\,u\,v\,w\,z$$

contains a recursive subderivation from A to A, which can be repeated any number $j \geq 0$ of times, thus producing strings of type:

$$y = t \underbrace{u \ldots\ldots u}_{\text{for } j \text{ times}} v \underbrace{w \ldots\ldots w}_{\text{for } j \text{ times}} z$$

Now examine all the possibilities for the strings u and w:

- both contain only one and the same character, say a. Thus, as j increases, string y will have more a than b, hence is not in the language.
- string u contains two or more different characters, for instance $u = \ldots\ a\ \ldots\ b\ \ldots$. Then, by repeating the recursive part of the derivation, we obtain $u\,u = \ldots\ a\ \ldots\ b\ \ldots\ a\ \ldots\ b\ \ldots$, wherein characters a and b are mixed up, hence string y is not in the language. We omit the analogous case when string w contains two different characters.
- string u contains only one character, say a, and string w contains only one different character, say b. When j increases, string y contains a number of a greater than that of c; hence, it is not valid. □

This reasoning exploits the possibility of pumping a sentence by repeating a recursive derivation. It is a useful conceptual tool for proving that certain languages are not in the family *CF*.

Although the language with three equal powers does not have any practical relevance, it illustrates a kind of agreement or concordance that cannot be enforced by context-free rules. The next case considers a construct more relevant for technical languages.

Language of Copies or Replica

An outstanding abstract paradigm is *replica*, to be found in many technical contexts, whenever two lists contain elements that must be pairwise identical or more generally agree with each other. A concrete case is provided by the declaration and the invocation of the same procedure: the correspondence between a 'formal parameter' list, like, e.g., $P(f_1 : int, f_2 : real, \ldots)$, and a matching 'actual parameter' list, like $P(i, r, \ldots)$. A similar example from English is:

cats, vipers, crickets and lions are respectively
mammals, snakes, insects and mammals

In the most abstract form, the two lists are made by the same alphabet and replica is the following language $L_{replica}$:

$$L_{replica} = \left\{ u\,u \mid \quad u \in \Sigma^+ \right\}$$

Let $\Sigma = \{a,\ b\}$. In some respect, a sentence $x = a\,b\,b\,b\,a\,b\,b\,b = u\,u$ is analogous to a palindrome $y = a\,b\,b\,b\,b\,b\,b\,a = u\,u^R$, but string u is copied in x and specularly reversed in y. We may say that the symmetry of the sentences of $L_{replica}$ is translational, not specular. Strangely enough, whereas palindromes are a most simple context-free language, the replica language is not context-free. The reason is that the two symmetries require quite different control mechanisms: a *LIFO* (last in first out) pushdown stack for specular symmetry and a *FIFO* (first in first out) queue for translational symmetry. We will see in Chap. 4 that the algorithms (or automata) that recognize context-free languages use a *LIFO* memory.

In order to show that replica is not in *CF*, one should apply again the pumping reasoning, but before doing so, we have to filter the language and render it similar to the three equal power languages of Property 2.86.

We focus on the following subset of language $L_{replica}$, obtained by means of an intersection with a regular language:

$$L_{abab} = \left\{ a^m b^n a^m b^n \mid \ m, n \geq 1 \right\} = L_{replica} \cap a^+ b^+ a^+ b^+$$

We state (anticipating the proof on p. 213) that the intersection of a context-free language with a regular one is always a context-free language. Therefore, if we prove that L_{abab} is not in *CF*, we may conclude the same for $L_{replica}$.

For brevity, the analysis of the possible cases for pumping strings is omitted, since it closely resembles the discussion in the previous proof (p. 93).

2.7.2 Closure Properties of Families *REG* and *CF*

We know that language operations are used to combine existing languages into new ones with the aim of extending, filtering, or modifying a given language. Yet not all the operations preserve the class or family the given languages belong to. When the result of an operation exits from the starting family, it cannot be generated with the same type of grammar.

We continue the comparison between regular and context-free languages, and in Table 2.9 we resume their closure properties with respect to language operations: some closures are already known, others are immediate, and a few need to await the results of automata theory to be proved. We denote a generic context-free language and a regular language as L and R, respectively. A few comments and examples follow.

- A nonmembership, such as $\neg L \notin CF$, means that the left term does not always belong to the family. Anyway, this does not exclude, for instance, that the complement of some context-free language is still context-free.
- The reversal of language $L(G)$ is generated by the *mirror grammar*, which is obtained by reversing the rule right parts. Clearly, if grammar G is right-linear, the mirror one is left-linear and defines a regular language.

Table 2.9 Closure properties of language families *REG* and *CF*

Reversal	Star	Union or concatenation	Complement	Intersection
$R^R \in REG$	$R^* \in REG$	$R_1 \oplus R_2 \in REG$	$\neg R \in REG$	$R_1 \cap R_2 \in REG$
$L^R \in CF$	$L^* \in CF$	$L_1 \oplus L_2 \in CF$	$\neg L \notin CF$	$L_1 \cap L_2 \notin CF$ $L \cap R \in CF$

- We know that the star, union, and concatenation of context-free languages are context-free. Let G_1 and G_2 be the grammars of languages L_1 and L_2, let S_1 and S_2 be their axioms, and suppose that their nonterminal sets are disjoint, i.e., $V_1 \cap V_2 = \emptyset$. To obtain the new grammar in the three cases, add to the rules of G_1 and G_2 the following axiomatic rules:

star	*union*	*concatenation*
$S \to S\,S_1 \mid \varepsilon$	$S \to S_1 \mid S_2$	$S \to S_1\,S_2$

  In the case of union, if the grammars are right-linear, so is the new grammar. On the contrary, although the new rules introduced for concatenation and star are not right-linear, we have observed before that the grammars are not self-nesting and define regular languages (p. 91); the family *REG* is in fact closed under such operations (Property 2.23 on p. 25).
- The complement of a regular language is regular; see the construction using finite automata in Chap. 3 (p. 189).
- In general, the intersection of context-free languages is not context-free, as witnessed by the language with three equal powers (Property 2.86 on p. 93):

$$\left\{ a^n\, b^n\, c^n \mid \ \ n \geq 1 \right\} = \left\{ a^n\, b^n\, c^+ \mid \ \ n \geq 1 \right\} \cap \left\{ a^+\, b^n\, c^n \mid \ \ n \geq 1 \right\}$$

  where the two components are easily defined by context-free grammars.
- As a consequence of the De Morgan identity, in general the complement of a context-free language is not context-free. As $L_1 \cap L_2 = \neg\,(\neg L_1 \cup \neg L_2)$, if the complement were context-free, a contradiction would ensue since the union of two context-free languages is context-free.
- On the other hand, the intersection of a context-free language and a regular one is context-free. The proof will be given on p. 213.

In order to make a grammar more discriminatory, the last property can be exploited to filter a language with a regular one, which forces some constraints on the original sentences.

Example 2.87 (*regular filter on the Dyck language (p. 49)*) It is instructive to see how the freely parenthesized sentences of a Dyck language L_D of alphabet $\Sigma = \left\{ a,\ a' \right\}$ can be filtered, by intersecting with the regular languages:

$$\begin{aligned} L_1 &= L_D \cap \neg \left(\Sigma^*\, a'\, a'\, \Sigma^* \right) = \left(a\, a' \right)^* \\ L_2 &= L_D \cap \neg \left(\Sigma^*\, a'\, a\, \Sigma^* \right) = \left\{ a^n \left(a' \right)^n \mid \ \ n \geq 0 \right\} \end{aligned}$$

The first intersection preserves the sentences that do not contain any substring $a'\,a'$, i.e., it eliminates all the sentences with nested parentheses. The second filter preserves the sentences that have exactly one parenthesis nest. Languages L_1 and L_2 are context-free, and the former is even regular. □

2.7.3 Alphabetic Transformations

It is a common feeling to find conceptually similar the languages that only differ by their concrete syntax, i.e., the choice of terminals. For instance, in different languages multiplication is denoted by a sign $\times$, a sign $*$, or a dot.

The term *alphabetic transliteration* or *homomorphism* refers to the linguistic operation that replaces individual characters by others.

Definition 2.88 (*alphabetic transliteration or homomorphism*[33]) Consider two alphabets: *source* Σ and *target* Δ. An alphabetic transliteration is a function:

$$h\colon \Sigma \to \Delta \cup \{\varepsilon\}$$

The transliteration or image of a character $c \in \Sigma$ is $h(c)$, i.e., an element of the target alphabet Δ. If $h(c) = \varepsilon$, the character c is erased. We call *nonerasing* a transliteration such that, for all source character c, the image $h(c)$ is in Δ.

The image of a source string $a_1\, a_2\, \ldots\, a_n$, with $a_i \in \Sigma$, is the string $h(a_1)\, h(a_2)\, \ldots\, h(a_n)$ obtained by concatenating the images of the individual characters. The image of the empty string ε is itself, i.e., $h(\varepsilon) = \varepsilon$. □

Transliteration is compositional: the image of the concatenation of two strings v and w is the concatenation of the images of the two strings:

$$h(v \cdot w) = h(v) \cdot h(w)$$

Example 2.89 (*text printer*) An obsolete text printer cannot print Greek letters and instead it prints the special character '□'. Moreover, the text sent to the printer may contain control characters, such as start-text and end-text, which are not printable. Such text transformation (disregarding the uppercase letters) is described by the transliteration:

$$\begin{aligned}
&h(c) = c && \text{if } c \in \{\, a,\ b,\ \ldots,\ z,\ 0,\ 1,\ \ldots,\ 9\,\}\\
&h(c) = c && \text{if } c \text{ is a punctuation mark or a blank space}\\
&h(c) = \square && \text{if } c \in \{\, \alpha,\ \beta,\ \ldots,\ \omega\,\}\\
&h(\textsf{start-text}) = h(\textsf{end-text}) = \varepsilon
\end{aligned}$$

An example of transliteration is:

$$\begin{aligned}
&h(\underbrace{\textsf{start-text const. } \pi \textsf{ has value 3.14 end-text}}_{\text{source string}}) && \text{– text sent to printer}\\
&= \underbrace{\textsf{const. } \square \textsf{ has value 3.14}}_{\text{target string}} && \text{– printout on paper}
\end{aligned}$$

□

[33] We anticipate that it is a simple case of the translation functions studied in Chap. 5.

An interesting special case of erasing homomorphism is the *projection*: a function that erases some source characters and leaves the others unchanged.

Transliteration into Words

The preceding qualification of transliteration (or homomorphism) as alphabetic means that the image of a character is still a character (or the empty string), not a longer string. Otherwise, an example of nonalphabetic transliteration is the conversion of an assignment statement $a \rightarrow b + c$ into the form $a := b + c$ through the function:

$$h(\text{`} \rightarrow \text{'}) = \text{`} := \text{'} \qquad \text{and } h(c) = c \text{ for any other } c \in \Sigma$$

This case is also called *transliteration into words*.

One more example. The vowels with umlaut of the German alphabet have strings of two characters as their images in the English alphabet:

$$h(\text{ä}) = \text{ae} \qquad h(\text{ö}) = \text{oe} \qquad h(\text{ü}) = \text{ue}$$

2.7.3.1 Language Substitution

A further generalization leads us to a language transformation termed *substitution*, already informally introduced when discussing linguistic abstraction on p. 28. Now a source character can be replaced by any string of a specified language over another alphabet.

Such language transformation is commonly used in language technology and compilation to separate a language definition into two levels, syntactic and lexical. The syntactic level is defined by a context-free grammar having as terminal alphabet a set of elements called lexical classes. The lexical level defines each lexical class as a formal language over the keyboard characters. To illustrate, the syntax of a programming language defines declarations, statements, expressions, etc., by means of grammar rules that contain as terminals the lexical class symbols, typically the following: *keyword* (**begin**, **if**, **then**, …), *operators* (such as plus or times), *identifier*, *integer_constant*, *real_constant*, *comment*. Then, an *identifier* is defined in the lexicon as, say, the formal language generated by the regular expression $(a \ldots z)\,(a \ldots z \mid 0 \ldots 9)^*$, etc. (The syntactic-lexical decomposition will be further discussed on p. 447.)

Substitution is also useful in the early phases of language design, in order to leave a syntactic construct unspecified in the working grammar, and to expand it later as project proceeds. The construct is denoted by a symbol, e.g., ⟨ARRAY⟩, temporarily considered as terminal. As the project progresses by stepwise refinements, the symbol will be substituted with the definition of the corresponding construct, by means of grammar rules or regular expression. See Example 2.92 at the end of this section.

Formally, given a source alphabet $\Sigma = \{a, b, \ldots\}$, a substitution h associates each character a, b, …with a language $h(a) = L_a$, $h(b) = L_b$, …over the target

alphabet Δ. By applying substitution h to a source string $a_1\, a_2\, \ldots\, a_n$, where every a_i is a character, we obtain a set of strings:

$$h\,(a_1\, a_2\, \ldots\, a_n) = \left\{\, y_1\, y_2\, \ldots\, y_n \mid \quad \forall\, 1 \leq i \leq n \quad y_i \in L_{a_i} \,\right\}$$

We may say that a transliteration into words is a substitution such that each image language contains only one string, and if the string has length one or zero, the transliteration is alphabetic.

2.7.3.2 Closure Under Alphabetic Transformation

Let L be a source language, context-free or regular, and let h be a substitution such that the image of each source character is a language in the same family as the source language. Then the substitution maps the set of source sentences, i.e., L, onto a set of image strings, called the *image* or *target* language, i.e., $L' = h\,(L)$. Is the target language still a member of the same family as the source language? The answer is affirmative and will be given by means of a construction that is valuable for modifying without effort the regular expression or the grammar of the source language.

Property 2.90 (closure under substitution – I) *The family* CF *is closed under the operation of substitution with languages of the same family (therefore also under transliteration).* □

Proof Let G be the grammar of language L over alphabet Σ, and let h be a substitution such that, for every $a \in \Sigma$, the language L_a is context-free. Then language L_a is defined by a grammar G_a with axiom S_a. Moreover, we assume that the nonterminal sets of grammars $G, G_a, G_b, \ldots$ are pairwise disjoint (otherwise it suffices to rename the common nonterminals).

Next, we construct the grammar G' of language $h\,(L)$ by transliterating the rules of G by means of the following mapping f:

$$\begin{array}{ll} f\,(a) = S_a & \text{for every terminal } a \in \Sigma \\ f\,(A) = A & \text{for every nonterminal } A \text{ of } G \end{array}$$

The rules of grammar G' are constructed as follows:

- apply transliteration f to every rule $A \to \alpha$ of G, and so replace each terminal character with the axiom of the corresponding target grammar
- add to G the rules of grammars $G_a, G_b, \ldots$

It is clear that the new grammar G' generates language $h\,(L\,(G))$. □

If the substitution h is a simple transliteration, constructing grammar G' is more direct: in G replace every terminal character $c \in \Sigma$ with its image $h\,(c)$.

For regular languages, it holds a result analogous to Property 2.90.

Property 2.91 (closure under substitution – II) *The family* REG *is closed under substitution with regular languages (therefore also under transliteration).* □

Essentially, the same construction in the proof of Property 2.90 can be applied to the source language r.e., and it computes the target language r.e.

Example 2.92 (*transliterated grammar*) The source language $i\ (\ ';'\ i\ )^*$, defined by rules:

$$S \to i\ ';'\ S\ \mid\ i$$

schematizes a program that includes a list of instructions i separated by semicolons. Now, imagine that the instructions have to be defined as assignments. Then, the following transliteration g into words is appropriate:

$$g\,(i) = v\ '\leftarrow'\ e$$

where v is a variable and e an expression. This produces the grammar:

$$\begin{cases} S \ \to\ A\ ';'\ S\ \mid\ A \\ A \ \to\ v\ '\leftarrow'\ e \end{cases}$$

As a next refinement, the definition of arithmetic expression can be plugged in by means of a substitution $h\,(e) = L_E$, where the image language L_E is well-known. Suppose it is defined by a grammar with axiom E. The grammar of the language after expression expansion is:

$$\begin{cases} S \ \to\ A\ ';'\ S\ \mid\ A & \text{– list of instructions separated by semicolon} \\ A \ \to\ v\ '\leftarrow'\ E & \text{– instruction expanded as an assignment} \\ E \ \to\ \ldots & \text{– the usual rules for arithmetic expressions} \end{cases}$$

As a last refinement, the symbol v, which stands for a variable, should be replaced with the regular language of identifier names. □

2.7.4 Grammars with Regular Expressions

The legibility of a regular expression is especially good for lists and similar structures, and it would be a pity to do without them when defining technical languages by means of grammars. Since we know that recursive rules are indispensable for parenthesis structures, it is attractive to combine r.e. and grammar rules into one notation, called *extended context-free grammar* or *EBNF*,[34] which takes the best of each formalism:

[34] Extended *BNF*.

simply enough, we allow a rule right part to be an r.e. over terminals and nonterminals. Such extended grammars have a nice graphical representation, i.e., the syntax charts to be shown in Chap. 4 on p. 237, which represents the blueprint of the flowchart of a syntax analyzer.

First, since family *CF* is closed under all the r.e. operations, the language family defined by *EBNF* grammars coincides with family *CF*.

In order to appreciate the clarity of the extended rules with respect to the basic ones, we examine a few typical constructs of programming languages.

Example 2.93 (*EBNF grammar of declaration lists*) Consider a list of variable declarations:

char text1, text2;
real temp, result;
int alpha, beta2, gamma;

to be found in the programming languages, with possible syntactic variations.

The alphabet is $\Sigma = \{ c, i, r, v, \text{`,'}, \text{`:'}, \text{`;'} \}$, where characters c, i, and r stand for the keywords char, int, and real, and character v for a variable name. The language of declaration lists is defined by r.e. e_D, below (center):

$$e_D = \big((c \mid i \mid r)\ v\ (\text{`,'}\ v)^*\ \text{`;'} \big)^+$$

$$\begin{cases} D \to D\,E \mid E \\ E \to A\,N\ \text{`;'} \\ A \to c \mid i \mid r \\ N \to v\ \text{`,'}\ N \mid v \end{cases} \qquad \begin{cases} D \to E^+ \\ E \to A\,N\ \text{`;'} \\ A \to c \mid i \mid r \\ N \to v\ (\text{`,'}\ v)^* \end{cases}$$

We apply Algorithm 2.70 on p. 82 and convert this r.e. into the above (left) basic grammar with axiom D for language e_D. The grammar uses recursive rules for nonterminals D and N, is a little longer than the r.e., and is subjectively less perspicuous in evidencing the two-level list structure of the declarations, which is immediately visible in the r.e. Moreover, the choice of the metasymbols A, E, and N is arbitrary and may cause some confusion when several individuals jointly work on a grammar. The grammar in the right part is *EBNF*: in the first and last rule, we respectively see the cross and star operators already present in the r.e. □

Definition 2.94 (EBNF *grammar*) An *extended context-free (or EBNF) grammar* $G = (V, \Sigma, P, S)$ contains exactly $|V| \geq 1$ rules, each one in the form $A \to \alpha$, where A is a nonterminal and α is an r.e. over the alphabet $V \cup \Sigma$. For a better legibility and concision, also other derived operators are permitted in the r.e., like cross, power, and option. □

We add to the preceding language (Example 2.93) some typical block structures.

Example 2.95 (*Algol-like language*) A language block B embraces an optional declarative part D followed by a mandatory imperative part I, between the marks b (begin) and e (end):

$$B \to b\ [\,D\,]\ I\ e$$

The definition of the declarative part is taken from the preceding example:

$$D \to \big(\ (c \mid i \mid r)\ v\ (\text{`,'}\ v)^*\ \text{`;'}\big)^+$$

The imperative part is a list of phrases F separated by semicolon:

$$I \to F\ (\text{`;'}\ F)^*$$

Last, a phrase F can be an assignment a or a block B:

$$F \to a \mid B$$

As an exercise, though worsening legibility, we eliminate as many nonterminals as possible by applying nonterminal expansion (p. 66) to D:

$$B \to b\Big[\big(\ (c \mid i \mid r)\ v\ (\text{`,'}\ v)^*\ \text{`;'}\big)^+\Big]\ I\ e$$

A further expansion of I leads to:

$$B \to b\big(\ (c \mid i \mid r)\ v\ (\text{`,'}\ v)^*\ \text{`;'}\big)^*\ F\ (\text{`;'}\ F)^*\ e$$

Last, symbol F can be eliminated, thus obtaining a one-rule grammar G':

$$B \to b\big(\ (c \mid i \mid r)\ v\ (\text{`,'}\ v)^*\ \text{`;'}\big)^*\ (a \mid B)\ \big(\text{`;'}\ (a \mid B)\ \big)^*\ e$$

This rule cannot be reduced to an r.e., because nonterminal B cannot be eliminated. In fact, it is needed to generate nested blocks, e.g., $b\,b\,\ldots\,e\,e$, by a self-nesting derivation, in agreement with Property 2.82 on p. 91. □

Example 2.96 (*EBNF grammar for JSON*) The *JSON* grammar of Example 2.43, shown in Fig. 2.2 on p. 48, is equivalent to the next *EBNF* form:

$$\left\{\begin{array}{rcl}
S & \to & \langle\text{OBJECT}\rangle \\
\langle\text{OBJECT}\rangle & \to & \text{`\{'}\ (\varepsilon \mid \langle\text{MEMBERS}\rangle)\ \text{`\}'} \\
\langle\text{MEMBERS}\rangle & \to & \langle\text{PAIR}\rangle\ (\text{,}\ \langle\text{PAIR}\rangle)^* \\
\langle\text{PAIR}\rangle & \to & \langle\text{STRING}\rangle : \langle\text{VALUE}\rangle \\
\langle\text{VALUE}\rangle & \to & \langle\text{STRING}\rangle \mid \langle\text{OBJECT}\rangle \mid \langle\text{ARRAY}\rangle \mid \text{num} \mid \text{bool} \\
\langle\text{STRING}\rangle & \to & \text{``}\ (\varepsilon \mid \langle\text{CHARS}\rangle)\ \text{''} \\
\langle\text{ARRAY}\rangle & \to & \text{`['}\ (\varepsilon \mid \langle\text{ELEMENTS}\rangle)\ \text{`]'} \\
\langle\text{ELEMENTS}\rangle & \to & \langle\text{VALUE}\rangle\ (\text{,}\ \langle\text{VALUE}\rangle)^* \\
\langle\text{CHARS}\rangle & \to & \langle\text{CHAR}\rangle^+ \\
\langle\text{CHAR}\rangle & \to & \text{char}
\end{array}\right.$$

A further variant makes use of the optionality operator '[ ]' and replaces the second rule above with ⟨OBJECT⟩ → '{' [⟨MEMBERS⟩] '}' (here square brackets are metasymbols); and similarly for rules STRING and ARRAY. □

Usually, language reference manuals specify grammars by means of *EBNF* rules. Anyway, beware that an excessive grammar conciseness is often contrary to clarity. Moreover, if a grammar is split into smaller rules, it may be easier for the compiler writer to associate simple specific semantic actions to each rule, as we will see in Chap. 5.

2.7.4.1 Derivations and Trees in Extended Grammars

The right part α of an extended rule $A \to \alpha$ of a grammar G is an r.e., which in general derives an infinite set of strings: each of them can be viewed as the right part of a nonextended rule with unboundedly many alternatives. For instance, rule $A \to (\, a\, B\,)^+$ stands for the infinitely many alternatives:

$$A \to a\,B \mid a\,B\,a\,B \mid \ldots$$

The notion of derivation can be defined for extended grammars, too, via the notion of r.e. derivation introduced on p. 20.

Briefly, for an *EBNF* grammar G, consider a rule $A \to \alpha$, where α is an r.e. possibly containing the choice operators star, cross, union, and option. Let α' be a string that derives from α, according to the definition of r.e. derivation, and does not contain any choice operator. For every pair of (possibly empty) strings δ and η, there is a one-step derivation:

$$\delta\, A\, \eta \Rightarrow_G \delta\, \alpha'\, \eta$$

Then one can define a multi-step derivation that starts from the axiom and produces a terminal string, and consequently can define the language generated by an *EBNF* grammar, in the same manner as for basic grammars. An example should suffice to clarify.

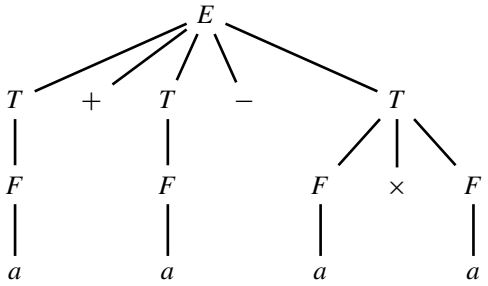
Example 2.97 (*derivation in an EBNF grammar*) The grammar G below:

$$G \begin{cases} E \to [+ \mid -]\, T\, \big(\, (+ \mid -)\, T\, \big)^* \\ T \to F\, \big(\, (\times \mid /)\, F\, \big)^* \\ F \to a \mid \text{'('}\, E\, \text{')'} \end{cases}$$

generates arithmetic expressions with four infix binary operators '+', '−', '×', and '/', two prefix unary operators '+' and '−' (both optional), and parentheses, i.e., subexpressions. Terminal a stands for a numeric argument. Square brackets are metasymbols denoting option. The left derivation:

$$E \Rightarrow T + T - T \Rightarrow F + T - T \Rightarrow a + T - T \Rightarrow a + F - T \Rightarrow a + a - T$$
$$\Rightarrow a + a - F \times F \Rightarrow a + a - a \times F \Rightarrow a + a - a \times a$$

corresponds to the syntax tree:



Notice that the node degree can be unbounded, e.g., a node of type T can have 1, 3, 5, 7, …siblings. Consequently, the tree breadth increases and its height decreases, compared to the tree of an equivalent nonextended grammar. □

Ambiguity in Extended Grammars

It is obvious that an ambiguous BNF grammar remains such also when viewed as an *EBNF* grammar. Moreover, a different form of ambiguity may arise in an *EBNF* grammar, caused by the ambiguity of an r.e. in a rule right part.

Recall that an r.e. is ambiguous (p. 22) if it derives a string through two different left derivations. For instance, the r.e. $a^* b \mid a\, b^*$, numbered $a_1^* b_2 \mid a_3\, b_4^*$, is ambiguous because string $a\, b$ can be derived as $a_1\, b_2$ or $a_3\, b_4$. As a consequence, the extended grammar:

$$S \to a^*\, b\, S \mid a\, b^*\, S \mid c$$

is ambiguous as well.

2.8 More General Grammars and Language Families

We have seen that context-free grammars cover the main constructs occurring in technical languages, namely hierarchical lists and nested structures, but fail with other syntactic structures even as simple as the replica language or the three equal power language on p. 93.

From the early days of language theory, such shortcomings have motivated much research on more powerful formal grammars. It is fair to say that none of the formal models has been successful: the more powerful ones are too obscure and difficult to use, and those marginally superior to the context-free one do not offer significant advantages. In practice, the application of such grammars to compilation has been

episodic and quickly abandoned.[35] Since the basis of all the subsequent developments is the grammar classification due to the linguist Noam Chomsky, it is appropriate to briefly present it for reference, before moving on to more applied topics in the coming chapters.

2.8.1 Chomsky Classification of Grammars

The historical classification of phrase-structure grammars based on the form of the rewriting rules is shown in Table 2.10. Surprisingly enough, very small differences in the rule form determine substantial changes in the properties of the corresponding language family, both in terms of decidability and of algorithmic complexity.

A rewriting rule has a left part and a right part, and both parts are strings on the terminal and nonterminal alphabets Σ and V. The left part is replaced by the right part in the derivation process. Chomsky defined four rule types, characterized as follows:

- a rule of *type 0* can replace an arbitrary nonempty string over terminals and nonterminals, with another arbitrary string
- a rule of *type 1* is more constrained than a type 0 rule: the right part of a rule must be at least as long as the left part
- a rule of *type 2* is context-free: the left part is one nonterminal
- a rule of *type 3* coincides with the unilinear form (p. 85)

For completeness, Table 2.10 lists the names of the automaton model, i.e., abstract string recognition algorithm, that corresponds to each rule type, although the notion of automaton will not be introduced until the next chapter. The language families are strictly included one inside the other from bottom to top, and this inclusion chain justifies the name of hierarchy.

Partly anticipating later matters, the difference between rule types is mirrored by differences in the computational resources needed to recognize the strings. Concerning space, i.e., memory complexity for string recognition, type 3 uses a finite memory, whereas the others need an unbounded memory.

Other properties are worth mentioning, without any claim to completeness. All four language families are closed under union, concatenation, star, reversal, and intersection with a regular language. Yet for other operators their properties differ: for instance, families 1 and 3, but not 0 and 2, are closed under complement.

Concerning the decidability of various properties, the difference between the apparently similar types 0 and 1 is striking. For type 0, it is undecidable (more precisely semi-decidable) whether a string is in the language generated by a gram-

[35] In computational linguistic, some grammar types more powerful than the context-free one have gained acceptance. We mention the possibly best known example: the Tree Adjoining Grammar (TAG) of Joshi, described, e.g., in [18].

Table 2.10 Chomsky grammar classification with corresponding languages and machines

Grammar type	Rule form	Lang. family	Recognizer model
Type 0	$\beta \to \alpha$ with $\alpha,\ \beta \in (\Sigma \cup V)^+$	Recursively enumerable	Turing machine
Type 1 context-dependent context-sensitive	$\beta \to \alpha$ with $\alpha,\ \beta \in (\Sigma \cup V)^+$ and $\lvert\beta\rvert \leq \lvert\alpha\rvert$	Contextual context-depend.	Turing machine with space complexity limited by input length
Type 2 context-free *BNF*	$A \to \alpha$ with $A \in V$ and $\alpha \in (\Sigma \cup V)^*$	Context-free *CF* algebraic	Pushdown automaton
Type 3 unilinear either right-linear or left-linear	$A \to u\,B$ (right) $A \to B\,u$ (left) with $A,\ u \in V,\ \Sigma^*$ and $B \in (V \cup \{\varepsilon\})$	Regular *REG* rational finite-state	Finite automaton

mar. For type 1 the same problem is decidable, though its time complexity is not polynomial. Last, only for type 3 the equivalence problem of two grammars is decidable.

We finish with two examples of grammars and languages of type 1 (context-sensitive).

Example 2.98 (*type 1 grammar of the three equal power language*) The language of three equal powers, proved on p. 93 to be not *CF*, is:

$$L = \left\{ a^n\, b^n\, c^n \mid \ n \geq 1 \right\}$$

Language L is generated by the context-sensitive (type 1) grammar:

$$\begin{cases} 1\colon\ S \ \to\ a\,S\,B\,C \\ 2\colon\ S \ \to\ a\,b\,C \end{cases} \qquad \begin{cases} 3\colon\ C\,B \ \to\ B\,C \\ 4\colon\ b\,B \ \to\ b\,b \end{cases} \qquad \begin{cases} 5\colon\ b\,C \ \to\ b\,c \\ 6\colon\ c\,C \ \to\ c\,c \end{cases}$$

For grammars of type 0 and 1, a derivation cannot be represented as a tree, because a rule left part typically contains more than one symbol. However, the derivation can be visualized as a graph, where the application of a rule such as $B\,A \to A\,B$ is displayed as a bundle of arcs (or hyper-edge) that connect the left part nodes to the right part ones.

Coming from our experience with context-free grammars, we would expect to be able to generate all the sentences through left derivations. But, if we try to generate sentence $a\,a\,b\,b\,c\,c$ by proceeding from left to right:

$$S \overset{1}{\Rightarrow} a\,S\,B\,C \overset{2}{\Rightarrow} a\,a\,b\,C\,B\,C \overset{5}{\Rightarrow} a\,a\,b\,c\,B\,C \quad \text{halt !}$$

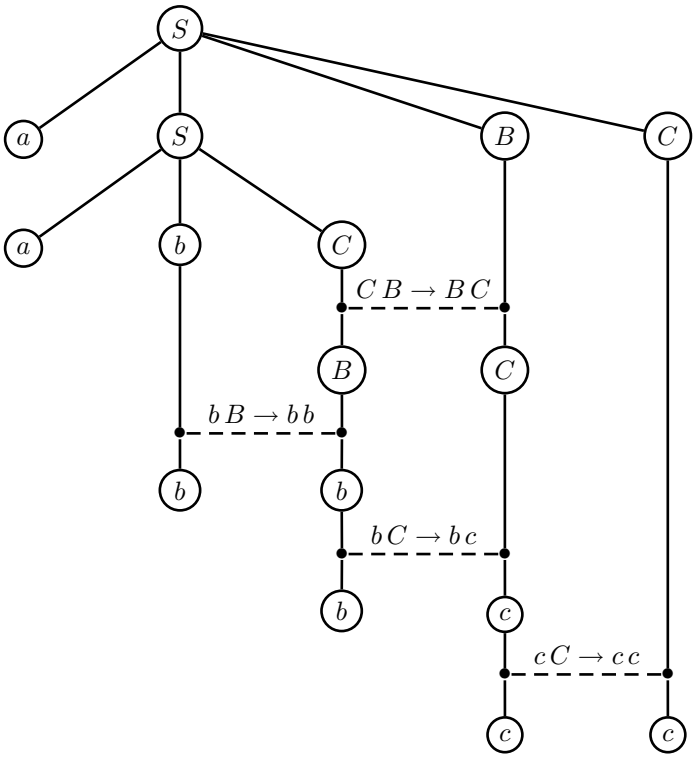


Fig. 2.3 Graph representation of a context-sensitive (type 1) derivation (Example 2.98)

surprisingly, the derivation gets stuck before eliminating all the nonterminals. To generate this string, we need a nonleftmost derivation, shown in Fig. 2.3.

Intuitively, the derivation produces the requested number of letters a with rule 1 (a type 2 self-nesting rule) and then with rule 2. In the sentential form, letters b, B, and C appear with the correct number of occurrences, but mixed up. To produce the valid string, all the letters C must be shifted to the right, by using rule 3: $C\,B \to B\,C$, as far as they reach the suffix position.

First, the application of rule 3 reorders the sentential form, to the aim that letter B becomes adjacent to letter b. This enables rule 4: $b\,B \to b\,b$. Then, the derivation continues in the same manner, by alternating rules 3 and 4, until the sentential form is left with just C as nonterminal. The occurrences of C are transformed into terminals c, by means of rule 5: $b\,C \to b\,c$, followed by repeated applications of rule 6: $c\,C \to c\,c$. □

We stress the finding that the language generated by a type 1 grammar may not coincide with the sentences generated only through left derivations,[36] unlike type 2 grammars. This is a cause of difficulty in string recognition algorithms.

Type 1 grammars have the power to generate the replica language, i.e., lists with agreement between elements, which is a construct we have singled out as practically relevant but exceeding the power of context-free grammars.

Example 2.99 (*type 1 grammar of replica with center*) The language L:

$$L = \left\{ y\, c\, y \mid \ \ y \in \{a,\, b\}^{+} \ \right\}$$

contains sentences such as $a\,a\,b\,c\,a\,a\,b$, where a prefix and a suffix are divided by a central separator c and must be equal.

To simplify the grammar, we assume that sentences are terminated on the right with an end-of-text or *terminator*, i.e., $\dashv$. Here is a grammar for L:

$$\begin{array}{l|l|l|l|l}
S \to X \dashv & X\,A \to X\,A' & A'\,A \to A\,A' & A' \dashv \to a & B'\,a \ \to b\,a \\
X \to a\,X\,A & X\,B \to X\,B' & A'\,B \to B\,A' & B' \dashv \to b & B'\,b \ \to b\,b \\
X \to b\,X\,B & & B'\,A \to A\,B' & A'\,a \to a\,a & X\,a \ \to c\,a \\
 & & B'\,B \to B\,B' & A'\,b \to a\,b & X\,b \ \to c\,b
\end{array}$$

To generate a sentence, the grammar follows this strategy. First it generates a palindrome, say $a\,a\,b\,X\,B\,A\,A$, where symbol X marks the center and the right half is uppercase. Then the right half, modified as $B'\,A\,A$, is reflected and converted into $A'\,A'\,B'$ in a few steps. Last, the primed uppercase symbols are rewritten as $a\,a\,b$ and the center symbol X is converted into c.

We illustrate the derivation of sentence $a\,a\,b\,c\,a\,a\,b$. For legibility, at each step we underline the left part of the rule being applied:

$$\begin{aligned}
&\underline{S} \Rightarrow \underline{X} \dashv \Rightarrow a\,\underline{X}\,A \dashv \Rightarrow a\,a\,\underline{X}\,A\,A \dashv \Rightarrow a\,a\,b\,\underline{X\,B}\,A\,A \dashv \\
&\quad \Rightarrow a\,a\,b\,X\,\underline{B'\,A}\,A \dashv \Rightarrow a\,a\,b\,\underline{X\,A}\,B'\,A \dashv \Rightarrow a\,a\,b\,X\,A'\,\underline{B'\,A} \dashv \\
&\quad \Rightarrow a\,a\,b\,X\,\underline{A'\,A}\,B' \dashv \Rightarrow a\,a\,b\,\underline{X\,A}\,A'\,B' \dashv \Rightarrow a\,a\,b\,X\,A'\,A'\,\underline{B' \dashv} \\
&\quad \Rightarrow a\,a\,b\,X\,A'\,\underline{A'\,b} \Rightarrow a\,a\,b\,X\,\underline{A'\,a}\,b \Rightarrow a\,a\,b\,\underline{X\,a}\,a\,b \Rightarrow a\,a\,b\,c\,a\,a\,b
\end{aligned}$$

For instance, the rule applied in the fifth step is $X\,B \to X\,B'$. □

Notice that if the same strategy used for generation were applied in reverse order, it would check whether a string is a valid sentence. Starting from the given string, the algorithm should store on a memory tape the strings obtained after each reduction, i.e., the reverse of derivation. Such a procedure is essentially a Turing machine computation that never goes out of the tape portion containing the original string, but may overprint its symbols.

[36] For any type 1 grammar, the language generated using just left derivations is, quite surprisingly, context-free.

2.8.2 Further Developments and Final Remarks

The complication of type 1 grammars already for such simple examples should convince the reader of the difficulty of designing and applying context-sensitive rules to more realistic cases. It is a fact that the interaction between type 1 grammar rules is hard to understand and control.

Truly, type 1 and 0 grammars can be viewed as a particular notation for writing algorithms. All sorts of problems can be programmed in this way, even mathematical ones, through the very simple string rewriting mechanism.[37] Not surprisingly, using such an elementary mechanism as the only data and control structure makes the algorithm description very tangled.

Undoubtedly, the development of language theory toward models with a higher computational capacity has a mathematical and speculative interest, but, until now, it has been almost irrelevant for language engineering and compilation. For historical honesty, we mention that context-sensitive grammars have been occasionally considered by language and compiler designers. The programming language Algol 68 was defined with a special class of type 1 grammars termed '2-level grammar,' also known as *VW*-grammar.[38]

Further Attempts: Conjunctive and Boolean Grammars

This notwithstanding, research efforts toward new grammar models have continued and have proposed quite different approaches, the practicality of which is open to debate. We finish with a recent interesting example: the conjunctive grammars and the more general Boolean grammars of Okhotin [21]. To understand the idea behind them, it helps to view each collection of alternative rules $A \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_m$ of an ordinary context-free grammar G as the logical *or* (disjunction) of m rules. In other words, nonterminal A can derive any string of its language $L_A(G)$ starting with the alternative β_1 *or* β_2, etc. What is different in a *conjunctive grammar* is the use of the logical *and* (conjunction, denoted by &) in addition to *or*. For a single nonterminal A, each collection of rules has the form:

$$\left\{ \begin{array}{rcl} A & \to & \left(\alpha_{1,1} \,\&\, \alpha_{1,2} \,\&\, \ldots \,\&\, \alpha_{1,n_1}\right) \mid \\ & & \left(\alpha_{2,1} \,\&\, \alpha_{2,2} \,\&\, \ldots \,\&\, \alpha_{2,n_2}\right) \mid \\ & & \ldots \mid \\ & & \left(\alpha_{m,1} \,\&\, \alpha_{m,2} \,\&\, \ldots \,\&\, \alpha_{2,n_m}\right) \end{array} \right.$$

where each term α is a string of terminal and nonterminal symbols, as usual. Each parenthesized subrule is the conjunction of one or more cases; the parentheses may be omitted since *and* takes precedence over *or*. If all the values $n_1, n_2, \ldots, n_m$ are equal

[37] An extreme case is a type 1 grammar presented in [12] to generate the prime numbers encoded in unary base, i.e., the language $\{ a^n \mid \ n \text{ is a prime number} \}$.

[38] From Van Wijngarten [19]; see also Cleaveland and Uzgalis [20].

to 1, the rule is the ordinary context-free rule $A \to \alpha_{1,1} \mid \cdots \mid \alpha_{m,1}$. We explain how a conjunctive grammar defines a language by means of a familiar example.

Example 2.100 (*conjunctive grammar of the three equal power language*) This language L, already defined by a type 1 grammar in the Example 2.98 above, is the intersection of two context-free languages:

$$L = \left\{ a^+ b^n c^n \mid \quad n \geq 1 \right\} \cap \left\{ a^n b^n c^+ \mid \quad n \geq 1 \right\}$$

and is generated by the following conjunctive grammar:

$$\begin{cases} S \to A\,B \,\&\, D\,C \\ A \to a\,A \mid a \\ B \to b\,B\,c \mid b\,c \\ C \to c\,C \mid c \\ D \to a\,D\,b \mid a\,b \end{cases}$$

In this grammar, there is only one conjunctive rule, which says that S derives a string x if, and only if, the following derivations hold:

$$\left(A\,B \stackrel{*}{\Rightarrow} x\right) \textit{ and } \left(D\,C \stackrel{*}{\Rightarrow} x\right)$$

The rules for the nonterminals A and B are context-free, and string $A\,B$ derives strings $a^+ b^n c^n$. Similarly, string $C\,D$ derives strings $a^n b^n c^+$; therefore, the axiom S derives all and only the valid strings $a^n b^n c^n$. □

Example 2.100 gives the positive impression that conjunctive grammars are an elegant and simple extension of context-free grammars, which permits to define in a natural way some context-sensitive languages. Unfortunately, such simplicity is lost for other quite basic context-sensitive languages such as the *replica with center* of the Example 2.99 above, the conjunctive grammar of which [21] is (subjectively) complicated. Moreover, for the language *replica* (without center), no conjunctive grammar is known, and one has to resort to the more powerful model of Boolean grammars. As the name says, such grammars can use in their rules not just conjunction and disjunction, but also complementation, written '¬'. A rule of the form $A \to \neg\, B$ says that a string is in the language $L_A\,(G)$ if it is in the complement of the language defined by nonterminal B.

Example 2.101 (*Boolean grammar of replica*) We recall the definition:

$$\textit{replica} = L = \left\{ y\,y \mid \quad y \in \{a,\ b\}^+ \right\}$$

This language is defined by the following Boolean grammar G [21]:

$$G \begin{cases} S \rightarrow \neg A\,B\,\&\,\neg B\,A\,\&\,C \\ A \rightarrow X\,A\,X \mid a \\ B \rightarrow X\,B\,X \mid b \\ C \rightarrow X\,X\,C \mid a\,a \mid a\,b \mid b\,a \mid b\,b \\ X \rightarrow a \mid b \end{cases}$$

In grammar G, there is only one Boolean rule, which says that a string x is defined by nonterminal S if:

- x is not defined by the concatenation of the languages of A and B, *and*
- x is not defined by the concatenation of the languages of B and A, *and*
- x is defined by the language of C

We detail the languages involved:

$$L_A(G) = \left\{ u\,a\,v \mid \quad u, v \in (a \mid b)^* \text{with } |u| = |v| \right\}$$
$$L_B(G) = \left\{ u\,b\,v \mid \quad u, v \in (a \mid b)^* \text{ with } |u| = |v| \right\}$$

hence:

$$L_{AB}(G) = \left\{ u\,a\,v\,x\,b\,y \mid \quad u, v, x, y \in (a \mid b)^* \text{ with } |u| = |v|, |x| = |y| \right\}$$
$$L_{BA}(G) = \left\{ u\,b\,v\,x\,a\,y \mid \quad u, v, x, y \in (a \mid b)^* \text{ with } |u| = |v|, |x| = |y| \right\}$$

Notice that language $L_{AB}(G)$ contains all the strings of even length such that a letter a and a letter b are mismatched. Such strings are of the form:

$$u\,a\,v\,x\,b\,y \qquad \text{with } |u| = |x| \textit{ and } |v| = |y|$$

The form of language $L_{BA}(G)$ is similar, with the only difference that, for the mismatched letters, b is on the left of a. Clearly, a string that presents such a mismatch is not a replica.

Then, the first rule of grammar G defines any string that, being defined by C, is of even length and does not contain any mismatching pair, in formula:

$$L_S(G) = \overline{L_{AB}(G)} \cap \overline{L_{BA}(G)} \cap (a\,a \mid a\,b \mid b\,a \mid b\,b)^+ \qquad \square$$

Grammar G of Example 2.101 correctly defines language *replica* and applies the principle that sometimes definitions *ex negativo* may be simpler than those *ex positivo*. But the reasoning behind it is perhaps too mathematically ingenious for a practical use in the technical definitions of languages.

Another difficulty is that the introduction of negation makes such grammars much more difficult to control, since in general the use of negation can lead to logical

contradiction, as when we have a grammar rule $S \to \neg S$. For that reason, a rigorous formulation of a language defined by a Boolean grammar is far from trivial. In particular, the notion of derivation, which is fundamental in all the kinds of Chomsky grammars, is lost and has to be replaced by the formalism of recursive language equations.

To sum up our evaluation of such new models, although conjunctive grammars only moderately increase the power of context-free grammars, they are quite easy to understand. Moreover, the algorithms for parsing with such grammars are similar to the classic algorithms for context-free grammars and may pay an acceptable extra cost in terms of efficiency. On the other hand, general Boolean grammars are too difficult, at least at their current development stage.

In conclusion, we have to admit that the state of the art in formal language theory does not entirely satisfy the need for a powerful and practical formal grammar model, capable of accurately defining the entire range of constructs found in technical languages. Context-free grammars are the best available compromise between expressivity and simplicity. The compiler designer will supplement their weaknesses by other methods and tools, termed semantic, which come from general-purpose programming methodologies. These methods and tools will be introduced in Chap. 5.

References

1. Searls DB (2002) The language of genes. Nature 420:211–217
2. Rozenberg G (ed) (1997) Handbook of graph grammars and computing by graph transformation: volume i. Foundations. World Scientific Publishing Co., Inc., River Edge
3. Comon H, Dauchet M, Gilleron R, Löding C, Jacquemard F, Lugiez D, Tison S, Tommasi M (2007) Tree automata techniques and applications. http://www.grappa.univ-lille3.fr/tata
4. Gecseg F, Steinby M (1997) Tree languages. In: Rozenberg G, Salomaa A (eds) Handbook of formal languages, vol. 3: beyond words. Springer, New York, pp 1–68
5. Crespi Reghizzi S, Giammarresi D, Lonati V (2019, to appear) Two-dimensional models. In: Pin J-É (ed) Handbook of automata theory, vol. I: theoretical foundations. European Mathematical Society, Zürich, pp 281–315
6. Giammarresi D, Restivo A (1997) Two-dimensional languages. In: Rozenberg G, Salomaa A (eds) Handbook of formal languages, vol. 3: beyond words. Springer, New York, pp 215–267
7. Perrin D, Pin JE (2004) Infinite words: automata, semigroups, logic and games. Academic Press, New York
8. Thomas W (1997) Languages, automata, and logic. In: Rozenberg G, Salomaa A (eds) Handbook of formal languages, vol. 3: beyond words. Springer, New York, pp 389–455
9. Hopcroft J, Ullman J (1969) Formal languages and their relation to automata. Addison-Wesley, Reading
10. McNaughton R (1982) Elementary computability, formal languages and automata. Prentice-Hall, Englewood Cliffs
11. Naur P (1963) Revised report on the algorithmic language ALGOL 60. Commun ACM 6:1–33
12. Salomaa A (1973) Formal languages. Academic Press, New York
13. Harrison M (1978) Introduction to formal language theory. Addison Wesley, Reading

14. Autebert J, Berstel J, Boasson L (1997) Context-free languages and pushdown automata. In: Rozenberg G, Salomaa A (eds) Handbook of formal languages, vol. 1: word, language, grammar. Springer, New York, pp 111–174
15. Crespi Reghizzi S, Della Vigna P, Ghezzi C (1976) Linguaggi formali e compilatori. ISEDI, Milano
16. Blattner M, Ginsburg S (1982) Position-restricted grammar forms and grammars. Theor Comput Sci 17:1–27
17. Anselmo M, Giammarresi D, Varricchio S (2003) Finite automata and non-self-embedding grammars. In: CIAA 2002, Springer LNCS, pp 47–56
18. Jurafsky D, Martin JH (2009) Speech and language processing: an introduction to natural language processing, speech recognition, and computational linguistics. Prentice-Hall, Englewood Cliffs
19. Van Wijngarten A (1969) Report on the algorithmic language ALGOL 68. Numer Math 22:79–218
20. Cleaveland J, Uzgalis R (1977) Grammars for programming languages. North-Holland, Amsterdam
21. Okhotin A (2013) Conjunctive and boolean grammars: the true general case of the context-free grammars. Comput Sci Rev 9:27–59



Finite Automata as Regular Language Recognizers

3

3.1 Introduction

Regular expressions and grammars are widely used in the specifications of technical languages, but the actual design and implementation of a compiler need a way to describe the algorithms, termed *recognizers* or *acceptors*, that examine a string and decide if it is a valid sentence of the language. In this chapter we study the recognizers for regular languages, in Chap. 4 those for context-free languages, and in Chap. 5 we extend the recognizer to perform also translation.

The need of recognizing whether a text is valid for a given language is quite common, especially as a first step for text processing or translation. A compiler analyzes a source program to check its correctness; a document processor makes a spell check on the words, and then it verifies the syntax; and a graphic user interface must check that data are correctly entered. Such a control is carried out by a *recognition* procedure, which can be conveniently specified by using minimalist models, termed *abstract machines* or *automata*. The advantages are that the automata do not depend on the programming languages and techniques, i.e., on the implementation, and that in this way their properties (such as their time and memory complexity) are more clearly related with the source language family.

In this chapter we briefly introduce more general automata, and then we focus on those that have a finite memory, because they match the regular language family and have countless applications in computer science and beyond. First, we consider the deterministic machines and we describe some basic methods for cleaning and minimizing. Then, we motivate and introduce nondeterministic models, and we show that they correspond to the unilinear grammars of the previous chapter. Conversion back to deterministic models follows.

A central part deals with the transformations back and forth from regular expressions to automata. The methods presented are: *BMC* for the direction from automaton to regular expression, and Thompson and Berry–Sethi for the other one. The latter

© Springer Nature Switzerland AG 2019

S. Crespi Reghizzi et al., *Formal Languages and Compilation*,
Texts in Computer Science, https://doi.org/10.1007/978-3-030-04879-2_3

method relies on the theory of local regular languages, which have a simpler recognition procedure that makes use of a sliding window.

The ambiguous regular expressions are used for string matching, and they need advanced recognition methods capable of returning the tree structure of a string, i.e., to perform parsing; thus, the *BSP* parser is presented.

Then, we return to the operations of complement and intersection on regular languages, from the standpoint of their recognizers, and we introduce the composition of automata through cartesian product. The chapter ends with a synopsis of the interrelation between finite automata, regular expressions and grammars.

In the compilation process, finite automata have several uses to be described here and in Chap. 5: in lexical analysis for extracting the shortest meaningful strings from a text and for making simple translations, and in static flow analysis and optimization for modeling and analyzing program properties.

3.2 Recognition Algorithms and Automata

To check if a string is valid for a specified language, we need a *recognition algorithm*, a type of algorithm that produces a *yes/no* answer, commonly referred to in the computational complexity studies as a *decision* algorithm. For instance, a famous problem is to decide whether two given graphs are isomorphic: the problem domain (a pair of graphs) differs from the case of language recognition, but the answer is again yes/no.

For the string membership problem, the input domain is a set of strings over an alphabet Σ. The application of a recognition algorithm α to a given string x is denoted as $\alpha\,(x)$. We say that a string x is *recognized* or *accepted* if it holds $\alpha\,(x) = yes$, otherwise it is *rejected*. The language recognized is denoted $L\,(\alpha)$ and is the set of accepted strings:

$$L\,(\alpha) = \left\{\, x \in \Sigma^* \mid \;\; \alpha\,(x) = yes \,\right\}$$

A recognition algorithm is usually assumed to terminate for every input, so that the membership problem is decidable. However, it may happen that, for some string x, the algorithm does not terminate; i.e., the value of $\alpha\,(x)$ is undefined; hence string x is not a valid sentence of language $L\,(\alpha)$. In such a case we say that the membership problem for L is *semidecidable*, or also that language L is recursively enumerable.

In principle, if the membership problem of an artificial language is semidecidable, we cannot exclude that, for some input strings, the compiler will fall into an endless loop. In practice we do not have to worry about such decidability issues, central as they are for computation theory,[1] because in language processing the only language families of concern are decidable, and efficiently so.

[1] Many books cover the subject, e.g., Bovet and Crescenzi [1], Floyd and Beigel [2], Hopcroft and Ullman [3], Kozen [4], McNaughton [5] and Rich [6].

Incidentally, even within the family of context-free languages, some decision problems, different from string membership, are undecidable: in Chap. 2 we have mentioned the problem of deciding if two grammars are weakly equivalent, and the one of checking if a grammar is ambiguous.

In fact, recognizing a string is just the first step of the compilation process. In Chap. 5 we will study the translation from a language to another: clearly the codomain (or image) of a translation algorithm is much more complex than a pure yes/no, since it is itself a set of strings, the target (or destination) language.

All the algorithms, including those for string recognition, may be ranked in complexity classes, measured by the amount of computational resources (time or memory, i.e., space) required to solve the problem. In the field of compilation, it is common to consider time rather than space complexity. With rare exceptions, all the problems of interest for compilation have a low time complexity: linear or at worst polynomial with respect to the size of the problem input. Time complexity is closely related with electric power consumption, which is a significant parameter for portable programmed devices.

A tenet of computation theory is that the complexity of an algorithm should be measured as the number of steps, rather than as the actual execution time spent by a program implementing the algorithm. The reason is that the complexity should be a property of the algorithm and should not depend on the actual implementation and processing speed of a particular computer. Even so, several choices are open for modeling a computational step: they may range from a Turing machine move, to a machine instruction or to a high-level language statement. Here we consider a step to be an elementary operation of an abstract machine or automaton, as customary in all the theoretical and applied studies on formal languages. This approach has several advantages: it gives evidence to the relation between the algorithmic complexity and the family of languages under consideration; it allows to reuse optimized abstract algorithms with a low cost of adaptation to the language, and it avoids a premature commitment and many tedious details of implementation. Moreover, sufficient hints will be given for a programmer to easily transform the automaton into a program, by hand or by using widespread compiler generation tools.

3.2.1 A General Automaton

An automaton or abstract machine is an ideal computer that features a very small set of simple instructions. Starting from the Turing machine of the 1930s, the research on abstract machines has spawned many models, but only few of them are important for the scope of the book.[2] In its most general form, a recognizer is schematized in Fig. 3.1. It comprises three parts: input tape, control unit and (auxiliary) memory. The

[2] Other books have a broader and deeper coverage of automata theory, such as Salomaa [7], Hopcroft and Ullman [3,8], Harrison [9], Shallit [10] and the handbook [11]; for finite automata, a specific reference is Sakarovitch [12].

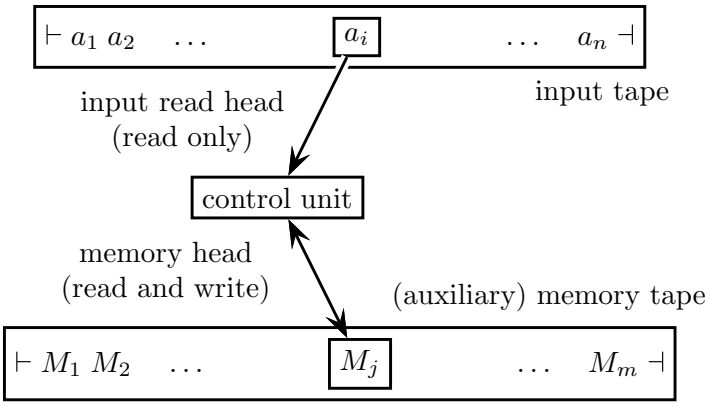


Fig. 3.1 General model (Turing machine) of a recognizer automaton

control unit has a limited store, represented by a finite set of states; on the other hand, the auxiliary memory has an unbounded capacity. The read-only input tape contains the given input or *source* string, one character per case, and the string length n is 0 if the string is empty. The tape cases on the left and right of the input string contain two delimiters: the start-of-text mark ⊢, and the end-of-text mark or terminator ⊣.

A peculiarity of such automaton is that the auxiliary memory is also a tape, instead of the random access memory or counter memory used in other computational models. The memory tape can be read and written, and contains a string of $m \geq 0$ symbols from another alphabet (called memory alphabet).

The automaton can perform the following instantaneous actions: reading the *current character* a_i from input, shifting the read head, reading the *current symbol* M_j from memory and replacing it with another symbol, moving the memory head and changing the *current state* of the control unit to the *next* one.

The automaton processes the source by performing a series of moves; the choice of a move depends on the current two symbols (input and memory) and on the current state. A move may have the following effects, some of which may be missing:

- shifting the input head to the left or right by one position
- overwriting the current memory symbol with another one, and shifting the memory head to the left or right by one position
- changing the state of the control unit.

A machine is *unidirectional* if the input head only moves from left to right: this is the model to be considered in the book, because it well represents how the text is usually processed. For the unidirectional machines the start-of-text mark is superfluous.

At any time, the future behavior of the machine depends on a 3-tuple, called an (instantaneous) *configuration*, made by the following pieces of information:

- the suffix of the input string still unread, which lies on the right of the input head
- the contents of the memory tape and the position of the memory head
- the state of the control unit.

The *initial configuration* has the input head positioned on character a_1, i.e., just to the right of the start-of-text mark, the control unit in an initial state and the memory containing a specific starting symbol (or sometimes a fixed string).

Then the machine performs a *computation*, i.e., a sequence of moves, that leads to new configurations. If for a configuration at most one move can be applied, the change of configuration is *deterministic*. A *nondeterministic* (or indeterministic) automaton is essentially a manner of representing an algorithm that in some situations may explore alternative paths.

A *configuration* is *final* if the control unit is in a state specified as final and the input head is on the terminator. Sometimes, instead of being, or in addition to being in a final state, a final configuration is qualified by the fact that the memory tape contains a specified symbol or string: a frequent choice is for the memory to be empty.

The *source string* x is *accepted* if the automaton, starting in the initial configuration with $x \dashv$ as input, performs a computation that leads to a final configuration (a nondeterministic automaton may reach a final configuration by different computations). The *language accepted or recognized* by the machine is the set of accepted strings.

Notice that a computation terminates either when the machine has entered a final configuration or when in the current configuration no move can be applied any longer. In the latter case the source string is not accepted by that computation, but it may be accepted by another computation if the machine is nondeterministic.

Two automata that accept the same language are called *equivalent*. Of course, two machines, though equivalent, may belong to different models or may have different computational complexities.

Turing Machine

The preceding description (see Fig. 3.1) substantially reproduces the automaton model introduced by A. Turing in 1936 and widely taken as the best available formalization of any sequential algorithm. The family of languages accepted is termed *recursively enumerable*. In addition, a language is termed *decidable* (or recursive) if there exists a Turing machine that accepts it and halts for every input string. The family of decidable languages is smaller than the one of recursively enumerable languages.

Such a machine is the recognizer of two of the language families of the Chomsky classification (p. 105). The languages generated by type 0 grammars are exactly the recursively enumerable ones. The languages of context-sensitive or type 1 grammars, on the other hand, correspond to the languages accepted by a Turing machine that is constrained in its use of the memory: the length of the memory tape is bounded by the length of the input string.

Turing machines are not relevant for practical applications, but they are a significant comparison term for the practical machine models used to recognize and transform technical languages. Such models can be viewed as Turing machines with drastic memory limitations. When no auxiliary memory is available, we have the finite-state or simply finite automaton, the most fundamental type of computing machine, which corresponds to the regular languages. If the memory is organized as an *LIFO* (last in first out) store, the machine is termed a pushdown automaton (studied in Chap. 4) and recognizes the context-free languages.

The memory limitations have a profound effect on the properties and in particular on the performance of automata. Considering the worst-case time complexity, which is a primary parameter for program efficiency, a finite automaton is able to recognize a string in linear time, or more exactly in real time, that is, with a number of steps equal to the input length. In contrast, a space-bounded Turing machine may take a non-polynomial time to recognize a context-sensitive language, another reason making it unpractical to use. The context-free language recognizers lie in between: the number of steps is bounded by a polynomial of small degree of the input string length.

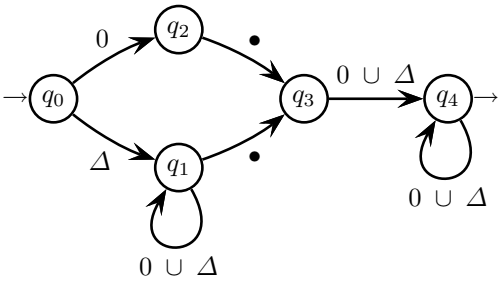
3.3 Introduction to Finite Automata

Finite automata are surely the simplest and most fundamental abstract computational device. Their mathematical theory is very stable and deep, and they are able to support innumerable applications in diverse areas, from digital circuit design to system model checking and speech recognition, to mention just a few. Our presentation will focus on some aspects important for language and compiler design, but, to make the presentation self-contained, we briefly introduce the essential definitions and theoretical results.

Conforming to the general scheme, a finite automaton comprises what follows: an input tape with the source string $x \in \Sigma^*$; a control unit; and a reading head, which is initially placed on the first character of x and scans the string as far as the end, unless an error occurs before. Upon reading a character, the automaton updates the control unit state and advances its reading head. Upon reading the last character, the automaton accepts string x if, and only if, its state is accepting (final).

A well-known representation of an automaton is by a *state-transition diagram* or *graph*. This is a directed graph with the states as nodes. Each arc is labeled with a terminal and represents the state change or *transition* caused by reading the terminal.

state-transition diagram (graph)



state-transition table

current state	*current character* 0 1 ... 9	•
$\rightarrow q_0$	q_2 q_1 ... q_1	–
q_1	q_1 q_1 ... q_1	q_3
q_2	– – ... –	q_3
q_3	q_4 q_4 ... q_4	–
$q_4 \rightarrow$	q_4 q_4 ... q_4	–

Fig. 3.2 State-transition diagram or graph (left) and state-transition table (right) for numerical (fixed-point decimal) constants (Example 3.1)

Example 3.1 (*numerical constants*) The set L of fixed-point decimal constants has alphabet $\Sigma = \Delta \cup \{ 0, \bullet \}$, where $\Delta = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9 \}$ is the set of nonzero digits. The regular expression (r. e.) of L is:

$$L = \big(0 \cup \Delta \ (0 \cup \Delta)^*\big) \bullet (0 \cup \Delta)^+$$

The recognizer of language L is specified by the *state-transition diagram* or by the equivalent *state-transition table* in Fig. 3.2. For convenience, if two or more arcs with distinct labels span the same two nodes, only one arc is drawn with multiple labels. Thus, the arc $q_0 \xrightarrow{\Delta} q_1$ represents a bundle of nine arcs with labels 1, 2, …, 9 and can be equivalently written as $q_0 \xrightarrow{1,\,2,\,\ldots\,9} q_1$. The initial and final states q_0 and q_4 are marked by an incoming and outgoing arrow "$\rightarrow$", respectively. The transition table is the incidence matrix of the graph: for each pair (*current state*, *current character*) the corresponding matrix entry contains the next state.

Given the source string $0 \bullet 2 \bullet$, the automaton transits through the states q_0, q_2, q_3 and q_4. In the last state, since no transition allows reading the character •, the computation stops before consuming the entire input, and the source string is rejected. On the other hand, the input string $0 \bullet 2\,1$ would be accepted. If we prefer to modify the language so that constants such as $3\,0\,5 \bullet$, which just ends with a decimal point, are legal, then the state q_3 must be marked as final, too. □

We have seen that an automaton may have two or more final states, but only one initial state, otherwise it would not be deterministic.

3.4 Deterministic Finite Automata

The previous ideas are formalized in the next definition.

Definition 3.2 (*deterministic finite automaton*) A *deterministic finite automaton* (*DFA*) M comprises five items:

Q	the *state set* (finite and not empty)
Σ	the *input or terminal alphabet*
$\delta\colon Q \times \Sigma \to Q$	the *transition function*
$q_0 \in Q$	the *initial state*
$F \subseteq Q$	the *set of final states* (may be empty). □

The transition function δ specifies the moves: the meaning of $\delta\,(q,\ a) = r$ is that the machine M, in the current state q, reads an input symbol a and moves to the next state r. If the value $\delta\,(q,\ a)$ is undefined, automaton M stops, and we assume it enters the error state (more on that later). As an alternative notation, we indicate the move from state q to state r reading character a, as $q \xrightarrow{a} r$, i.e., by the corresponding arc of the state-transition graph.

Automaton M processes a non-empty string x by a series of moves. Take $x = a\,b$: on reading the first character, the first step $\delta\,(q_0,\ a) = q_1$ leads to state q_1, and then to state q_2 by the second step $\delta\,(q_1,\ b) = q_2$. In short, instead of writing $\delta\left(\delta\,(q_0,\ a),\ b\right) = q_2$, we combine the two steps into one $\delta\,(q_0,\ a\,b) = q_2$, to say that on reading string $a\,b$ the machine M moves to state q_2. Notice that now the second argument of function δ is a string. A special case is the empty string, for which we assume no change of state:

$$\forall\, q \in Q \quad \delta\,(q,\ \varepsilon) = q$$

Following the stipulations above, the transition function δ has the domain $Q \times \Sigma^*$, and it is recursively defined as follows:

$$\delta\,(q,\ y\,a) = \delta\left(\delta\,(q,\ y),\ a\right) \qquad \text{where } a \in \Sigma \text{ and } y \in \Sigma^*$$

We observe a correspondence between the values of δ and the paths in the state-transition graph: $\delta\,(q,\ y) = q'$ if, and only if, there exists a path from node q to node q', such that the concatenated labels of the path arcs make string y. We say that string y is the *label* of the path, and the path itself represents a *computation* of automaton M.

A string x is *recognized* or accepted by automaton M if it is the label of a path from the initial state to a final state, i.e., $\delta\,(q_0,\ x) \in F$. Notice that the empty string ε is recognized if, and only if, the initial state is also final, i.e., $q_0 \in F$. The *language* $L\,(M)$ *recognized* or accepted by automaton M is:

$$L\,(M) = \left\{\, x \in \Sigma^* \mid \ \delta\,(q_0,\ x) \in F \,\right\}$$

The languages accepted by such automata are termed *finite-state recognizable*. Two (finite) automata are *equivalent* if they accept the same language.

Example 3.3 (*Example* 3.1 *continued*) The automaton M of p. 121 is defined by:

$Q = \{ q_0,\ q_1,\ q_2,\ q_3,\ q_4 \}$	state set
$\Sigma = \{ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 0,\ \bullet \}$	(input) alphabet
$q_0 = q_0$	initial state
$F = \{ q_4 \}$	final state set

Examples of transitions:

$$\begin{aligned} \delta\,(q_0,\ 3\ \bullet\ 1) &= \delta\left(\delta\,(q_0,\ 3\,\bullet),\ 1\right) \\ &= \delta\left(\delta\left(\delta\,(q_0,\ 3),\ \bullet\right),\ 1\right) \\ &= \delta\left(\delta\,(q_1,\ \bullet),\ 1\right) \\ &= \delta\,(q_3,\ 1) = q_4 \end{aligned}$$

Since it holds $q_4 \in F$, string $3 \bullet 1$ is accepted. On the contrary, since state $\delta\,(q_0,\ 3\,\bullet) = q_3$ is not final, string $3\,\bullet$ is rejected, as well as string $0\,2$ because function $\delta\,(q_0,\ 0\,2) = \delta\,(\delta\,(q_0,\ 0),\ 2) = \delta\,(q_2,\ 2)$ is undefined. □

The automaton executes one step per input character, and the total number of steps executed equals the length of the input string. Such machine is very efficient and recognizes the input in real time by a single left-to-right scan.

3.4.1 Error State and Total Automaton

If a move is not defined in a state q when reading an input character a, we say that the automaton falls into the *error state* q_{err}. The automaton can never exit from the error state, thus justifying its other name of *sink* or *trap* state. Obviously the error state is not final.

The state-transition function δ can be made *total* by adding the error state and the transitions from/to it:

$$\forall\, q \in Q \quad \forall\, a \in \Sigma \quad \text{if } \delta\,(q,\ a) \text{ is undefined, then set } \delta\,(q,\ a) = q_{err}$$

$$\forall\, a \in \Sigma \quad \text{set } \delta\,(q_{err},\ a) = q_{err}$$

The recognizer of numerical constants of Fig. 3.2, completed with the error state, is shown in Fig. 3.3. Since a computation entering the error state is trapped there and cannot reach a final state, the total automaton accepts the same language as

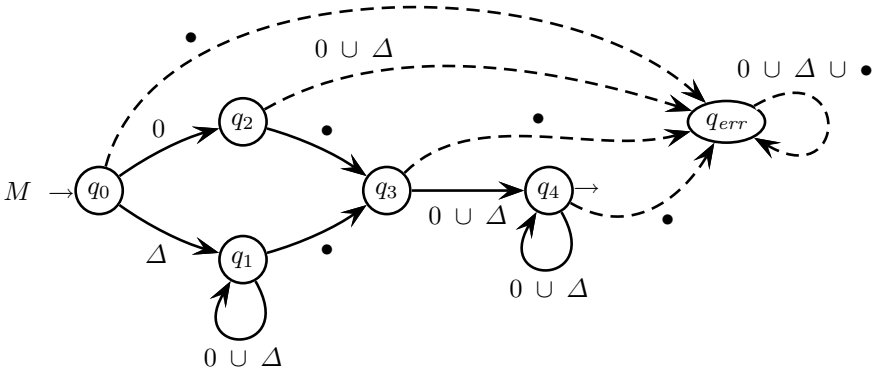


Fig. 3.3 Recognizer of numerical constants completed with the sink (error or trap) state

the original one. It is customary not to represent the error state, nor specifying the transitions that touch it.

3.4.2 Clean Automaton

An automaton may contain useless parts that do not contribute to any accepting computation, which are best eliminated. Notice that the following concepts hold for nondeterministic finite automata as well, and in general for any kind of abstract machine.

A state q is *reachable* from a state p if a computation exists going from p to q. A state is *accessible* if it can be reached from the initial state, and it is *post-accessible* if a final state can be reached from it. A state is called *useful* if it is accessible and post-accessible; otherwise it is *useless*. In other words, a useful state lays on some path from the initial state to a final one. An automaton is *clean* or *reduced* if every state is useful.

Property 3.4 (clean automaton existence) *For every finite automaton there exists an equivalent clean automaton.* □

To construct the clean machine, we identify the useless states and we delete them together with all the arcs that touch them.

Example 3.5 (*elimination of useless states*) Figure 3.4 shows a machine with useless states and the corresponding clean machine. Notice that the error state is never post-accessible, hence always useless. □

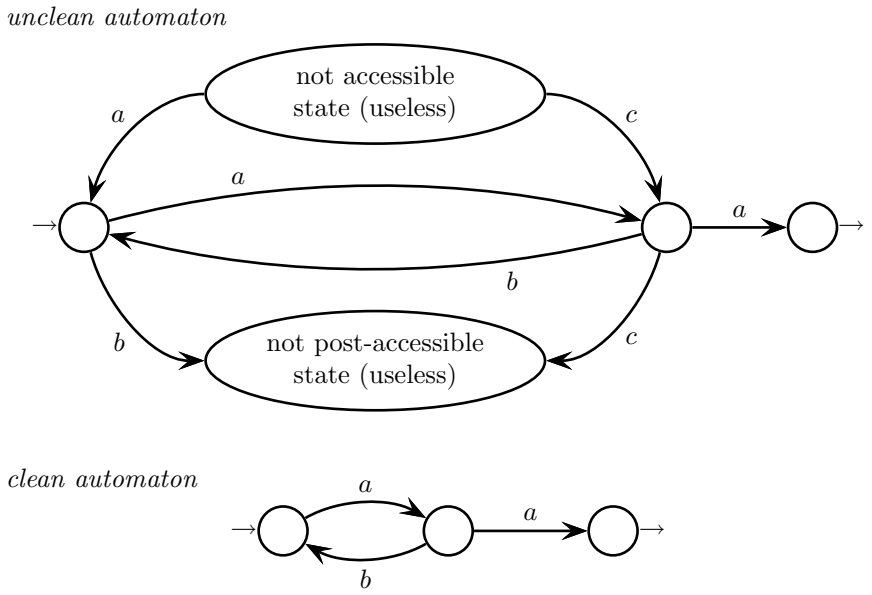


Fig. 3.4 Automaton with useless states (top) and equivalent clean automaton (bottom)

3.4.3 Minimal Automaton

We focus on the automata that recognize the same language, using different state sets and transition functions. A central property says that out of such equivalent machines, there exist one, and only one, that is the smallest in the following sense.

Property 3.6 (minimal automaton uniqueness) *For every finite-state language, the (deterministic) finite recognizer minimal with respect to the number of states is unique (apart from renaming the states).* □

Conceptually, this statement is fundamental, as it permits to represent every collection of equivalent machines by a standard one, which moreover is minimal; in practice, for compiler applications of finite automata, the requirement of minimality is not common.

We describe the standard minimization algorithm. First, we need to introduce a binary relation between equivalent states, and then we show how to compute it.[3] We assume that the given automaton is clean. Some of its states can be redundant, in the sense that they can be merged at no consequence for the strings accepted or

[3] Other subtler and more efficient algorithms have been invented. We refer the reader to the survey in [13].

rejected. Any two such states are termed *undistinguishable* from each other, and the corresponding binary relation is termed *undistinguishability* (or of Nerode).

Definition 3.7 (*state undistinguishability*) Two states p and q are *undistinguishable* if, and only if, for every input string $x \in \Sigma^*$, either both the next states $\delta(p, x)$ and $\delta(q, x)$ are final, or neither one is. The complementary relation is termed *distinguishability*. □

Spelling out the condition, two states p and q are undistinguishable if, by starting from them and scanning the same arbitrarily chosen input string x, it never happens that one computation reaches a final state and the other does not. Notice that:

1. The sink state q_{err} is distinguishable from every other state p, which by hypothesis is post-accessible; therefore for some string x, $\delta(p, x)$ is a final state, while $\delta(q_{err}, x) = q_{err}$.
2. Two states p and q are distinguishable if p is final and q is not, because it holds $\delta(p, \varepsilon) \in F$ and $\delta(q, \varepsilon) \notin F$.
3. Two states p and q are distinguishable if, for some input character a, their next states $\delta(p, a)$ and $\delta(q, a)$ are distinguishable.

In particular, notice that state p is distinguishable from state q if the sets of labels of the arcs outgoing from p and from q are different. In that case, there exists a character a such that the move from state p reaches a state p', while the move from q is not defined; i.e., it reaches the sink; then, from the condition 3 above, p and q are distinguishable.

Undistinguishability as a relation is symmetric, reflexive and transitive; i.e., it is an equivalence relation. Its equivalence classes are computed by a straightforward procedure next described by means of an example.

Example 3.8 (*equivalence classes of undistinguishable states*) For the automaton M of Fig. 3.5 (top), we construct a square table of size $|Q| \times |Q|$ to contain the undistinguishability relation. More precisely, since the relation is symmetric and reflexive, it suffices to fill the cases under the main diagonal, which form a triangular matrix of side $|Q| - 1$.

The procedure marks with "×" the case (p, q), when it discovers that states p and q are distinguishable. Initially, we mark with "×" every case (p, q) such that only one state is final:

q_1			
q_2	×	×	
q_3	×	×	
	q_0	q_1	q_2

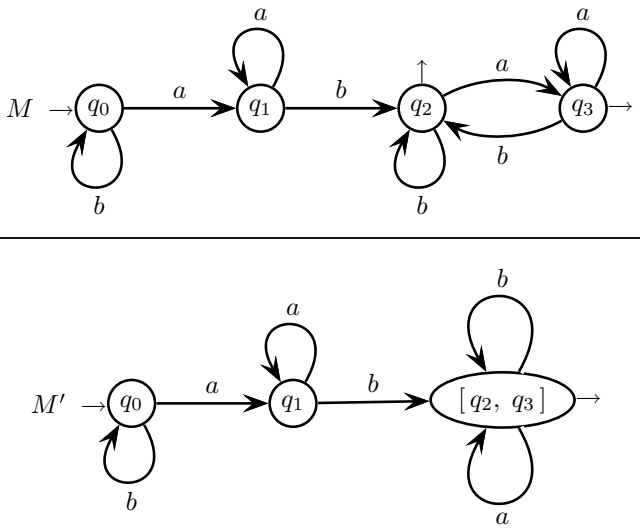


Fig. 3.5 Non-minimal automaton M of Example 3.8 (top). Minimal automaton M' (bottom)

Then, we examine each unmarked case $(p,\ q)$, and for every character a, we consider the next states $r = \delta\,(p,\ a)$ and $s = \delta\,(q,\ a)$. If case $(r,\ s)$ is marked, meaning that state r is distinguishable from state s, then we also mark the case $(p,\ q)$ because states p and q too are distinguishable. Otherwise, if case $(r,\ s)$ is not marked, the state pair $(r,\ s)$ is written into case $(p,\ q)$, as a future obligation, if case $(r,\ s)$ will get marked, to mark case $(p,\ q)$ as well.

The pairs of next states are listed in the table on the left. The result of this step is the table on the right:

	q_0	q_1	q_2
q_1	(1, 1) (0, 2)		
q_2	×	×	
q_3	×	×	(3, 3) (2, 2)

	q_0	q_1	q_2
q_1	×		
q_2	×	×	
q_3	×	×	(3, 3) (2, 2)

Notice that case $(1,\ 0)$ is marked because the state pair $(0,\ 2) \equiv (2,\ 0)$ was already marked. Now all the cases are filled and the algorithm terminates. The cases not marked with "$\times$" identify the undistinguishable state pairs, here only the pair $(q_2,\ q_3)$. By definition, an equivalence class contains all the pairwise undistinguishable states. Here the equivalence classes are $[q_0]$, $[q_1]$ and $[q_2,\ q_3]$. □

It is worth analyzing what happens when the transition function δ is not total. To this end, imagine to modify the automaton M of Fig. 3.5 as follows. We erase the self-loop $\delta\ (q_3,\ a) = q_3$, by redefining the function as $\delta\ (q_3,\ a) = q_{err}$. As a consequence, states q_2 and q_3 become distinguishable, because $\delta\ (q_2,\ a) = q_3$ and $\delta\ (q_3,\ a) = q_{err}$. Now every equivalence class is a singleton, meaning that the automaton is minimal.

3.4.3.1 Construction of the Minimal Automaton

The minimal automaton M', equivalent to the given automaton M, has for states the equivalence classes of the undistinguishability relation. It is simple to construct the transition function of M'. Machine M' contains the arc:

$$\overbrace{[\ldots,\ p_r,\ \ldots]}^{C_1} \xrightarrow{b} \overbrace{[\ldots,\ q_s,\ \ldots]}^{C_2}$$

between the equivalence classes C_1 and C_2, if, and only if, machine M contains an arc $p_r \xrightarrow{b} q_s$, between two states, respectively, belonging to the two classes. Notice that the same arc of M' may derive from several arcs of M.

Example 3.9 (*Example* 3.8 *continued*) The minimal automaton M' is in Fig. 3.5 (bottom). It has the smallest number of states of all equivalent machines. In fact, we could easily check that by merging any two states, the resulting machine would not be equivalent and would accept a language larger than the original. □

From this, it is now straightforward to check whether two given machines are equivalent. First, minimize both machines, and then compare their state-transition graphs to see whether: they are isomorphic, corresponding arcs have the same label, and corresponding states have the same attributes as initial and final.[4]

Obvious economy reasons would make the minimal machine a preferable choice, but the saving is often negligible for the cases of concern in compiler design. What is more, in certain situations state minimization should be avoided. When the automaton is enriched with actions computing an output function, to be seen in Sect. 3.9 and in Chap. 5, two states that are undistinguishable for the recognizer may produce different output actions, and merging would spoil the intended function.

Finally, we anticipate that the uniqueness property of the minimal automaton does not hold for the nondeterministic machines to be introduced next.

[4] Other more efficient equivalence tests do without first minimizing the automaton.

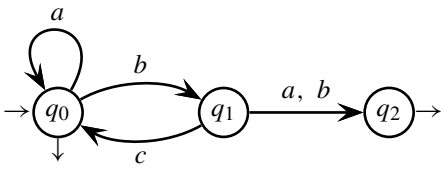
3.4.4 From Automaton to Grammar

Without much effort we are going to realize that finite automata and unilinear (or type 3) grammars of Definition 2.74 on p. 85 are alternative but equivalent notations to define the same language family. First, we show how to construct a right-linear grammar equivalent to a given automaton.

The nonterminal set V of the grammar is the state set Q of the automaton, and the axiom is the initial state. For each move $q \xrightarrow{a} r$, the grammar has the rule $q \to a\, r$. If state q is final, the grammar has also a terminal rule $q \to \varepsilon$.

Clearly, a one-to-one (bijective) correspondence holds between automaton computations and grammar derivations: a string x is accepted by the automaton if, and only if, it is generated by a derivation $q_0 \overset{+}{\Rightarrow} x$.

Example 3.10 (*from automaton to right-linear grammar*) The automaton and corresponding grammar are:



$$\begin{cases} q_0 \to a\, q_0 \mid b\, q_1 \mid \varepsilon \\ q_1 \to c\, q_0 \mid a\, q_2 \mid b\, q_2 \\ q_2 \to \varepsilon \end{cases}$$

Sentence $b\, c\, a$ is recognized in three steps by the automaton and derives from the axiom in $3 + 1$ steps:

$$\underbrace{q_0 \overset{q_0 \to b\, q_1}{\Longrightarrow} b\, q_1 \overset{q_1 \to c\, q_0}{\Longrightarrow} b\, c\, q_0 \overset{q_0 \to a\, q_0}{\Longrightarrow} b\, c\, a\, q_0}_{\text{automaton moves}} \underbrace{\overset{q_0 \to \varepsilon}{\Longrightarrow} b\, c\, a\, \varepsilon}_{\text{acceptance}} = b\, c\, a$$

Observe that the grammar contains empty rules, but of course it can be turned into the non-nullable normal form through the transformation on p. 66. First, we find the set of nullable nonterminals $Null = \{\, q_0,\ q_2 \,\}$, and then we construct the equivalent rules:

$$q_0 \to a\, q_0 \mid b\, q_1 \mid a \mid \varepsilon \qquad q_1 \to c\, q_0 \mid a\, q_2 \mid b\, q_2 \mid a \mid b \mid c$$

Now, symbol q_2 is deleted because its only rule is empty. At last, grammar cleaning produces the rules:

$$q_0 \to a\, q_0 \mid b\, q_1 \mid a \mid \varepsilon \qquad q_1 \to c\, q_0 \mid a \mid b \mid c$$

Of course, the empty rule $q_0 \to \varepsilon$ must stay there because it is needed to make the empty string into the language. □

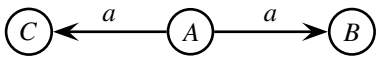
We observe that in the non-nullable form, a move entering a final state r, $q \xrightarrow{a} r$, may correspond to two grammar rules: $q \to a\, r \mid a$, the former producing the nonterminal r, the latter of the terminal type.

We have seen that conversion from automaton to grammar is straightforward, but to make the reverse transformation, we need to modify the automaton definition to allow nondeterministic behavior.

3.5 Nondeterministic Automata

A right-linear grammar may contain two alternative rules:

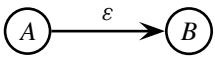
$A \to a\, B \mid a\, C$
where $a \in \Sigma$ and $A,\ B,\ C \in V$



that start with the same character a. In this case, by converting the rules into machine transitions, two arcs with an identical label exit from the same state A and enter two distinct states B and C. This means that in state A, reading character a, the machine is free to choose the next state to enter, and its behavior is not deterministic. Formally, the transition function takes two values, $\delta\,(A,\ a) = \{\ B,\ C\ \}$.

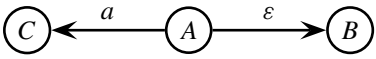
Similarly, a copy rule:

$A \to B \quad$ where $B \in V$



is represented by an unusual machine transition from state A to state B, which does not read any terminal character (it would be odd to say that it reads the empty string).

A move or arc that does not read an input character is termed a *spontaneous* or *epsilon move* (ε-move). Spontaneous moves too may cause nondeterminism, as in the following situation:



where in state A the automaton can choose to move without reading to B, or to read the current character, and if it is a, to move to C.

3.5.1 Motivation of Nondeterminism

The mapping from grammar rules to transitions has forced us to introduce identically labeled arcs with different destinations and spontaneous moves, which are the main forms of nondeterminism. Since this may appear a useless theoretical concept, we hasten to list its multiple pros.

3.5.1.1 Concision

Defining a language with a nondeterministic machine often results in a more readable and compact definition, as in the next example.

Example 3.11 (*penultimate character*) Every sentence of language L_2 is characterized by the presence of a letter b in the penultimate position (2nd from the end), e.g., $a\,b\,a\,a\,b\,b$. The r.e. and the nondeterministic recognizer N_2 are shown in Fig. 3.6.

We have to explain how machine N_2 works: given an input string, the machine seeks a computation, i.e., a path from the initial state to the final, labeled with the input string, and if it succeeds, the string is accepted. Thus, string $b\,a\,b\,a$ is recognized with the computation:

$$q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2$$

Notice that other computations are possible, for instance the path:

$$q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0$$

fails to recognize, because it does not reach the final state.

The same language is accepted by the deterministic automaton M_2 as shown in Fig. 3.7. Clearly, this machine is not just larger than the nondeterministic type in Fig. 3.6, but it makes it less perspicuous that the penultimate character must be b. □

To strengthen the argument, consider a generalization of language L_2 to language L_k, such that, for some $k \geq 2$, the k-th character before the last is a letter b. With little thought, we see that the nondeterministic automaton would have $k + 1$ states, while one could prove that the number of states of the minimal deterministic machine is an

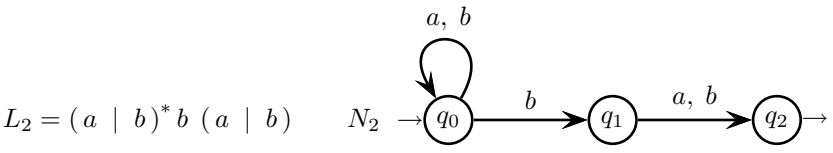


Fig. 3.6 R.e. and corresponding nondeterministic machine N_2 checking that the penultimate character is b

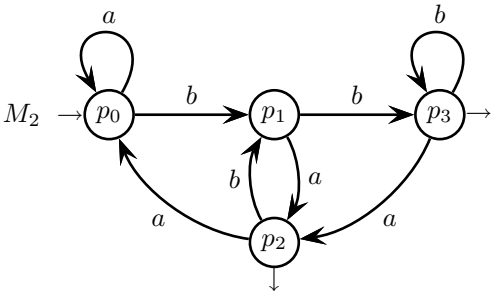


Fig. 3.7 Deterministic recognizer M_2 of the strings having a letter b next to the last one

exponential function of k. In conclusion, nondeterminism sometimes allows much shorter definitions.

3.5.1.2 Left–Right Interchange and Language Reflection

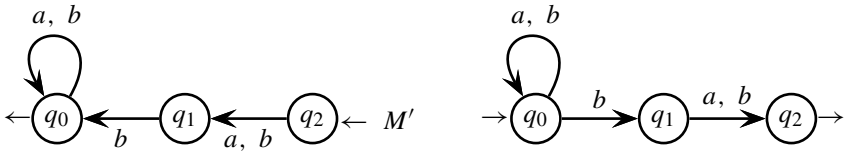
Nondeterminism also arises in string reversal, when a given deterministic machine is transformed to recognize the reversal L^R of language L. This is sometimes required when for some reason, a string must be scanned from right to left.

The new machine is straightforward to derive: interchange the initial and final states[5] and reverse all the arrows. Clearly, this may give birth to nondeterministic transitions.

Example 3.12 (*automaton of reversed language*) The language with b as penultimate character (L_2 of Fig. 3.6) is the reflection of the language with the second character equal to b:

$$L' = \{\, x \in \{a,\ b\}^* \mid \ b \text{ is the second character of } x \,\} \qquad L_2 = (L')^R$$

Language L' is recognized by the deterministic automaton M' (left):



[5] If the machine has multiple final states, multiple initial states result, thus causing another form of nondeterminism to be dealt with later.

By transforming the automaton M' as explained above, we obtain the nondeterministic machine for language L_2 (right), which is identical to the one in Fig. 3.6. □

As a further motivation, we anticipate that nondeterministic machines are the intermediate product of some procedures for converting r.e. to automata, widely used for designing lexical analyzers or scanners.

3.5.2 Nondeterministic Recognizers

We precisely define the concept of nondeterministic finite-state computation, first without spontaneous moves. A *nondeterministic finite automaton* N, without spontaneous moves, is defined by:

- the state set Q
- the terminal alphabet Σ
- two subsets of Q: the set I of *initial* states and the set F of *final* states
- the transition relation δ, included in the Cartesian product $Q \times \Sigma \times Q$.

Notice that the machine may have multiple initial states. The graphic representation of the machine is as for the deterministic case.

As before, a *computation* of length n is a series of n transitions such that the origin of each transition coincides with the destination of the preceding one:

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \ \ldots \ \xrightarrow{a_n} q_n$$

In brief, the computation is also written $q_0 \xrightarrow{a_1\, a_2 \ldots a_n} q_n$. The computation *label* is the string $a_1\, a_2\, \ldots\, a_n$.

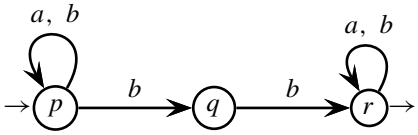
A computation is *successful* if the first state q_0 is initial and the last state q_n is final. A string x is *recognized or accepted* by the automaton, if it is the label of a successful computation.

Let us focus on the empty string. We stipulate that every state is the origin and termination of a computation of length 0, which has the empty string ε as label. It follows that the empty string is accepted by an automaton if, and only if, there exists an initial state that is also final.

The *language* L_N *recognized* by automaton N is the set of accepted strings:

$$L(N) = \left\{ x \in \Sigma^* \mid \ q \xrightarrow{x} r \text{ with } q \in I \text{ and } r \in F \right\}$$

Example 3.13 (*finding a word in a text*) Given a string or word y and a text x, does x contain y as substring? The following machine recognizes the texts that contain one or more occurrences of y, that is, the language $(a \mid b)^* \; y \; (a \mid b)^*$. We illustrate with the word $y = b\,b$:



String $a\,b\,b\,b$ is the label of several computations originating in the initial state:

$$p \xrightarrow{a} p \xrightarrow{b} p \xrightarrow{b} p \xrightarrow{b} p \qquad p \xrightarrow{a} p \xrightarrow{b} p \xrightarrow{b} p \xrightarrow{b} q \qquad \textit{unsuccessful}$$
$$p \xrightarrow{a} p \xrightarrow{b} p \xrightarrow{b} q \xrightarrow{b} r \qquad p \xrightarrow{a} p \xrightarrow{b} q \xrightarrow{b} r \xrightarrow{b} r \qquad \textit{successful}$$

The first two computations do not find the word looked for. The last two find the word, respectively, at positions $a\,b\,\underbrace{b\,b}$ and $a\,\underbrace{b\,b}\,b$. □

3.5.2.1 Transition Function

The transition relation δ of a nondeterministic automaton can still be considered a finite function, though one computing sets of values. For a machine $N = (Q, \Sigma, \delta, I, F)$, devoid of spontaneous moves, the functionality of the state-transition function δ is the following:

$$\delta : Q \times \Sigma \to \wp(Q)$$

where symbol $\wp(Q)$ indicates the powerset of set Q, i.e., the set of all the subsets of Q. Now the meaning of function $\delta(q, a) = [\, p_1, \; p_2, \; \ldots, \; p_k \,]$ is that the machine, on reading a in the current state q, can arbitrarily move to any of the states p_1, …, p_k. As we did for deterministic machines, we extend the function δ to any string y, including the empty one, as follows:

$$\forall\, q \in Q \quad \delta(q, \varepsilon) = [\, q \,]$$
$$\forall\, q \in Q \quad \forall\, y \in \Sigma^* \quad \delta(q, y) = \left[\, p \mid \; q \xrightarrow{y} p \,\right]$$

In other words, it holds $p \in \delta(q, y)$ if there exists a computation labeled y from q to p. For instance, for the automaton of Example 3.13 we have:

$$\delta(p, a) = [\, p \,] \qquad \delta(p, a\,b) = [\, p, \, q \,] \qquad \delta(p, a\,b\,b) = [\, p, \, q, \, r \,]$$

Therefore, we can reformulate the language accepted by automaton N as:

$$L(N) = \{\, x \in \Sigma^* \mid \exists\, q \in I \quad \text{such that } \delta(q,\, x) \,\cap\, F \neq \emptyset \,\}$$

i.e., the set computed by function delta must contain a final state, for a string to be recognized.

3.5.3 Automata with Spontaneous Moves

Another kind of nondeterministic behavior occurs when an automaton changes state without reading a character, thus performing a *spontaneous move*, represented by an ε-arc. Such arcs will prove to be expedient for assembling the automata that recognize a regular composition of finite-state languages. The next example illustrates the case for union and concatenation.

Example 3.14 (*compositional definition of numeric constants*) This language includes constants such as $90 \bullet 01$. The substring preceding the fractional point may be missing, as in $\bullet 01$ and it may not contain leading zeroes. Trailing zeroes are permitted at the end of the fractional part. The language is defined by the r.e.:

$$L = (\varepsilon \mid 0 \mid N) \bullet (0 \ldots 9)^+ \qquad \text{where } N = (1 \ldots 9)\,(0 \ldots 9)^*$$

The automaton in Fig. 3.8 mirrors the structure of the expression. Notice that the presence of spontaneous moves does not affect the way a machine performs recognition: a string x is *recognized* by a machine with spontaneous moves if it is the label of a path that originates in an initial state and terminates in a final state.

Observe that by taking the spontaneous move from state A to state C, the integer part N vanishes. String $34 \bullet 5$ is accepted with the computation:

$$A \xrightarrow{3} B \xrightarrow{4} B \xrightarrow{\varepsilon} C \xrightarrow{\bullet} D \xrightarrow{5} E$$

□

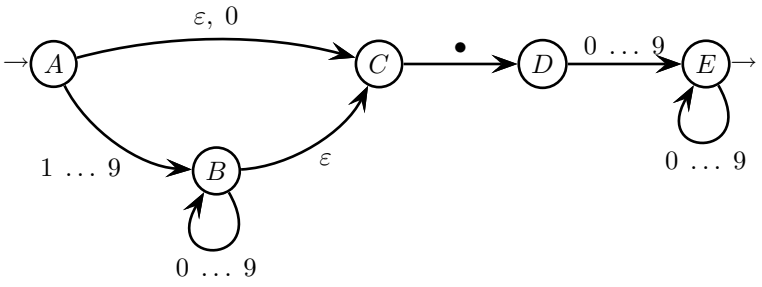


Fig. 3.8 Automaton with spontaneous moves that recognizes the numerical constants of Example 3.14

On the other hand, the number of steps (time complexity) of the computation can exceed the length of the input string, because of the presence of ε-arcs. As a consequence, the recognition algorithm no longer works in real time. Yet, the time complexity remains linear, because it is possible to assume that the machine does not perform a cycle of spontaneous moves in any computation.

The family of languages recognized by such nondeterministic automata is also called *finite-state*.

3.5.3.1 Uniqueness of the Initial State

The definition of nondeterministic machine on p. 133 allows two or more initial states. However, it is easy to construct an equivalent machine with only one: add to the machine a new state q_0, which will be the only initial state, and add the ε-arcs going from it to the former initial states of the automaton. Clearly any computation of the new automaton accepts a string if, and only if, the old automaton does so.

This transformation trades the form of nondeterminism linked with multiple initial states, with the form related to spontaneous moves. We will see on p. 142 that such moves can be eliminated as well.

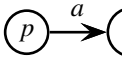
3.5.4 Correspondence Between Automata and Grammars

We collect in Table 3.1 the mapping between nondeterministic automata, also with spontaneous moves, and unilinear grammars. The correspondence is so direct as to witnesses that the two models are essentially just notational variants. Consider a right-linear grammar $G = (V, \Sigma, P, S)$ and a nondeterministic automaton $N = (Q, \Sigma, \delta, q_0, F)$, which from the preceding discussion we may assume to have a single initial state. Initially, assume the grammar rules are strictly unilinear (p. 86). The states Q match the nonterminals V. The initial state corresponds to the axiom. Notice (row 3) that the pair of alternatives $p \to a\,q \mid a\,r$ corresponds to a pair of nondeterministic moves. A copy rule (row 4) matches a spontaneous move. A final state (row 5) matches a nonterminal having an empty rule.

It is easy to see that every grammar derivation matches a machine computation, and conversely, so that the following statement ensues.

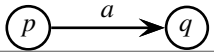
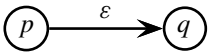
Property 3.15 (finite automaton and unilinear grammar) *A language is recognized by a finite automaton if, and only if, it is generated by a unilinear grammar.* □

Notice that the statement concerns also the left-linear grammars, since from Chap. 2 we know that they have the same generative capacity as the right-linear ones.

If a grammar contains non-empty terminal rules of type $p \to a$, with $a \in \Sigma$, the automaton is modified to include a new final state f, different from those listed at row 5 of Table 3.1, and the move 

.

Table 3.1 Correspondence between nondeterministic finite automata (*NFA*) and right-linear grammars

#	Right-linear grammar	NFA
1	Nonterminal alphabet $V = Q$	State set $Q = V$
2	Axiom $S = q_0$	Initial state $q_0 = S$
3	$p \to a\,q$ where $a \in \Sigma$ and $p,\ q \in V$	
4	$p \to q$ where $p,\ q \in V$	
5	$p \to \varepsilon$	Final state 

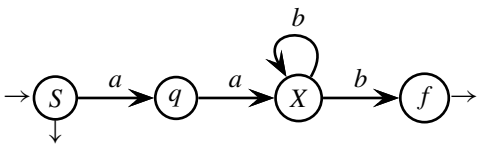
Example 3.16 (*right-linear grammar and nondet. automaton*) The grammar that matches the automaton of numeric constants (Fig. 3.8 on p. 135) is:

$$\begin{cases} A \to 0\,C \mid C \mid 1\,B \mid \ldots \mid 9\,B \\ C \to \bullet\,D \\ E \to 0\,E \mid \ldots \mid 9\,E \mid \varepsilon \end{cases} \qquad \begin{cases} B \to 0\,B \mid \ldots \mid 9\,B \mid C \\ D \to 0\,E \mid \ldots \mid 9\,E \end{cases}$$

where nonterminal A is the axiom.

Next, we drop the assumption of unilinearity in the strict sense. Observe the right-linear grammar (left) that matches the automaton (right) below:

$$\begin{cases} S \to a\,a\,X \mid \varepsilon \\ X \to b\,X \mid b \end{cases}$$

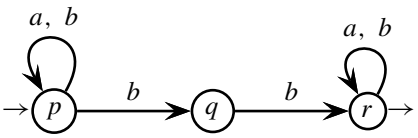


The non-strictly right-linear rule $S \to a\,a\,X$ is converted to a cascade of two arcs, with an intermediate state q after the first character a. Moreover, the new final state f mirrors the last derivation step, which uses rule $X \to b$. □

3.5.5 Ambiguity of Automata

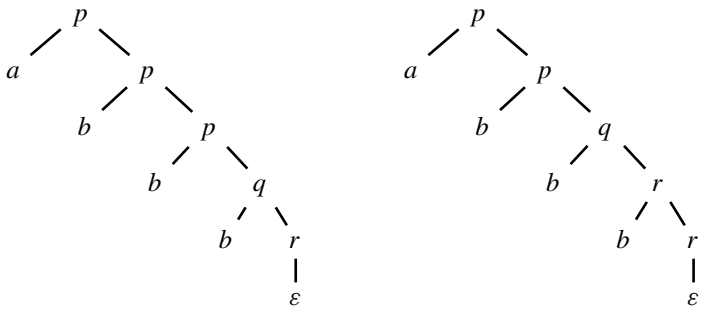
An automaton is *ambiguous* if it accepts a string with two different computations. Clearly, from the definition it follows that every deterministic automaton is not ambiguous. It is interesting to link the notions of ambiguity for automata and for unilinear grammars. From knowing that there is a one-to-one (bijective) correspondence between computations and grammar derivations, it follows that an automaton is ambiguous if, and only if, the right-linear equivalent grammar is ambiguous, i.e., if the grammar generates a sentence with two distinct syntax trees.

Example 3.17 (*ambiguity of automaton and grammar*) The automaton of Example 3.13 on p. 134, reproduced below:



$$\begin{cases} p \rightarrow a\,p \mid b\,p \mid b\,q \\ r \rightarrow a\,r \mid b\,r \mid \varepsilon \\ q \rightarrow b\,r \end{cases}$$

recognizes string $a\,b\,b\,b$ with two distinct successful paths. The equivalent grammar (right) generates the same string with two trees:



Therefore both models are ambiguous with the same degree of ambiguity. □

3.5.6 Left-Linear Grammars and Automata

We recall that family *REG* is also defined by using left-linear grammars. By interchanging left and right, it is simple to discover the mapping between such grammars and finite automata. Observe the forms of left-linear rules:

$$G: \quad A \rightarrow B\,a \qquad A \rightarrow B \qquad A \rightarrow \varepsilon$$

Consider such a grammar G (axiom A) and the reversed language $L^R = (\,L\,(G)\,)^R$. Language L^R is generated by the reversed grammar, denoted G_R, obtained (p. 95) by transforming the rules of the first form into $A \rightarrow a\,B$, while the other two forms are unchanged. Since grammar G_R is right-linear, we know how to construct a finite automaton N_R for L^R. To obtain the automaton of the original language L, we modify automaton N_R, by reversing the arc arrows and interchanging the initial and final states.

Example 3.18 (*from left-linear grammar to automaton*) Given grammar G:

$$G \begin{cases} S \rightarrow A\,a \mid A\,b \\ A \rightarrow B\,b \\ B \rightarrow B\,a \mid B\,b \mid \varepsilon \end{cases}$$

the reversed grammar G_R is:

$$G_R \begin{cases} S \rightarrow a\,A \mid b\,A \\ A \rightarrow b\,B \\ B \rightarrow a\,B \mid b\,B \mid \varepsilon \end{cases}$$

The corresponding automaton N_R that recognizes the mirror language $(L(G))^R$ and the recognizer N for language $L(G)$ are:



recognizer N_R *of* $\big(L(G)\big)^R$ *recognizer* N *of* $L(G)$

Incidentally, language $(L(G))^R$ has been met before: the penultimate letter of each sentence must be b. □

3.6 From Automaton to Regular Expression: *BMC* Method

In applications, one has sometimes to compute the r.e. for the language defined by a machine. We already know a rather indirect method: since an automaton is easily converted into an equivalent right-linear grammar, the r.e. of the language can be computed by solving linear simultaneous equations, as seen on p. 87. The next direct *elimination method*, named *BMC* after Brzozowski and McCluskey, is often more convenient.

For simplicity suppose that the initial state i is unique and no arc enters it; similarly the final state t is unique and without outgoing arcs. Otherwise, just add a new initial state i connected by spontaneous moves to the ex-initial states; similarly introduce a new unique final state t. Every state other than states i and t is called *internal*. We construct an equivalent automaton, termed *generalized*, which is more flexible as it allows the arc labels to be not just terminal characters, but also regular languages;

i.e., a label can be an r.e. The idea is to eliminate the internal states one by one, while compensating by introducing new arcs labeled with r.e., until only the initial and final states are left. Then the label of arc $i \rightarrow t$ is the r.e. of the language.

In Fig. 3.9, observe on the top an internal state q with all the adjoining arcs; on the bottom, the same machine fragment after eliminating state q and with compensatory arcs is labeled with the same strings produced when traversing state q. To avoid too many superpositions, some arcs and labels are not shown, but of course, for each state pair p_i and r_j there should be an arc $p_i \xrightarrow{H_i\, J^*\, K_j} r_j$. Notice that some states

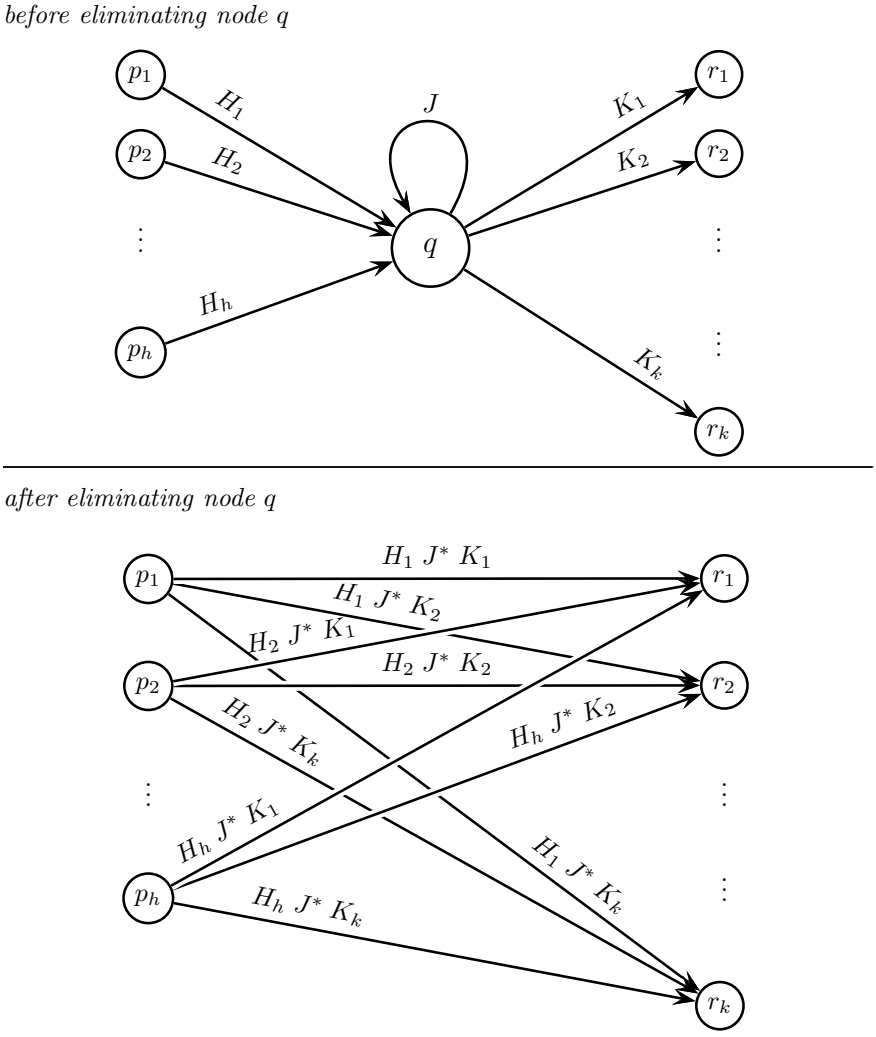


Fig. 3.9 *BMC* algorithm: deleting a node and compensating with new generalized transition arcs

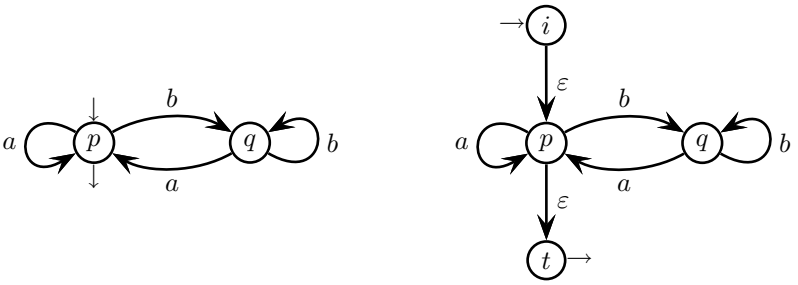


Fig. 3.10 Automaton before (left) and after (right) normalizing (Example 3.19)

p_i and r_j may coincide. Clearly, the set of the strings that may be read when the original automaton moves from state p_i to state r_j coincides with the language that labels the arc $p_i \to r_j$ of the new automaton.

In order to compute the r.e. of the language of a given automaton, the above transformation is applied over and over, every time by eliminating an internal state.

Example 3.19 (*from Sakarovitch* [12]) The automaton is shown in Fig. 3.10, before and after normalizing. In Fig. 3.11 we trace the execution of the *BMC* algorithm, by eliminating the states in the order q and p. The elimination order does not affect the result; though, as in solving simultaneous equations by elimination, it may yield more or less complex yet equivalent solutions. For instance, the order p and q would produce the r.e. $(a^* b)^+ a^+ \mid a^*$. □

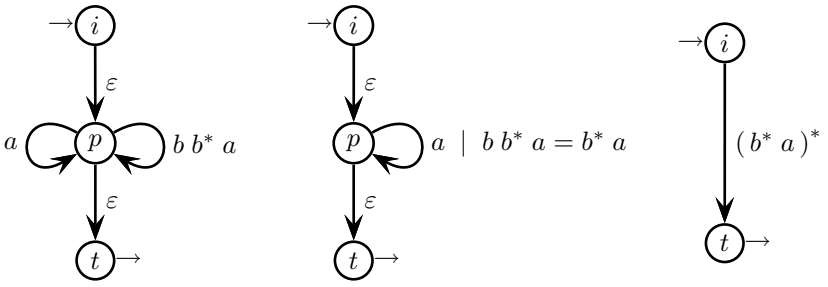


Fig. 3.11 From left to right: the automaton of Fig. 3.10 after eliminating node q, unifying the self-loops, simplifying the r.e. and eliminating node p

3.7 Elimination of Nondeterminism

We have argued for the use of nondeterministic machines in the language specifications and transformations, but the final stage of a project usually requires an efficient implementation, which can only be provided by a deterministic machine. There are rare exceptions, like when the cost to be minimized is the time needed to construct the automaton, instead of the time spent in recognizing strings. This happens for instance in a text editor, when a search algorithm is implemented through an automaton to be used only one time, to find a given string.

Next, we describe an algorithm for turning a nondeterministic automaton into an equivalent deterministic one; as a corollary every unilinear grammar can be made non-ambiguous. The algorithm consists of two sequential phases:

1. Elimination of spontaneous moves, thus obtaining a machine that in general is nondeterministic. We recall that, if the machine has multiple initial states, an equivalent machine can be obtained with only one initial state, which is connected to the former initial states via spontaneous moves.
2. Replacement of several nondeterministic transitions with one transition that enters a new state. This phase is called *powerset construction*, because the new states correspond to the subsets of the state set.

We anticipate that a different method that combines the two phases into one is Algorithm 3.43 on p. 163.

3.7.1 Elimination of Spontaneous Moves

Since spontaneous moves match the copy rules of the equivalent right-linear grammar, one way to eliminate such moves is to use the grammar transformation that removes copy rules (p. 67). But first we show how to directly eliminate the ε-arcs on the graph of the automaton.

For the automaton graph, we call ε-*path* a path made only by ε-arcs. The next algorithm initially enriches the automaton with a new ε-arc between any two states that are connected by an ε-path. Then, for any length-two path made by an ε-arc followed by a non-ε-arc, that we may also call a *scanning* move, the algorithm adds the arc that scans the same terminal character. At last, any state linked to a final state by an ε-arc is made final, and all the spontaneous arcs are deleted, as well as all the useless states.

Algorithm 3.20 (*direct elimination of spontaneous moves*) Let δ be the original state-transition graph, and let $F \subseteq Q$ be the set of final states.

Input: a finite-state automaton with ε-moves
Output: an equivalent finite-state automaton without ε-moves

1 Transitive closure of the ε-paths:
repeat
  if graph δ contains a path $p \xrightarrow{\varepsilon} q \xrightarrow{\varepsilon} r$ (with distinct p, q and r) **then**
    add the arc $p \xrightarrow{\varepsilon} r$ to graph δ
until no more arcs have been added in the last iteration

2 Backward propagation of the scanning moves over the ε-moves:
repeat
  if graph δ contains a path $p \xrightarrow{\varepsilon} q \xrightarrow{b} r$ (with distinct p and q) **then**
    add the arc $p \xrightarrow{b} r$ to graph δ
until no more arcs have been added in the last iteration

3 New final states: $F := F \cup \{\, q \mid$ the ε-arc $q \xrightarrow{\varepsilon} f$ is in δ and $f \in F \,\}$

4 Clean-up: delete all the ε-arcs then and all the states that are not accessible from the initial state □

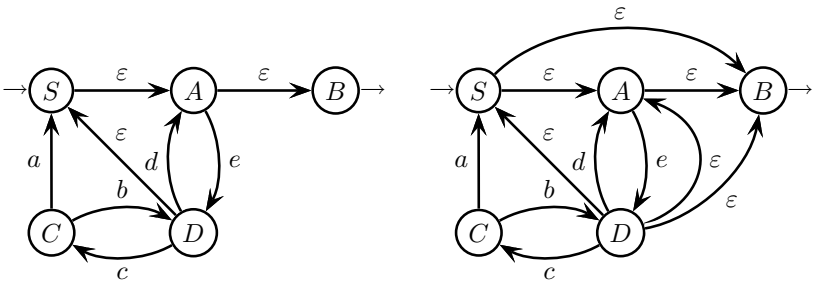
Example 3.21 (*ε-move elimination by Algorithm 3.20*) Consider the automaton in Fig. 3.12, top left. On the top right it is shown the graph after step 1; in the middle after steps 2, 3 and the part of 4 that deletes ε-arcs; then, since state B becomes inaccessible it is deleted.

As said, an alternative way to solve the problem transforms the automaton into a right-linear grammar, from which the copy rules are then eliminated as explained on p. 67.[6] The bottom part of Fig. 3.12 shows the details. □

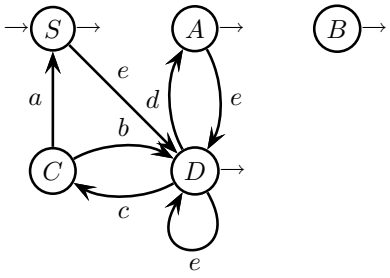
After this machine transformation, if the result is not deterministic yet, the next transformation must be applied.

[6] Although the copy-rule elimination algorithm on p. 67 does not handle empty rules, for right-linear grammars it continues to work also when empty rules are in the grammar.

automaton with ε-arcs *after the transitive closure of ε-paths*



after back-propagation, final state creation, and deletion of ε-arcs



original left-linear grammar

$$\begin{cases} S \to A \\ C \to a\,S \mid b\,D \end{cases} \qquad \begin{cases} A \to B \mid e\,D \\ D \to S \mid c\,C \mid d\,A \end{cases} \qquad B \to \varepsilon$$

after eliminating copy rules and a rule unreachable from axiom

$$\begin{cases} S \to \varepsilon \mid e\,D \\ C \to a\,S \mid b\,D \end{cases} \qquad \begin{cases} A \to \varepsilon \mid e\,D \\ D \to \varepsilon \mid e\,D \mid c\,C \mid d\,A \end{cases} \qquad \cancel{B \to \varepsilon}$$

Fig. 3.12 Automaton of Example 3.21 (top left). The automaton after step 1 of Algorithm 3.20 (top right). After steps 2, 3 and the first part of step 4, with state B nonaccessible (middle). The equivalent grammar and the version after elimination of copy rules (bottom)

3.7.2 Construction of Accessible Subsets

Given a nondeterministic automaton N without spontaneous moves, we explain how to construct an equivalent deterministic machine, denoted by M'. We start by observing that, if machine N contains the following nondeterministic moves:

$$p \xrightarrow{a} p_1 \qquad p \xrightarrow{a} p_2 \qquad \dots \qquad p \xrightarrow{a} p_k$$

then, after reading letter a, it can be in any of the next states p_1, p_2, …, p_k, i.e., in a state of uncertainty. Therefore, to simulate this behavior, in machine M' we create a new state, that we may qualify as *collective*, and we name it by means of the set of states:

$$[\, p_1,\ p_2,\ \ldots,\ p_k \,]$$

To connect the new state to the others, we proceed to construct the outgoing arcs according to the following rule. If the collective state contains the states p_1, p_2, …, p_k, for each of them we consider, in the machine N, the outgoing arcs are labeled with the same letter a:

$$p_1 \xrightarrow{a} [\, q_1,\ q_2,\ \ldots \,] \qquad p_2 \xrightarrow{a} [\, r_1,\ r_2,\ \ldots \,] \qquad \text{etc.}$$

and we merge together the next states:

$$[\, q_1,\ q_2,\ \ldots \,] \cup [\, r_1,\ r_2,\ \ldots \,] \cup \ldots$$

thus obtaining the collective state reached by the transition:

$$[\, p_1,\ p_2,\ \ldots,\ p_k \,] \xrightarrow{a} [\, q_1,\ q_2,\ \ldots,\ r_1,\ r_2,\ \ldots,\ \ldots \,]$$

If such a state does not already exist, it is added to the current state set of M'. The next Algorithm 3.22 formalizes such a construction.[7]

Given a nondeterministic automaton $N = (Q, \Sigma, \delta, q_0, F)$ having one initial state[8] and without spontaneous moves, the algorithm constructs an equivalent deterministic automaton M'.

Algorithm 3.22 (*powerset construction*) Machine $M' = \left(Q',\ \Sigma,\ \delta',\ [q_0],\ F' \right)$ is defined by the following components:

1. The state set Q' is the powerset of Q, i.e., $Q' = \wp(Q)$
2. Initial state: $[\, q_0 \,]$
3. Final states: $F' = \left\{\ p' \in Q' \mid\ \ p' \cap F \neq \emptyset \ \right\}$, i.e., the states that contain a final state of N
4. The transition function δ' is as follows, for all the states $p' \in Q'$ and for all the characters $a \in \Sigma$:

$$p' \xrightarrow{a} \left[\, s \in Q \mid\ \ q \in p' \ \wedge\ \text{arc}\left(q \xrightarrow{a} s\right) \text{is in } N \,\right]$$

□

[7] If an *NFA* is represented by the equivalent left-linear grammar, the powerset algorithm above performs exactly as the construction for transforming a context-free grammar into invertible form by eliminating the repeated right parts of rules, which was presented in Chap. 2 on p. 70.

[8] If the given automaton N has multiple initial states, the initial state of machine M' is the set of all of them.

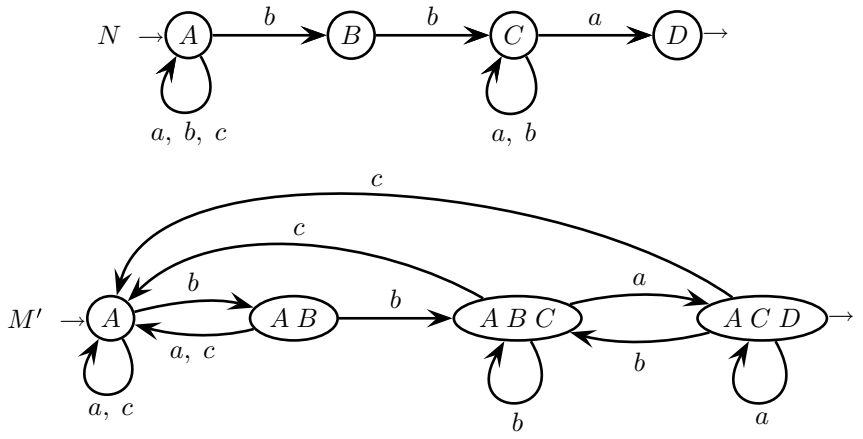


Fig. 3.13 From nondeterministic machine N (top) to deterministic machine M' (bottom)

Notice that at line 4, if an arc $q \xrightarrow{a} q_{err}$ leads to the error state, it is not added to the collective state. In fact, a computation entering the sink never recognizes any string and can be ignored.

Since the states of machine M' are the subsets of Q, in the worst case the cardinality of Q' is exponentially larger than that of Q. This confirms the previous findings that deterministic machines may be larger; remember the exponential explosion of the number of states in the language that has a specific character in the k-th position before the end (p. 131).

Algorithm 3.22 can be improved: machine M' often contains states inaccessible from the initial state, hence useless. Instead of erasing them with the clean-up procedure, it is better to altogether avoid their creation: we draw only the collective states that can be reached from the initial state.

Example 3.23 (*determinization by powerset construction*) The nondeterministic automaton N in Fig. 3.13 (top) is transformed to the deterministic one M' (bottom). From $\delta(A, b) = \{ A, B \}$ we draw the collective state $[A, B]$, and from this the transitions:

$$[A, B] \xrightarrow{a} \big(\delta(A, a) \cup \delta(B, a) \big) = [A]$$

$$[A, B] \xrightarrow{b} \big(\delta(A, b) \cup \delta(B, b) \big) = [A, B, C]$$

Then, we create the transitions from the new collective state $[A, B, C]$:

$$[A, B, C] \xrightarrow{a} \big(\delta(A, a) \cup \delta(B, a) \cup \delta(C, a) \big) = [A, C, D] \text{ final state}$$

$$[A, B, C] \xrightarrow{b} \big(\delta(A, b) \cup \delta(B, b) \cup \delta(C, b) \big) = [A, B, C] \text{ self-loop}$$

etc. The algorithm ends when step 4, applied to the current state set Q', does not generate any new state. Notice that not all the subsets of Q correspond to an accessible state; e.g., the subset and state $[\, A,\ C\,]$ would be useless. □

To justify the algorithm correctness, we show that a string x is recognized by machine M' if, and only if, it is accepted by machine N. If a computation of N accepts string x, there exists a path labeled with x from the initial state q_0 to a final state q_f. The algorithm ensures then that in M' there exists a labeled path x from $[\, q_0\,]$ to a state $[\, \ldots,\ q_f,\ \ldots\,]$ containing q_f.

Conversely, if string x is the label of a successful computation of M', from the initial state $[\, q_0\,]$ to a final state $p' \in F'$, then by definition state p' contains at least one final state q_f of N. By construction, there exists in N a path labeled with x from q_0 to q_f. We summarize with a fundamental statement.

Property 3.24 (recognition power of deterministic automata) *Every finite-state language can be recognized by a deterministic finite automaton.* □

This property ensures that the recognition algorithm of finite-state languages works in real time. This means that it completes the job within a number of transitions equal to the length of the input string (or fewer if an error occurs before the string has been entirely scanned).

As a corollary of Property 3.24, for every language recognized by a finite automaton, there exists an unambiguous unilinear grammar, which is the one (p. 129) naturally corresponding to the deterministic automaton. This also says that, for any finite-state language, we have a procedure to eliminate ambiguity from the grammar; that is, finite-state languages cannot be inherently ambiguous (p. 62).

3.8 From Regular Expression to Recognizer

When a language is specified with a regular expression, instead of a unilinear grammar, we do not yet know how to construct its automaton. Since this requirement is quite common in applications such as compilation, document processing and text searching, several methods have been invented. They differ with respect to the automaton being deterministic or not, with or without spontaneous moves, as well as regarding the algorithmic complexity of the construction.

We describe two construction methods. The first, due to Thompson, is termed *structural* or *modular*, as it analyzes the expression into smaller and smaller subexpressions until the atomic ones. Then the subexpression recognizers are constructed and interconnected into a graph that implements the language operations (union, concatenation and star) present in the expression. In general, the result is nondeterministic with spontaneous moves.

The second method, named after Berry and Sethi, builds a deterministic machine, typically non-minimal in the number of states. The Thompson method, however,

can be also combined with the powerset Algorithm 3.22, to the effect of directly producing a deterministic automaton.

At the end of this section, we will be able to transform language specifications back and forth from automata, grammars and regular expressions. This proves that the three models are equivalent.

3.8.1 Thompson Structural Method

Given an r.e., we analyze it into simple parts, we produce the corresponding component automata, and we interconnect them to obtain the complete recognizer. In this construction, each component machine is assumed to have exactly one initial state without incoming arcs and one final state without outgoing arcs. If not so, simply introduce two new states, as done for the *BMC* algorithm on p. 139. The Thompson algorithm[9] incorporates the mapping rule between simple r.e. and automata schematized in Table 3.2. The machines in Table 3.2 have many nondeterministic bifurcations, with outgoing ε-arcs. The validity of the Thompson method comes from it being an operational reformulation of the closure properties of regular languages under concatenation, union and star (stated in Chap. 2 on p. 25).

Example 3.25 (*from r.e. to automaton by the Thompson method*) Decompose the r.e. $(a \cup \varepsilon)\, b^*$ into subexpressions, identified by the names $E_{i,j}$:

$$\underset{0}{\Big(}\ \underset{1}{\big(}\ \underset{2}{(}a\underset{3}{)}\ \cup\ \underset{4}{(}\varepsilon\underset{5}{)}\ \underset{6}{\big)}\ \cdot\ \underset{7}{\big(}\ \underset{8}{(}b\underset{9}{)}\ \underset{10}{\big)^*}\ \underset{11}{\Big)}$$

$$\underbrace{\underbrace{\Big(\ \underbrace{a}_{E_{2,3}}\ \cup\ \underbrace{\varepsilon}_{E_{4,5}}\ \Big)}_{E_{1,6}}\ \cdot\ \underbrace{\underbrace{b}_{E_{8,9}}\ ^*}_{E_{7,10}}}_{E_{0,11}}$$

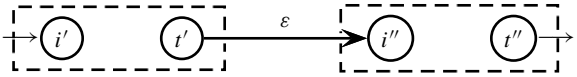
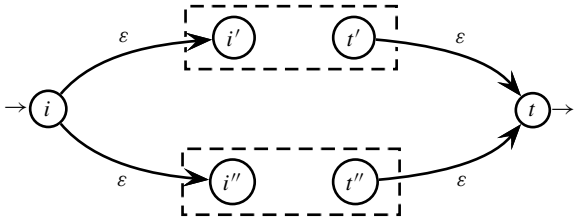
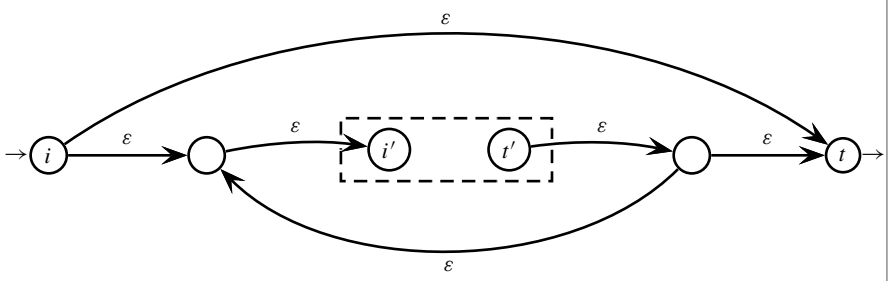
Then, apply the mapping to each subexpression, thus producing the automaton of Fig. 3.14. Notice that we have moderately simplified the construction, to avoid a proliferation of states. Of course, several states are redundant and could be coalesced. □

Some existing tools use improved versions of the algorithm to avoid constructing redundant states. Other tool versions combine the algorithm with the one for the elimination of spontaneous moves. It is interesting to look at the Thompson algorithm as a converter from the notation of an r.e. to the state-transition graph of a machine. From this standpoint, this is a typical case of syntax-directed analysis and translation, where the syntax is the one of the r.e. Such an approach to translator design is presented in Chap. 5.

[9] Originally presented in [14]. It forms the base of the popular tool *lex* (or *GNU flex*) used for building scanners.

Table 3.2 From component subexpression to component subautomaton. A rectangle depicts a component subautomaton with its unique initial (left) and final (right) states

Atomic expressions

Concatenation of expressions

Union of expressions

Star closure of an expression


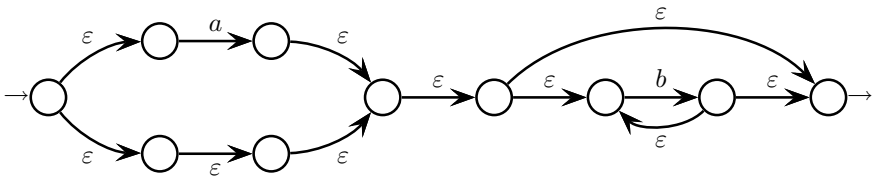


Fig. 3.14 Machine derived from r.e. $(a \cup \varepsilon)\, b^*$ (Example 3.25) through the structural (Thompson) method

3.8.2 Berry–Sethi Method

Another classical method, due to Berry and Sethi [15], derives a *deterministic* automaton that recognizes the language of a given regular expression. To justify this construction, we preliminary introduce the local languages,[10] a simple subfamily of regular languages, and the related notions of local automaton and linear regular expression.

3.8.2.1 Locally Testable Language and Local Automaton

Some regular languages are extremely simple to recognize, as it suffices to test if certain short substrings are present. An example is the set of the strings that start with b, end with a or b, and contain only $b\,a$ or $a\,b$ as substrings.

Definition 3.26 (*local sets*) For a language L over an alphabet Σ, the *set of initials* is:

$$Ini\,(L) = \left\{\, a \in \Sigma \mid \;\; a\;\Sigma^* \;\cap\; L \neq \emptyset \,\right\}$$

i.e., the starting characters of the sentences. The *set of finals* is:

$$Fin\,(L) = \left\{\, a \in \Sigma \mid \;\; \Sigma^*\,a \;\cap\; L \neq \emptyset \,\right\}$$

i.e., the ending characters of the sentences. The *set of digrams* (factors) is:

$$Dig\,(L) = \left\{\, x \in \Sigma^2 \mid \;\; \Sigma^*\,x\;\Sigma^* \;\cap\; L \neq \emptyset \,\right\}$$

i.e., the substrings of length two present in the sentences. The three sets above are called *local*. The complementary digrams are:

$$\overline{Dig\,(L)} = \Sigma^2 \setminus Dig\,(L)$$

□

Example 3.27 (*local language*) The local sets for language $L_1 = (\,a\,b\,c\,)^*$ are:

$$Ini\,(L_1) = \{\, a \,\} \qquad Fin\,(L_1) = \{\, c \,\} \qquad Dig\,(L_1) = \{\, a\,b,\;\; b\,c,\;\; c\,a \,\}$$

and the complement of the digram set is:

$$\overline{Dig\,(L_1)} = \{\, a\,a,\;\; a\,c,\;\; b\,a,\;\; b\,b,\;\; c\,b,\;\; c\,c \,\}$$

□

[10] We follow the conceptual path of Berstel and Pin [16].

We observe that the non-empty sentences of this language are precisely defined by the three local sets, in the sense of the following identity:

$$L_1 \setminus \{ \varepsilon \} = \left\{ x \;\middle|\; \begin{array}{l} Ini\,(x) \in \{ a \} \;\wedge\; Fin\,(x) \in \{ c \} \\ \wedge\; Dig\,(x) \subseteq \{ a\,b,\; b\,c,\; c\,a \} \end{array} \right\}$$

We generalize the example to the next definition.

Definition 3.28 (*local language*) A language L is called *local* (or *locally testable*) if it satisfies the following identity:

$$L \setminus \{ \varepsilon \} = \left\{ x \;\middle|\; \begin{array}{l} Ini\,(x) \in Ini\,(L) \;\wedge\; Fin\,(x) \in Fin\,(L) \\ \wedge\; Dig\,(x) \subseteq Dig\,(L) \end{array} \right\} \tag{3.1}$$

In other words, the non-empty phrases of language L are defined precisely by sets *Ini*, *Fin* and *Dig*. □

Notice that Definition 3.28 only refers to the *non-empty* strings; hence it does not say anything about the inclusion of string ε in language L. For instance, language $L_2 = (\,a\,b\,c\,)^+ = L_1 \setminus \{ \varepsilon \}$ is also local.

Clearly, not all the languages are local, but it should be clear that every language L satisfies a modified condition (3.1), where the equal sign "=" is replaced by inclusion "$\subset$". In fact, by definition every sentence starts (resp. ends) with a character of $Ini\,(L)$ (resp. $Fin\,(L)$) and its digrams are included in $Dig\,(L)$. But such conditions may be also satisfied by other strings that do not belong to the language. When this happens, the language is not local, as it does not include all the strings that can be generated from *Ini*, *Fin* and *Dig*.

Example 3.29 (*non-local language*) For language $L_2 = b\;(\,a\,a\,)^+\;b$, we have:

$$Ini\,(L_2) = Fin\,(L_2) = \{ b \}$$

$$Dig\,(L_2) = \{ a\,a,\; a\,b,\; b\,a \} \qquad \overline{Dig\,(L_2)} = \{ b\,b \}$$

The length of every sentence of language L_2 is even. On the other hand, among the strings that start and end with letter b and that do not contain the digram $b\,b$, some exists of odd length, such as $b\,a\,a\,a\,b$, and also even-length strings such as $b\,a\,b\,a\,a\,b$. Thus, the language defined by condition (3.1) strictly includes language L_2, which therefore is not local. □

Automata Recognizing Local Languages

Our present interest[11] for local languages comes from the notable simplicity of their recognizers. To recognize a string, the machine scans it from left to right, checks that the initial character is in *Ini*, verifies that any pairs of adjacent characters are in *Dig*, and finally checks that the last character is in *Fin*. Such operations can be easily performed by a finite automaton. An automaton recognizing a local language $L \subseteq \Sigma^*$, specified by the local sets *Ini*, *Fin* and *Dig* of L, is speedily obtained, as next specified.

Algorithm 3.30 (*from local sets to recognizer*) The recognizer of the local language specified by the three local sets is constructed as follows.

- The state set is $\{ q_0 \} \cup \Sigma$, meaning that each non-initial state is identified by a terminal character.
- The final state set includes set *Fin* and, if the empty string ε is in the language, also the state q_0; no other state is final.
- The transitions are $q_0 \xrightarrow{a} a$ if $a \in Ini$, and $a \xrightarrow{b} b$ if $a\,b \in Dig$. □

Such an automaton is in the state identified by letter b if, and only if, the character most recently read is b. We may think that the machine has a sliding window with a width of two characters, which triggers the transition from the previous state to the current one if the digram is listed in the set *Dig*. The automaton produced by this construction is deterministic and has the following property that characterizes the family of *local automata*.

Definition 3.31 (*local automaton*) A deterministic automaton $A = (Q, \Sigma, \delta, q_0, F)$ is called *local* if it satisfies the condition:

$$\forall\, a \in \Sigma \quad |\{ \delta(q, a) \mid \quad q \in Q \}| \leq 1 \tag{3.2}$$

which forbids that two identically labeled arcs enter distinct states. □

We check that the recognizer computed by Algorithm 3.30 satisfies condition (3.2): in fact, each non-initial state is identified by a letter $a \in \Sigma$ and all the arcs labeled with a enter the state identified by a. Therefore, such an automaton is local according to Definition 3.31, and in addition it has two special properties:

1. The state set is $Q = \{ q_0 \} \cup \Sigma$. Therefore, the number of states is $|\Sigma| + 1$.
2. All and only the arcs labeled with the character $a \in \Sigma$ enter state a. Consequently, no arc enters the initial state q_0.

[11] In Chap. 5, local languages and automata are also used to model the control-flow graph of a program.

If a local automaton additionally satisfies the two conditions 1 and 2 above, it is called *normalized*.

We show that any local automaton A admits an equivalent normalized machine A'. Consider a machine A satisfying condition (3.2), but not condition 2 above. Then, automaton A contains two (or more) distinctly labeled arcs that enter the same state q, i.e., $p \xrightarrow{a} q \xleftarrow{b} r$, where $a \neq b$. To obtain the normalized machine A', we split state q into two states, say q_a and q_b, with the new arcs $p \xrightarrow{a} q_a$ and $q_b \xleftarrow{b} r$. Consequently, every arc $q \xrightarrow{c} s$ outgoing from q is replaced by the arcs $q_a \xrightarrow{c} s$ and $q_b \xrightarrow{c} s$. By repeating this transformation for all the states of machine A, we obtain a machine that meets condition 2, since for all the letters $a \in \Sigma$, all the arcs $\ldots \xrightarrow{a} \ldots$ enter state q_a. But such a machine may still have arcs that enter the initial state and violate condition 1: for instance, the pattern $p_c \xrightarrow{a} q_{0_a} \xrightarrow{b} r_b$. To comply with condition 1, we simply add the new initial state q'_0 and the arc $q'_0 \xrightarrow{b} r_b$. Since all the preceding machine transformations preserve the language recognized, the machine A' thus obtained is equivalent to A and is normalized local. Clearly, in general a normalized local machine has more states than the minimal equivalent local machine, since in the latter two or more states $q_a, q_b, \ldots$ may be coalesced into a single state q.

The final point we want to make[12] is that local automata (normalized or non-normalized) and local sets define the same language family.

Property 3.32 (local languages and local automata) *For any language L, the three conditions below are equivalent:*

1. *Language L is local (Definition 3.28).*
2. *Language L is recognized by a local automaton (Definition 3.31).*
3. *Language L is recognized by a normalized local automaton (Definition* 3.31 *and conditions 1 and 2 on p. 152)* □.

Example 3.33 (two local automata) We examine two languages. First, for the r.e. $(\,a\,b\,c\,)^*$ of Example 3.27, the normalized local automaton produced by Algorithm 3.30 is shown in Fig. 3.15 (left).

Second, for the r.e. $a\ (\,b\ \mid\ c\,)^*$, the minimal local automaton is in Fig. 3.15 (right). The larger normalized automaton is in Fig. 3.17 on p. 158. □

Composition of Local Languages Over Disjoint Alphabets

Before applying the sliding window idea to a generic regular expression, we need another conceptual step, based on the following observation: the basic operations, when applied to local languages, preserve the locality property, provided that the terminal alphabets of the languages to be combined are disjoint.

[12] We refer to [16] for a formal proof.

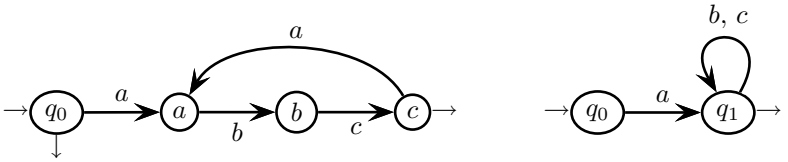


Fig. 3.15 Two local automata (Example 3.33). Left: normalized automaton for language $(a\,b\,c)^*$ (Example 3.27). Right: non-normalized automaton for language $a\,(b \mid c)^*$, see the normalized automaton on p. 158

Property 3.34 (local language closure) *Given two local languages $L' \subseteq \Sigma'^*$ and $L'' \subseteq \Sigma''^*$ over disjoint alphabets, i.e., $\Sigma' \cap \Sigma'' = \emptyset$, the languages obtained by union $L' \cup L''$, concatenation $L' \cdot L''$ and star L'^* (and cross L'^+ too) are local.* □

Proof Given the normalized local automata of L' and L'', which we also call L' and L'' with a slight abuse, the next simple Algorithm 3.35 constructs a normalized local automaton for the resulting language by combining the component machines. □

Let q_0' and q_0'' be the initial states, and let F' and F'' be the final state sets, of machines L' and L''. In general, the combined recognizer has the same states and transitions as L' and L'' have, with some adjustments on the initial and final states and on the related arcs. □

Algorithm 3.35 (*composition of normalized local automata over disjoint alphabets*) We separately consider the three operations.

For the union $L' \cup L''$:

- The new initial state q_0 is the merging of the initial states q_0' and q_0''.
- The arcs are those of L' plus L''.
- The set of final states is $F' \cup F''$ if $\varepsilon \notin L' \cup L''$; otherwise, it is $F' \cup F'' \cup \{q_0\}$

For the concatenation $L' \cdot L''$:

- The initial state is q_0'
- The arcs are those of L', plus those of L'', except the ones exiting from the initial state q_0'', such as $q_0'' \xrightarrow{a} q''$.
- For each omitted arc $q_0'' \xrightarrow{a} q''$ of L'', and for every final state $q' \in F'$, add the arc $q' \xrightarrow{a} q''$.
- The set of final states is F'' if $\varepsilon \notin L''$; otherwise, it is $F' \cup F''$.

For the star L'^*:

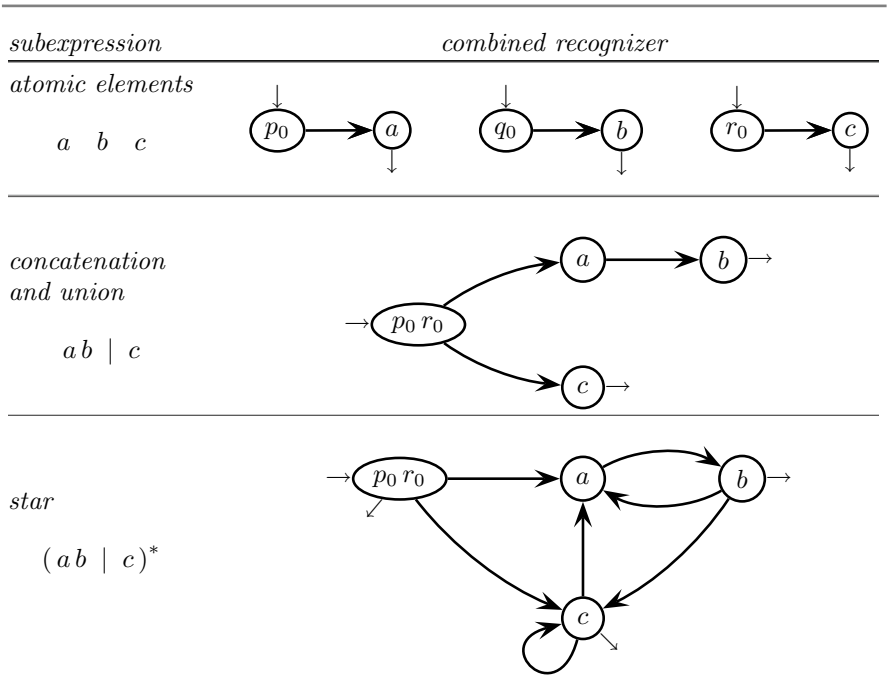


Fig. 3.16 Stepwise composition of normalized local automata for the linear r.e. $(a\,b \mid c)^*$ (Example 3.36)

- The initial state is q'_0
- The arcs are those of L', plus the following: for every final state $q \in F'$, for every arc $q'_0 \xrightarrow{a} r$ in L' (exiting from the initial state), add the arc $q \xrightarrow{a} r$.
- The final state set is $F' \cup \{q'_0\}$. □

We easily see that the recognizer produced by Algorithm 3.35 is by construction a normalized local automaton and correctly recognizes the union, concatenation or star of local languages.

Example 3.36 (*composition of local automata*) In Fig. 3.16 we apply Algorithm 3.35 to r.e. $(a\,b \mid c)^*$. The recognizer is constructed, starting from the atomic subexpressions (characters a, b and c), through the steps:

concatenate characters a and b
unite subexpression $a\,b$ to character c
apply star to subexpression $a\,b \mid c$ □

3.8.2.2 Linear Regular Expression and Its Local Sets

In a generic r.e., of course a terminal character may be repeated. An r.e. is called *linear* if no terminal character is repeated. For instance, r.e. $(a\,b\,c)^*$ is linear, whereas r.e. $(a\,b)^*\,a$ is not.

Property 3.37 (linear r.e.) *The language defined by a linear r.e. is local.* □

Proof Consider any two non-overlapping subexpressions that occur in the linear r.e. Due to linearity, such subexpressions necessarily have disjoint alphabets. Since the whole r.e. is obtained by composing subexpressions, from Property 3.34 on p. 154 it follows that the language is local. □

Notice that the linearity of an r.e. is a sufficient condition for the language to be local, but it is not necessary. For instance, the r.e. $(a\,b)^*\,a$, though it is not linear, defines a local language.

Now we know that the language of a linear r.e. is local, and the problem of constructing the recognizer melts down to computing the sets *Ini*, *Fin* and *Dig* of the language. We next explain how to orderly perform the job.

Computing the Local Sets of a Regular Language

In Table 3.3 we list the rules for computing the three local sets of a regular expression. Such rules apply to any r.e., but in fact we just use linear expressions. First, we must check if the r.e. e is *nullable*, i.e., $\varepsilon \in L(e)$. To this end, we inductively define the predicate $Null(e)$ on the structure of e by means of the rules in Table 3.3 (top part). To illustrate, we have:

$$\begin{aligned} Null\left((a \mid b)^*\,b\,a\right) &= Null\left((a \mid b)^*\right) \wedge Null(b\,a) \\ &= true \wedge (Null(b) \wedge Null(a)) \\ &= true \wedge (false \wedge false) = false \end{aligned}$$

Given a linear r.e., we compute the local sets, and then we apply Algorithm 3.30 (p. 152), which returns the local automaton that recognizes the language.

Example 3.38 (*recognizer of a linear r.e.*) For the r.e. $a\,(b \mid c)^*$, the local sets are:

$$Ini = \{\,a\,\} \qquad Fin = \{\,b,\ c\,\} \cup \{\,a\,\} = \{\,a,\ b,\ c\,\}$$

$$Dig = \{\,a\,b,\ a\,c\,\} \cup \{\,b\,b,\ b\,c,\ c\,b,\ c\,c\,\} = \{\,a\,b,\ a\,c,\ b\,b,\ b\,c,\ c\,b,\ c\,c\,\}$$

By Algorithm 3.30, we produce the normalized local automaton in Fig. 3.17. □

Numbered Regular Expression

Given any r.e. e over an alphabet Σ, we next specify how to obtain another regular expression, denoted by e', that is linear. We make distinct all the terminals that occur

Table 3.3 Rules for computing predicate *Null* and the local sets *Ini*, *Fin* and *Dig*

Nullability predicate
$Null\,(\emptyset) = false$
$Null\,(\varepsilon) = true$
$Null\,(a) = False$ for every terminal a
$Null\,(e \cup e') = Null\,(e) \vee Null\,(e')$
$Null\,(e \cdot e') = Null\,(e) \wedge Null\,(e')$
$Null\,(e^*) = true$
$Null\,(e^+) = Null\,(e)$
Set of initials
$Ini\,(\emptyset) = \emptyset$
$Ini\,(\varepsilon) = \emptyset$
$Ini\,(a) = \{\, a \,\}$ for every terminal a
$Ini\,(e \cup e') = Ini\,(e) \cup Ini\,(e')$
$Ini\,(e \cdot e') =$ **if** $Null\,(e)$ **then** $Ini\,(e) \cup Ini\,(e')$ **else** $Ini\,(e)$ **end if**
$Ini\,(e^*) = Ini\,(e^+) = Ini\,(e)$
Set of finals
$Fin\,(\emptyset) = \emptyset$
$Fin\,(\varepsilon) = \emptyset$
$Fin\,(a) = \{\, a \,\}$ for every terminal a
$Fin\,(e \cup e') = Fin\,(e) \cup Fin\,(e')$
$Fin\,(e \cdot e') =$ **if** $Null\,(e')$ **then** $Fin\,(e) \cup Fin\,(e')$ **else** $Fin\,(e')$ **end if**
$Fin\,(e^*) = Fin\,(e^+) = Fin\,(e)$
Set of digrams
$Dig\,(\emptyset) = \emptyset$
$Dig\,(\varepsilon) = \emptyset$
$Dig\,(a) = \emptyset$ for every terminal a
$Dig\,(e \cup e') = Dig\,(e) \cup Dig\,(e')$
$Dig\,(e \cdot e') = Dig\,(e) \cup Dig\,(e') \cup Fin\,(e) \cdot Ini\,(e')$
$Dig\,(e^*) = Dig\,(e^+) = Dig\,(e) \cup Fin\,(e) \cdot Ini\,(e)$

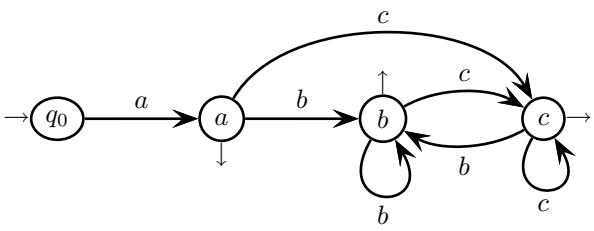


Fig. 3.17 Normalized local automaton of Example 3.38, which recognizes the language defined by the linear r.e. $a\ (b\ \mid\ c)^*$

in e by numbering each of them, say from left to right, with an integer subscript. The alphabet of e' is denoted Σ_N and is called the *numbered input alphabet*. For instance, the r.e. $e = (a\,b)^*\ a$ becomes $e' = (a_1\,b_2)^*\ a_3$ and the numbered alphabet is $\Sigma_N = \{\ a_1,\ b_2,\ a_3\ \}$.[13] Notice that the choice for numbering is immaterial, provided that all the subscripts are distinct.

3.8.2.3 From Generic Regular Expression to Deterministic Recognizer

We are ready to present an algorithm for obtaining a deterministic recognizer for any regular expression, through a sequence of steps based on the previous results for local languages and linear regular expressions. The next Algorithm 3.39 will be optimized into the Berry–Sethi algorithm on p. 161.

Algorithm 3.39 (*from r.e. to deterministic automaton*)

1. From the original r.e. e over alphabet Σ, derive the numbered linear r.e. $e'\ \dashv$ over alphabet $\Sigma_N\ \cup\ \{\ \dashv\ \}$ where $\dashv$ is the terminator, with $\dashv\ \notin\ \Sigma$.
2. Build the normalized local automaton that recognizes the (local) language $L\,(e'\ \dashv)$. The automaton has three state types: (*i*) an initial state q_0, (*ii*) for each element $c\ \in\ \Sigma_N$ a state neither initial nor final and (*iii*) a unique final state $\dashv$.
3. Tag each state with the set of all the symbols that label the arcs outgoing from that state:

   - The initial state q_0 is tagged with set $Ini\,(e'\ \dashv)$.
   - The final state $\dashv$ is tagged with the empty set $\emptyset$.
   - For each state c neither initial nor final, the set is called *set of followers* of c in the r.e. $e'\ \dashv$; such a set is denoted as $Fol\,(c)$ and is a subset of $\Sigma_N\ \cup\ \{\ \dashv\ \}$; set *Fol* is computed from the digrams, in this way:

   $$Fol\,(a_i) = \left\{\ b_j\ \mid\ \ a_i\,b_j\ \in\ Dig\,(e'\ \dashv)\ \right\} \qquad a_i \text{ and } b_j \text{ may coincide}$$

   the sets $Fol\,(c)$ altogether are equivalent to the local set *Dig* and, with the other two local sets *Ini* and *Fin*, they characterize a local language.

[13] In Sect. 2.3.2.1 we defined in the same way a numbered r.e., for the purpose of formulating a rough criterion for checking if an r.e. is ambiguous.

4. Merge all the states that are tagged with the same set of followers. The obtained automaton is equivalent to the previous one. In fact, since the recognized language is local, the states tagged with equal sets of followers are undistinguishable (see Definition 3.7 on p.126).
5. Remove the numeric subscript from the symbols that label the automaton arcs, with the effect of returning to alphabet Σ.[14] By construction, the resulting automaton, which may be nondeterministic, accepts language $L\,(e\,\dashv)$.
6. Construct the deterministic equivalent automaton by applying the accessible subset construction (Sect. 3.7.2 on p. 144). When two or more states, each one tagged with a set, are merged into one state, the tag of the latter is the union of all the tags. The resulting automaton recognizes language $L\,(e\,\dashv)$.
7. Remove from the automaton the final state, tagged with $\emptyset$, and all its incoming arcs, which carry $\dashv$ as label. Mark as final the states of the resulting automaton tagged with a set that includes the terminator $\dashv$.

The resulting automaton is deterministic and recognizes language $L\,(e)$. □

Historical note: restricting Algorithm 3.39 to steps 1, 2, 5 and 7 yields another classical method called *GMY* (due to Glushkov and to McNaughton–Yamada), to obtain a nondeterministic real-time machine that recognizes the language of a generic r.e. Now, each state is identified by one letter of the numbered alphabet Σ_N. In contrast to the result of Algorithm 3.39, a *GMY* automaton may be nondeterministic. For instance, from r.e. $a \mid a\,b$ numbered $a_1 \mid a_2\,b_3$, the nondeterministic moves $q_0 \xrightarrow{a} a_1$ and $q_0 \xrightarrow{a} a_2$ are obtained.

Example 3.40 (*det. recognizer for an r.e.*) For r.e. $e = (\,a \mid b\,b\,)^*\ (\,a\,c\,)^+$, we apply Algorithm 3.39 and obtain the intermediate results of Fig. 3.18. Explanation of the steps from a machine to the next one in Fig. 3.18 are as follows:

1. The numbered version of e is $e' = (\,a_1 \mid b_2\,b_3\,)^*\ (\,a_4\,c_5\,)^+$, and the alphabet is $\Sigma_N = \{\,a_1,\ b_2,\ b_3,\ a_4,\ c_5\,\}$.
2. Automaton A_1 is local normalized, and the local sets for building it are:

$$Ini\,(e'\,\dashv) = \{\,a_1,\ b_2,\ a_4\,\} \qquad\qquad Fin\,(e'\,\dashv) = \{\,\dashv\,\}$$

$$Dig\,(e'\,\dashv) = \{\,a_1a_1,\ a_1b_2,\ a_1a_4,\ b_2b_3,\ b_3a_1,\ b_3b_2,\ b_3a_4,\ a_4c_5,\ c_5a_4,\ c_5\,\dashv\}$$

3. In automaton A_2, the initial state is tagged with $Ini\,(e'\,\dashv) = \{\,a_1,\ b_2,\ a_4\,\}$ and the other states are tagged with sets of followers. Three distinct states are tagged with the same set, since $Ini\,(e'\,\dashv) = Fol\,(a_1) = Fol\,(b_3) = \{\,a_1,\ b_2,\ a_4\,\}$.

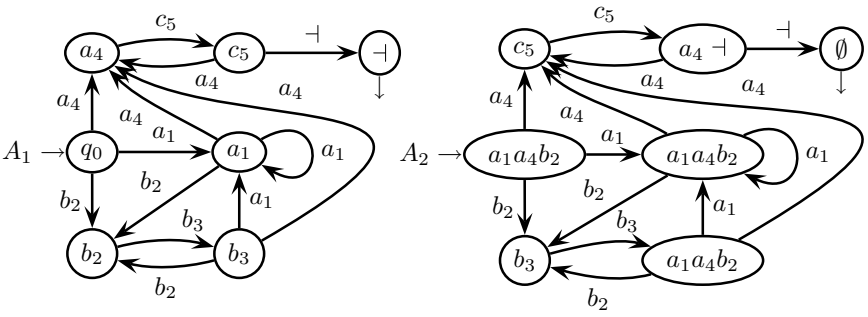
[14] This is an example of transliteration (homomorphism) as defined on p. 97.

numbered r.e.

$$e' \dashv = (a_1 \ \cup \ b_2\, b_3)^* \ (a_4\, c_5)^+ \dashv$$

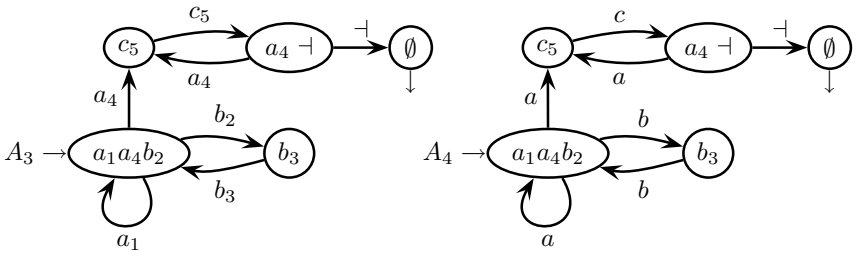
normalized local automaton $\downarrow$

A – *rename states* $\rightarrow$



merge the states with equal names $\downarrow$

A' – *relabel transitions* $\rightarrow$



build accessible subsets $\downarrow$

A'' – *final adjustment* $\rightarrow$

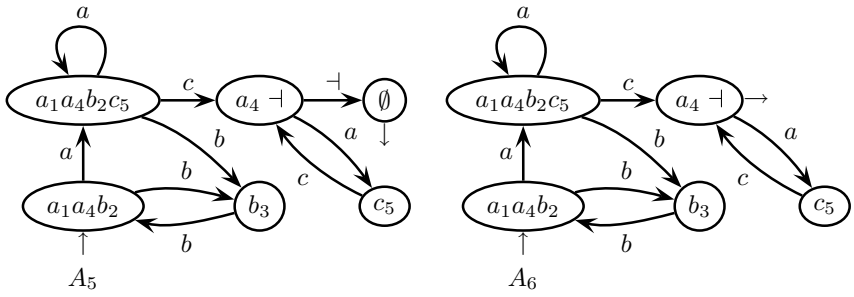


Fig. 3.18 Steps of Algorithm 3.39 for the recognizer of r.e. $e = (a \ | \ b\, b)^* \ (a\, c)^+$ (Example 3.40)

4. Automaton A_3 is obtained by merging all the identically tagged states.
5. Automaton A_4 recognizes language $L\left((a \mid b\,b)^* \, (a\,c)^+ \dashv \right)$ and is nondeterministic, due to the two a-labeled arcs from the initial state.
6. The deterministic automaton A_5 recognizes language $L\,(e \dashv)$.
7. The deterministic automaton A_6 is the recognizer of $e = (a \mid b\,b)^* \, (a\,c)^+$ and its (unique) final state is the one tagged with set $\{ a_4 \dashv \}$. □

3.8.2.4 Deterministic Recognizer by Berry–Sethi Algorithm

We combine in Algorithm 3.39 (p. 158) the construction of the normalized local automaton and the subsequent step that makes it deterministic, thus obtaining the Berry–Sethi algorithm (Algorithm 3.41).[15]

For the reader's convenience, we remind that the numbered version of a r.e. e over an alphabet Σ is denoted by e', and by $e' \dashv$ if followed by the end of text. By construction r.e. $e' \dashv$ is linear and its alphabet is $\Sigma_N \cup \{ \dashv \}$. For any symbol $a_i \in \Sigma_N$, the set of followers of a_i, i.e., $Fol\,(a_i) \subseteq \Sigma_N \cup \{ \dashv \}$, is defined as $Fol\,(a_i) = \left\{ b_j \mid a_i\,b_j \in Dig\,(e' \dashv) \right\}$.

Instead of starting from the local automaton for the linear r.e. $e' \dashv$ and transforming it into an equivalent deterministic automaton, the optimized algorithm incrementally generates the states of the final deterministic machine, each one identified by the set of followers of the last letter that has been read. Thus, the construction avoids state duplication and performs determinization on the fly, by unifying the states reached through distinctly subscripted versions $a_i, a_j \in \Sigma_N$ of the same letter $a \in \Sigma$.

Each state q is tagged and identified by a set of numbered letters and the terminator $\dashv$. Such a set is denoted by the notation $I\,(q) \subseteq \Sigma_N \cup \{ \dashv \}$ ("I" stands for Internal). We observe that in general $I\,(q)$ contains several elements, which just differ in their subscript, such as b_2 and b_5. It is convenient to refer to all such elements having in common the letter $b \in \Sigma$ by means of the notation $I_b\,(q) = \{ b_2, \; b_5 \} \subseteq I\,(q)$. We also say that the *class* of an element $b_j \in \Sigma_N$ is b.

Algorithm 3.41 (*Berry–Sethi algorithm—BS*) Each state is tagged and identified by a subset of $\Sigma_N \cup \{ \dashv \}$. A state is created marked as unvisited, and upon examination it is marked as visited to prevent the algorithm from reexamining it. The final states are those containing the end-of-text mark $\dashv$.

[15] For a thorough justification of the method we refer to [15].

Input: the sets *Ini* and *Fol* of a numbered r.e. $e' \dashv$

Output: recognizer $A = (\Sigma, Q, q_0, \delta, F)$ of the unnumbered r.e. e

$I(q_0) := Ini(e' \dashv)$ // create initial state q_0
unmark state q_0 // state q_0 is unexamined
$Q := \{ q_0 \}$ // initialize state set Q
$\delta := \emptyset$ // initialize transition function δ

while $\exists$ unmarked $q \in Q$ **do** //process each unexamined state q
  foreach $a \in \Sigma$ **do** // scan each input symbol a
    $I(q') := \emptyset$ // create new empty state q'
    unmark state q' // state q' is unexamined
    foreach $a_i \in I_a(q)$ **do** // scan each a_i of class a in q
      $I(q') := I(q') \cup Fol(a_i)$ // tag q' by a_i-followers
    if $q' \neq \emptyset$ **then** // if new state q' is not empty then
      if $q' \notin Q$ **then** // if q' is not in state set Q then
        $Q := Q \cup \{ q' \}$ // update state set Q
      $\delta := \delta \cup \{ q \xrightarrow{a} q' \}$ // update transition function δ
  mark state q // state q has been examined
$F := \{ q \in Q \mid I(q) \text{ contains } \dashv \}$ // create final state set F

Example 3.42 (*Berry-Sethi algorithm*) Apply Algorithm 3.41 to the r.e. $e = (a \mid b\,b)^* (a\,c)^+$ already considered in Example 3.40 on p. 159. The deterministic automaton obtained is shown in Fig. 3.19. For the numbered r.e. $e' \dashv$, the sets of initials and followers are:

$$e' \dashv = (a_1 \mid b_2\, b_3)^* (a_4\, c_5)^+ \dashv$$

$$Ini(e' \dashv) = \{ a_1, b_2, a_4 \}$$

$c \in \Sigma_N$	$Fol(c)$
a_1	$a_1\ b_2\ a_4$
b_2	b_3
b_3	$a_1\ b_2\ a_4$
a_4	c_5
c_5	$a_4\ \dashv$

We mention that the automaton thus produced may contain (moderately) more states than are necessary, but of course it can be minimized by the usual method. □

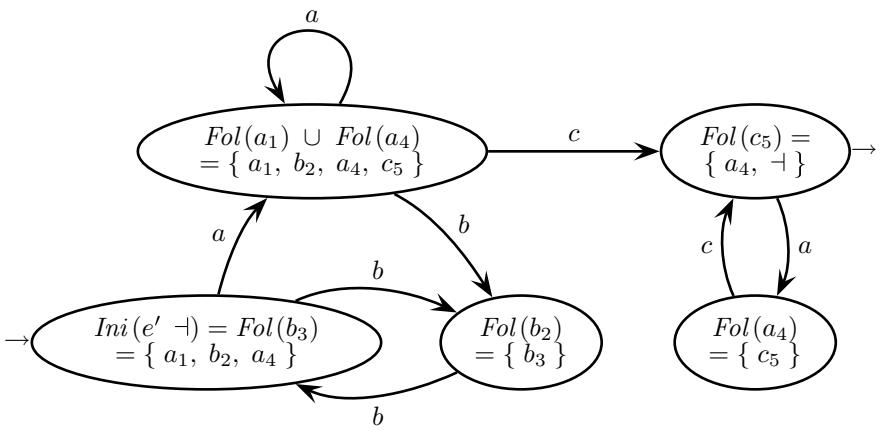


Fig. 3.19 Direct construction of a deterministic automaton for the r.e. $e = (a \mid b\,b)^*\ (a\,c)^+$ of Example 3.42 by *BS* algorithm

Use of Algorithm *BS* for Determinizing an Automaton

Algorithm *BS* is a valid alternative to the powerset construction (Algorithm 3.22) for converting a nondeterministic machine N into a deterministic one M. Actually, it is more flexible, since it applies to all nondeterministic forms, including multiple initial states and ε-arcs. We show how to proceed.

Algorithm 3.43 (*determinization of a finite automaton by algorithm BS*)

1. Distinctly number the labels of the non-ε-arcs of automaton N and so obtain the numbered automaton N', which has σ_N as alphabet.
2. Compute the local sets *Ini*, *Fin* and *Fol* for language $L(N')$, by inspecting the graph of N' and exploiting the identity $\varepsilon\,a = a\,\varepsilon = a$.
3. Construct the deterministic automaton M by applying algorithm *BS* (Algorithm 3.41) to the sets *Ini*, *Fin* and *Fol*. □

We justify the validity of Algorithm 3.43 by noting that the language recognized by automaton N' corresponds to all the successful computations, i.e., all the paths from an initial state to a final one, and that the sets *Ini*, *Fin* and *Fol*, which can be easily derived from the automaton graph by visual inspection, characterize exactly the labels of such successful computations. Thus, the language accepted by N' coincides with the set of the strings that can be obtained from the local sets in all possible ways, and it satisfies condition (3.1) on p. 151 and is local. We illustrate with an example.

Example 3.44 (*determinization of an automaton by algorithm BS*) Given the nondeterministic automaton N of Fig. 3.20 (top), by numbering the arc labels we obtain

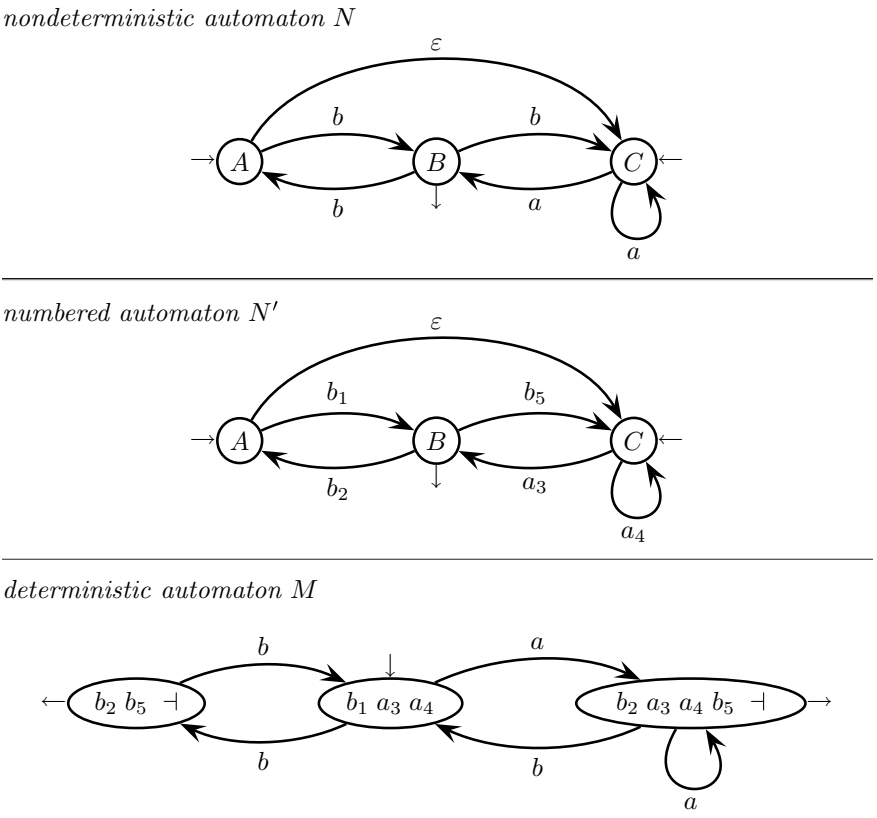


Fig. 3.20 Automaton N with spontaneous moves (top), numbered version N' (middle) and deterministic machine M constructed by algorithm *BS* (bottom) (Example 3.44)

automaton N' (middle). The language of N' is local and non-nullable. Then, compute the set of initials (left below):

$$Ini\left(L\left(N'\right)\ \dashv\right) = \{\ b_1,\ a_3,\ a_4\ \}$$

$c \in \Sigma_N$	$Fol\,(c)$
b_1	$b_2\ \ b_5\ \ \dashv$
b_2	$b_1\ \ a_3\ \ a_4$
a_3	$b_2\ \ b_5\ \ \dashv$
a_4	$a_3\ \ a_4$
b_5	$a_3\ \ a_4$

and note that $\varepsilon\, a_4 = a_4$ and $\varepsilon\, a_3 = a_3$. Proceed to compute the sets of followers (right above). At last, apply algorithm *BS* (Algorithm 3.41) to construct the deterministic automaton M of Fig. 3.20 (bottom). □

3.9 String Matching of Ambiguous Regular Expressions

Some recent widespread applications of regular expressions concern the analysis of large data sets, where r.e. is used to define *patterns* that need to be recognized. Sometimes the *structure* of the string to match is not defined uniquely; in this case it is typically described through an ambiguous r.e.

In this section we present a generalization of the Berry–Sethi method that allows one to build, from a possibly ambiguous r.e., a deterministic finite transducer, the output of which encodes all the possible ways to match the given string and the r.e. In Chap. 2 we have explained (Definition 2.17 on p. 21) how a string derives from an r.e., and in the continuation we show a tree representation of derivations and the way to compute it.

3.9.1 Trees for Ambiguous r.e. and Their Phrases

In the following, we assume that regular expressions are inductively defined as customary, with concatenation and union having $k \geq 2$ arguments, subexpressions enclosed within parentheses when necessary for operator precedence, and with the cross operator "+". The Kleene star operator "∗" is derived from the cross operator, with $e^* = (e)^+ \mid \varepsilon$. For convenience, we list the definition of an r.e. r:

$$r = \varepsilon \quad r = a \in \Sigma \quad r = r_1 \mid \ldots \mid r_k \quad r = r_1 \cdot \ldots \cdot r_k \quad r = (s) \quad r = (s)^+$$

where r, r_i (with $1 \leq i \leq k$) and s are r.e. (see also Sect. 2.3.1). The following r.e. will serve as running example:

Example 3.45 (*running example*) The r.e. below contains all the operators

$$\big((a)^+ \mid b\,a \mid a\,b\,a\big)^*\, b \tag{3.3}$$

and is equivalent to the following, without star:

$$e = \Big(\big((a)^+ \mid b\,a \mid a\,b\,a\big)^+ \mid \varepsilon\Big)\, b \tag{3.4}$$

□

We introduce an enriched representation of an r.e., called marked, which has the effect of inserting, in each phrase generated, certain new (meta)symbols that represent the derivation from the r.e. Intuitively, this process combines two familiar ideas: first the r.e. is parenthesized (similarly to the parenthesization of a context-free grammar), and then it is numbered to make each terminal symbol distinct.

Definition 3.46 (*marked regular expression*) Given an r.e. e over an alphabet Σ, the associated *marked regular expression*, shortly *m.r.e.*, denoted by $\check{e}$, is constructed through a two-step transformation.

Step 1 is a sort of translation: first (*i*) it replaces every (round) parenthesis present in r.e. e by a square bracket, to be viewed as a metasymbol of $\check{e}$; then (*ii*) it wraps each subexpression occurring between a pair of square brackets with a new pair of parentheses, which are instead meant as terminals for $\check{e}$. All the other original terminals, operators and epsilon letters of e are left unchanged in $\check{e}$.

Step 2 takes the result of step 1 and attaches a distinct subscript to all the terminals, both new and original, but not to the metasymbols.

Here is a snapshot for the r.e. e above:

$$
\begin{aligned}
e = (a\,b)^+ \mid c &\xRightarrow{\text{step 1.}i} [a\,b]^+ \mid c && \text{rewrite pair ( ) as [ ]}\\
&\xRightarrow{\text{step 1.}ii} ([a\,b]^+) \mid c && \text{wrap with a new pair ( )}\\
&\xRightarrow{\text{step 2}} {}_1([a_2\,b_3]^+)_1 \mid c_4 = \check{e} && \text{add subscript to terminals}
\end{aligned}
$$

Notice that the metasymbols "[ ]", "|" and "+" do not have a subscript. The use of identical subscripts for matching parentheses is just aesthetic.

More formally, the following (recursive) rules formalize the transformation:

step 1 Apply to e the translation function $T[\![\,]\!]$ defined inductively as follows, where all the e_i are r.e. ($1 \le i \le k$) and $a \in \Sigma$:

$$
\begin{aligned}
T[\![\,\varepsilon\,]\!] &= \varepsilon\\
T[\![\,a\,]\!] &= a\\
T[\![\,(e)\,]\!] &= \big(\,\big[\,T[\![\,e\,]\!]\,\big]\,\big)\\
T[\![\,(e)^+\,]\!] &= \big(\,\big[\,T[\![\,e\,]\!]\,\big]^+\,\big)\\
T[\![\,e_1 \cdot \ldots \cdot e_k\,]\!] &= T[\![\,e_1\,]\!] \cdot \ldots \cdot T[\![\,e_k\,]\!]\\
T[\![\,e_1 \mid \ldots \mid e_k\,]\!] &= T[\![\,e_1\,]\!] \mid \ldots \mid T[\![\,e_k\,]\!]
\end{aligned}
$$

step 2 In the expression $T[\![\,e\,]\!]$ obtained from step 1, number the following symbols, say, by using an integer counter increasing from left to right: input symbols, epsilon symbols and open round parentheses. Attach to each such symbol the number as a subscript. Attach to each closed round parenthesis the same subscript as the twin open parenthesis. Thus, we have obtained a linear r.e. since all the m.r.e. terminals are made distinct. □

Applying the function T of Definition 3.46 to the snapshot (step 1) yields:

$$
\begin{aligned}
T[\![\,(a\,b)^+ \mid c\,]\!] &= T[\![\,(a\,b)^+\,]\!] \mid T[\![\,c\,]\!]\\
&= \big(\big[\,T[\![\,a\,b\,]\!]\,\big]^+\big) \mid c\\
&= \big(\big[\,T[\![\,a\,]\!]\,T[\![\,b\,]\!]\,\big]^+\big) \mid c\\
&= ([a\,b]^+) \mid c
\end{aligned}
$$

and then attaching the subscripts (step 2) yields ${}_1([\,a_2\,b_3\,]^+)_1 \mid c_4$ as before. For the r.e. e of the running example (Example 3.45) the final m.r.e. $\check{e}$ follows:

$$\check{e} = {}_1\Big(\Big[\,{}_2([\,{}_3([\,a_4\,]^+)_3 \mid b_5\,a_6 \mid a_7\,b_8\,a_9\,]^+)_2 \mid \varepsilon_{10}\,\Big]\Big)_1\, b_{11} \tag{3.5}$$

The reader may observe that an m.r.e. is similar to the numbered r.e. defined on p. 156 for algorithm *BS* (Algorithm 3.41), but that in addition it contains square brackets and numbered parentheses. The reason for numbering parentheses and epsilons, and not square brackets and operators, is simple: epsilons and parentheses have changed their role from metasymbols to terminals of a new alphabet, while operators and square brackets have the role of metasymbols. In fact, the square brackets surrounding the argument of an iterator "+" or some other subexpression have taken the role of the original parentheses present in the r.e. e.

Thus, the *terminal alphabet* of an m.r.e., denoted Ω, is the union of two parts: $\Omega = \Sigma_N \cup M_N$, respectively, called the *numbered input alphabet* (already used on p. 156), and the set of *numbered metasymbols*. For instance, for m.r.e. (3.5) the terminal alphabet is:

$$\Omega = \underbrace{\{\, a_4,\ b_5,\ a_6,\ a_7,\ b_8,\ a_9,\ b_{11}\,\}}_{\text{numbered input alphabet } \Sigma_N} \cup \underbrace{\{\, {}_1(,\ )_1,\ {}_2(,\ )_2,\ {}_3(,\ )_3,\ \varepsilon_{10}\,\}}_{\text{numbered metasymbols } M_N}$$

and the metasymbols are $M = \{\ `[',\ `]',\ `+',\ `\cdot',\ `\mid'\ \}$.

For any numbered input symbol, e.g., $b_8 \in \Sigma_N$, we refer to the "plain" symbol $b \in \Sigma$ it comes from, by saying that b_8 *is of class* b. Also, we introduce for convenience a function *flatten* from $L(\check{e})$ to $L(e)$ that, given any string $\omega \in L(\check{e})$, returns the corresponding input phrase by deleting all the subscripts and then canceling all the epsilons and round parentheses. Thus, $\mathit{flatten}\left({}_1({}_2(a_7\,b_8\,a_9)_2)_1\,b_{11}\right) = a\,b\,a\,b$ and $\mathit{flatten}\left({}_1(\varepsilon_{10})_1\,b_{11}\right) = b$.

Since an m.r.e. has a parenthesized form (see p. 44), it helps to represent its structure as a tree graph named *marked abstract syntax tree* (*MAST*), which is inductively defined by the clauses in the following table:

r.e. type	*marked abstract syntax tree* (*MAST*)
ε or $a \in \Sigma$	a leaf node labeled epsilon or a
$e_1 \mid \ldots \mid e_k$	a node "|" with k children, each being the root of e_j subtree
$e_1 \cdot \ldots \cdot e_k$	a node "$\cdot$" with k children, each being the root of e_j subtree
$[\,e\,]^+$	node "+" with one child which is the root of e subtree
$[\,e\,]$	root node of e tree

The *MAST* for m.r.e. (3.5) associated with the r.e. (3.3) is depicted in Fig. 3.21.

Strings and Trees Defined by m.r.e.

Now we consider a string $\omega \in (\,\Sigma_N \cup M_N\,)^*$ generated from an m.r.e. $\check{e}$ through derivations. We know that ω includes numbered input symbols and numbered

$$e = \Big(\Big((a)^+ \mid b\,a \mid a\,b\,a \Big)^+ \mid \varepsilon \Big)\, b$$

$$\breve{e} = {}_1(\, \Big[\, {}_2(\, \Big[\, {}_3(\, [\, a_4 \,]^+ \,)_3 \mid b_5\, a_6 \mid a_7\, b_8\, a_9 \,\Big]^+ \,)_2 \mid \varepsilon_{10} \,\Big]\,)_1 b_{11}$$

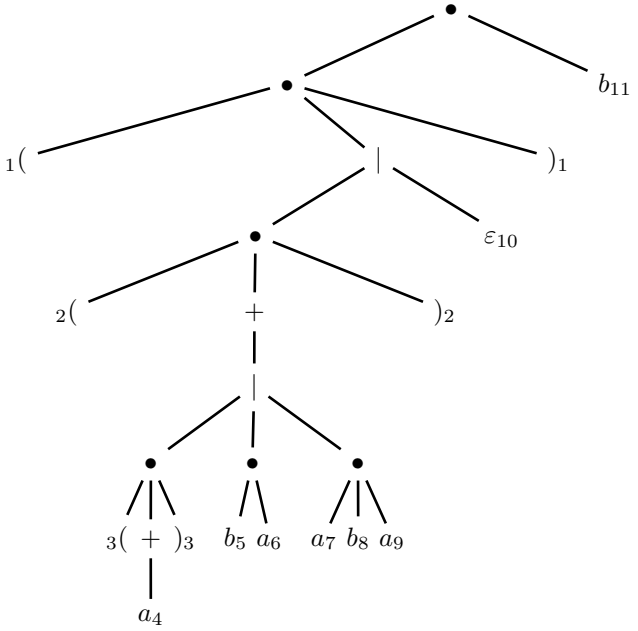


Fig. 3.21 Marked abstract syntax tree (*MAST*) of the running example r.e. e – cf. the corresponding m.r.e. $\breve{e}$ (3.5)

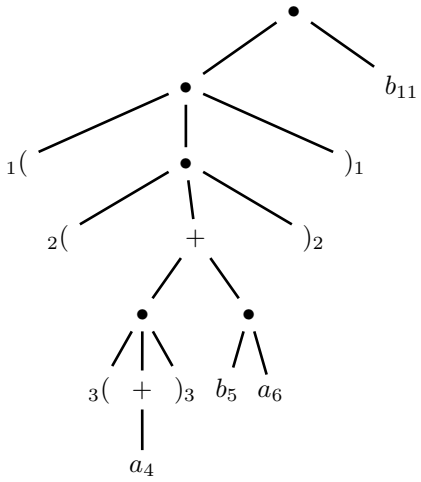
metasymbols, which altogether provide a complete description of how the phrase *flatten* $(\omega) \in \Sigma^*$ is derived from the original r.e. e.

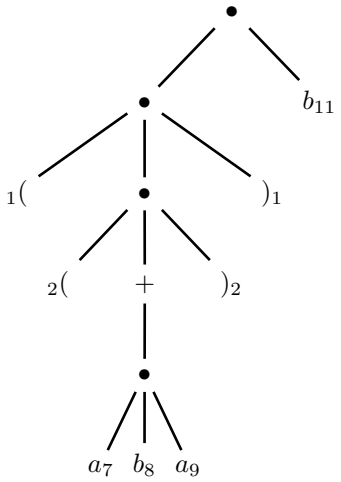
Suppose that r.e. e is ambiguous, and consider an ambiguous phrase x. We show that two or more strings are derived from the m.r.e. $\breve{e}$, all of which are mapped by function *flatten* to phrase x; each one encodes with numbered symbols and metasymbols, one of the alternative derivations of x from e.

To illustrate, the m.r.e. (3.5) derives the following two strings ω_1 and ω_2, which correspond to the same ambiguous phrase $a\,b\,a\,b$:

$$\omega_1 = {}_1(\, {}_2(\, {}_3(\, a_4 \,)_3\, b_5\, a_6 \,)_2 \,)_1\, b_{11} \quad \text{and} \quad \omega_2 = {}_1(\, {}_2(\, a_7\, b_8\, a_9 \,)_2 \,)_1\, b_{11} \tag{3.6}$$

Clearly, such strings have a parenthesized form that encodes a sort of syntax tree; therefore, it is appropriate to call them *linearized syntax trees* (*LST*). Next, we show how to move from such linearized tree to the tree graph in a way consistent with the representation of the r.e. by the *MAST* tree in Fig. 3.21. To reduce confusion, we call the tree of ω a *marked syntax tree* (*MST*), instead of "marked abstract syntax tree."





$\omega_1 = {}_1(\,{}_2(\,{}_3(\,a_4\,)_3\,b_5\,a_6\,)_2\,)_1\,b_{11}$ $\omega_2 = {}_1(\,{}_2(\,a_7\,b_8\,a_9\,)_2\,)_1\,b_{11}$

Fig. 3.22 Two *MST* trees for the ambiguous phrase $a\,b\,a\,b$ of r.e. $e = \big(((a)^+ \mid b\,a \mid a\,b\,a)^+ \mid \varepsilon\big)\,b$ – cf. Fig. 3.21. The two trees have as frontiers the strings ω_1 and ω_2 of $L(\check{e})$, called *linearized syntax trees* (*LST*)

The *MST* tree of an *LST* ω, defined by the m.r.e. $\check{e}$, is obtained from the *MAST* of the m.r.e. $\check{e}$ by applying essentially the same steps that are performed for deriving (Definition 2.17 on p. 21) ω from $\check{e}$. It suffices to outline the construction:

- When one alternative, say e_j, is chosen out of $e_1 \mid \ldots \mid e_k$, the subtrees of the discarded alternatives are eliminated from the *MAST*; the father node, labeled "|", becomes useless and is erased, taking care to preserve the path in the tree (this has the effect of shortening the path).
- When an iterator "+" is expanded by choosing the number of repetitions, an equal number of subtrees is appended to the iterator node "+", each subtree orderly corresponding to the substring generated in each iteration.

The *MST* trees obtained with the construction outlined, starting from the two *LST* $\omega_1,\ \omega_2 \in L(\check{e})$ (see (3.6)), are shown in Fig. 3.22. Clearly, the fact that phrase $a\,b\,a\,b$ has two different *LST* can be taken as a new criterion of ambiguity. The formal definitions and notations follow.

Definition 3.47 (*linearized syntax trees and ambiguity*) Let e and $\check{e}$ be an r.e and the corresponding m.r.e. Let $x \in L(e)$ be a phrase.

1. By taking all the *LST* of a single phrase x, or those of all the phrases in $L(e)$, we respectively define the *LST* set of one phrase or of a whole language, as follows:

$$LST(x) = \{\, \omega \in L(\check{e}) \mid \; x = flatten(\omega) \,\}$$
$$LST(e) = \bigcup_{x \in L(e)} LST(x)$$

2. Let $\omega \in LST(x)$ and τ be the tree graph *MST* having ω as frontier. Then τ and ω are, respectively, the graphic and the linearized representation of a possible derivation of phrase x from r.e. e.
3. A *phrase* $x \in L(e)$ is *ambiguous* if, and only if, set $LST(x)$ contains two or more strings; then we say that the r.e. is ambiguous. The *ambiguity degree* of x is the cardinality of set $LST(x)$. □

We show that Definition 3.47 specifies ambiguity more accurately than the previous criterion of Sect. 2.3.2.1 on p. 22, which was based on a simple but naive way of numbering an r.e.

Example 3.48 (*comparison of ambiguity criteria*) Consider the r.e. $e_1 = \big((a)^+ \mid b\big)^+ b$ and its m.r.e. $\check{e}_1 = {}_1([{}_2([a_3]^+)_2 \mid b_4]^+)_1 b_5$. For the phrase $a\,a\,b$, we have $LST(a\,a\,b) = \{\, \omega_1, \; \omega_2 \,\}$ where $\omega_1 = {}_1({}_2(a_3\,a_3)_2)_1 b_5$ and $\omega_2 = {}_1({}_2(a_3)_2\,{}_2(a_3)_2)_1 b_5$; therefore the ambiguity degree of $a\,a\,b$ is two. Figure 3.23 shows the two *MST*. The structure corresponding to ω_1 is obtained by iterating once the outer cross and twice the inner, whereas ω_2 iterates twice the outer cross and once the inner for each outer repetition.

On the other hand, the test on p. 22 fails to detect the ambiguity of string $a\,a\,b$, since from the naively numbered r.e. $e'_1 = \big((a_1)^+ \mid b_2\big)^+ b_3$ only one string derives that matches phrase $a\,a\,b$, namely string $a_1\,a_1\,b_2$. Differently stated, the two letters a match the same terminal a_1 of the numbered r.e. □

3.9.1.1 Infinite ambiguity

We focus on certain r.e. where the ambiguity degree is not finite, starting with an example.

Example 3.49 (*infinite ambiguity degree*) The r.e. $e_1 = (a \mid \varepsilon)^+$ defines language a^*. We consider the associated m.r.e:

$$\check{e}_1 = {}_1([a_2 \mid \varepsilon_3]^+)_1 \tag{3.7}$$

and we examine some phrases and their *LST*. The set $LST(a)$ of phrase a includes the following infinite list of strings, each string being represented by a different *MST* graph:

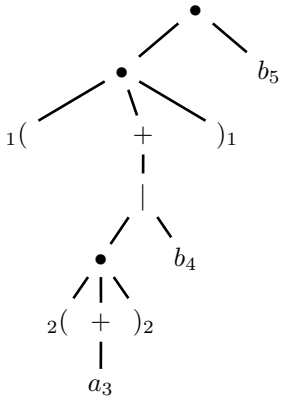
$${}_1(a_2)_1 \quad {}_1(\varepsilon_3\,a_2)_1 \quad {}_1(a_2\,\varepsilon_3)_1 \quad {}_1(\varepsilon_3\,a_2\,\varepsilon_3)_1 \quad {}_1(\varepsilon_3\,\varepsilon_3\,a_2)_1 \quad \ldots$$

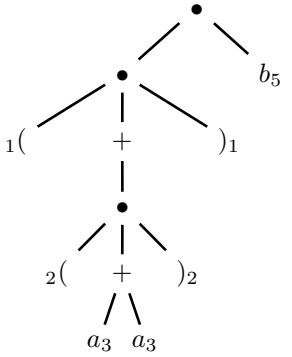
$e_1 = \big(\ (a)^+ \ \mid\ b\big)^+ b$

$\breve{e}_1 = \ {}_1(\ \big[\ {}_2(\ [a_3]^+\ )_2 \ \mid\ b_4\ \big]^+\ )_1\ b_5$

phrase $x = a\,a\,b$

$$LST(x) = \begin{cases} \omega_1 = \ {}_1(\ {}_2(\ a_3\ a_3\ )_2\ )_1\ b_5 \\ \omega_2 = \ {}_1(\ {}_2(\ a_3\ )_2\ {}_2(\ a_3\ )_2\ )_1\ b_5 \end{cases}$$





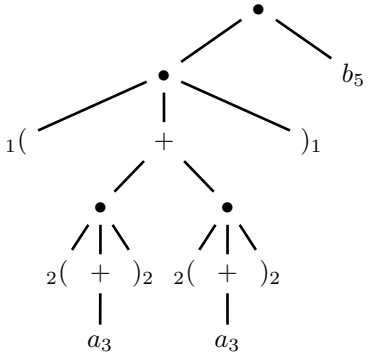


Fig. 3.23 *MAST* of the r.e. e_1 of Example 3.48 (top right). The two *MST* below represent the *LST* ω_1 and ω_2 (bottom)

Also, the empty string ε belongs to language $L\ (e_1)$ and has infinite ambiguity degree, since $LST\,(\varepsilon)$ contains ${}_1(\ \varepsilon_3\ )_1$, ${}_1(\ \varepsilon_3\ \varepsilon_3\ )_1$, etc. □

It is not difficult to see that every phrase has an infinite ambiguity degree, because the subexpression $(\ a\ \mid\ \varepsilon\ )$ is under cross "+" and is nullable.

It is customary to call *problematic* an r.e., as the one above, that generates some infinitely ambiguous phrase. More precisely, we define an r.e. as *problematic* if it contains (or coincides with) a subexpression of the form $(\ f\ )^*$ or $(\ f\ )^+$ where f is nullable; i.e., phrase ε is in $L\,(f)$. It can be proved[16] that an r.e. generates a phrase of infinite ambiguity degree if, and only if, it is problematic. In the next presentation of a string matching algorithm, we start from non-problematic r.e., and we defer the case of a problematic r.e. to the end.

[16] A rigorous discussion on problematic r.e. and ambiguity is in Frisch and Cardelli [17].

3.9.2 Local Recognizer of Linearized Syntax Trees

From Definition 3.47 we know that the m.r.e. $\check{e}$ associated to an r.e e defines all the linearized syntax trees $LST(x) \subseteq L(\check{e})$ of every phrase $x \in L(e)$. Of course, we can also view an m.r.e. as an r.e. over the alphabet $\Omega = \Sigma_N \cup M_N$ (p. 167). By construction, the symbols occurring in Σ_N are terminals made distinct by a subscript. Therefore, the m.r.e. is a *linear* r.e. and, from Property 3.37 on p. 156, the language $L(\check{e})$ is *local*. It follows that the recognizer of $L(\check{e})$ is a sliding window machine, which can be immediately obtained through Algorithm 3.30 on p. 152. We introduce the construction in the next example.

Example 3.50 (*local LST recognizer for non-problematic r.e.*) The middle part of Fig. 3.24 on p. 173 shows the *LST* recognizer for the ambiguous (non-problematic) r.e. e (3.4) on p. 165 (see also Figs. 3.21 and 3.22). The initial state and those entered by labeled input symbols are thicker. The bottom part of Fig. 3.24 shows a language-equivalent machine, which we may call "state trimmed": it is a machine with arcs labeled by a string or more generally by an r.e. To obtain the state-trimmed machine we use a method similar to the *BMC* elimination in Sect. 3.6: each maximal path composed by a series of arcs labeled with a metasymbol, followed by an arc with an input symbol label, is collapsed into a new arc, and the new arc label is the concatenation of the old labels. Therefore, in the state-trimmed graph each label is a string starting with zero or more metasymbols, and ending with a numbered input symbol or the terminator $\dashv$. Such strings are called *LST segments* or simply *segments*, and the input symbol at their end is called the *end symbol*.

In the middle part of Fig. 3.24 we observe that, since the the r.e. is non-problematic, i.e., not infinitely ambiguous, every path connecting two thick nodes without traversing a thick node is acyclic. Therefore, the number of strings that appear as arc labels of the bottom graph is finite. □

3.9.3 The Berry–Sethi Parser

We are almost ready for the final step: starting from the local recognizer of $LST(\check{e}\ \dashv)$, we construct a machine, called Berry–Sethi parser (*BSP*), that recognizes an input phrase $x \in L(e)$ and generates the set of linearized syntax trees $LST(x)$ of x, thus identifying one or more (if phrase x is ambiguous) ways by which phrase x can derive from r.e. e.

For future comparison, in Fig. 3.25 we show the classic *BS* recognizer (built by Algorithm 3.41 on p. 161) for the running example r.e. (3.3). We recall or sharpen certain terms pertaining to the classic *BS* recognizer, which are instrumental for the construction of the parser.

- The *BS* recognizer is now denoted by A_{BS}, and its components are accordingly denoted by Q_{BS} and δ_{BS}.

$$\breve{e} = {}_1(\left[{}_2(\left[{}_3([\,a_4\,]^+)_3 \;\mid\; b_5\,a_6 \;\mid\; a_7\,b_8\,a_9\right]^+)_2 \;\mid\; \varepsilon_{10}\right])_1\,b_{11}$$

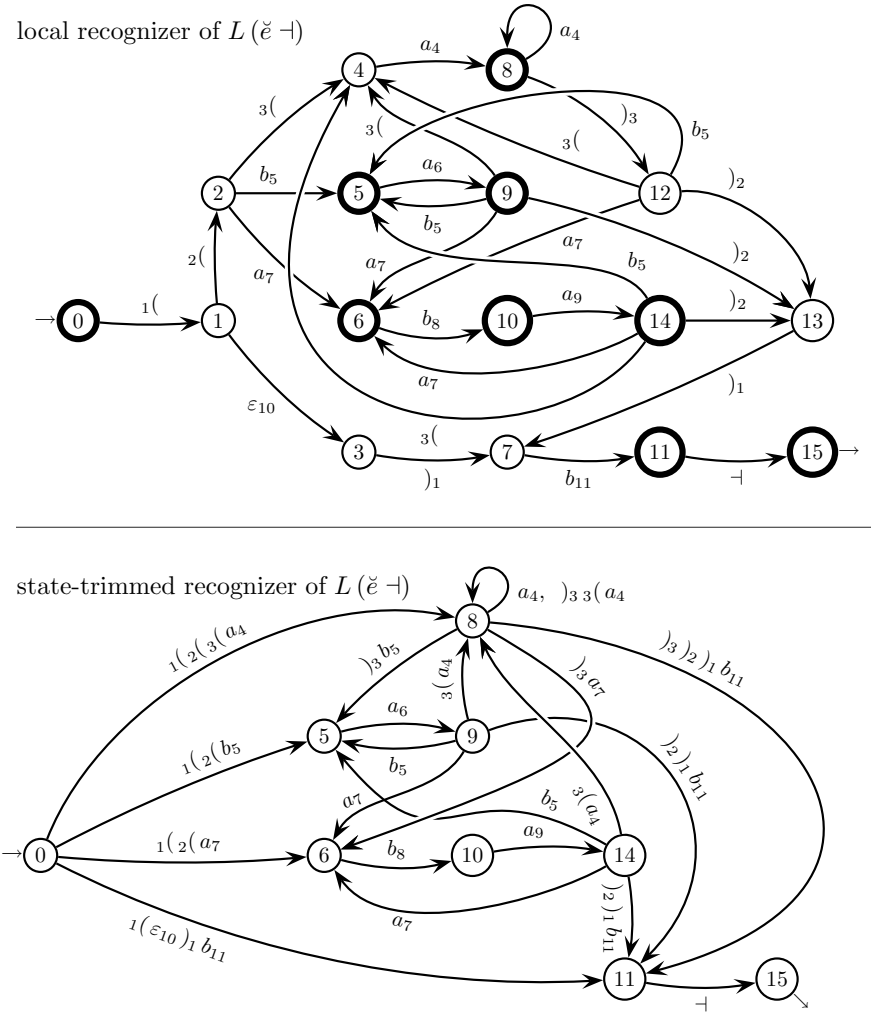


Fig. 3.24 m.r.e. (3.5) of Example 3.45 on p. 165 (top). The local recognizer of the *LST* language (middle). The language-equivalent recognizer retains the nodes entered by input characters (bottom), which are the thick ones of the local recognizer above. The arc labels of the bottom graph are called *segments*

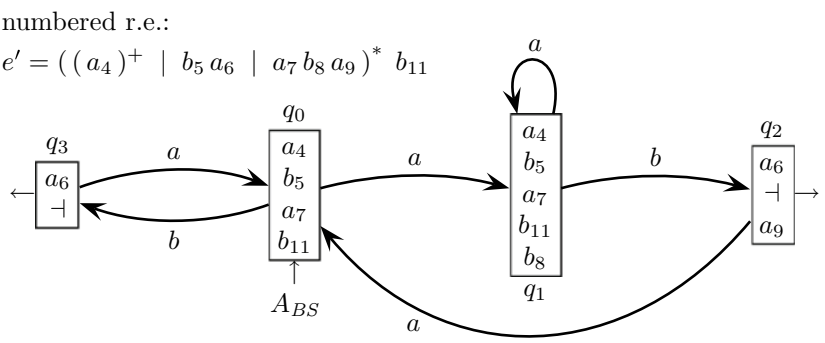


Fig. 3.25 Classic *BS* recognizer $A_{BS} = (\ Q_{BS},\ \Sigma,\ \delta_{BS},\ q_0,\ F_{BS}\ )$ for the running example r.e. e (Example 3.45), obtained from the numbered r.e. e' (p. 156)

- Each state $q \in Q_{BS}$ is identified by a set of numbered input symbols, denoted by $I\,(q) \subseteq \Sigma_N \cup \{\ \dashv\ \}$.
- For every terminal $a \in \Sigma$, we denote by $I_a\,(q) \subseteq I\,(q)$ the set of all the marked symbols of class a that are present in $I\,(q)$. For instance, in Fig. 3.25, it is $I\,(q_0) = \{\ a_4,\ b_5,\ a_7,\ b_{11}\ \}$, $I_a\,(q_0) = \{\ a_4,\ a_7\ \}$, $I_b\,(q_0) = \{\ b_5,\ b_{11}\ \}$ and $I_b\,(q_2) = \emptyset$.

The parser (also known as *LST* generator), named A_{BSP}, is a deterministic finite-state machine called a *transducer*,[17] which extends the recognizer A_{BS} with an *output function* denoted by ρ. At each transition, function ρ outputs a piece of information essentially consisting of metasymbols. The parser input is the string $\vdash x$, with $x \in \Sigma^+$, where for convenience the start-of-text symbol $\vdash$ is attached as a prefix. The initial state is *start*, and the initial transition from *start* reads $\vdash$ and outputs the first piece. When string x has been entirely scanned, the concatenation of such pieces encodes the set $LST\,(x)$, the cardinality of which is the ambiguity degree of x.

Output Function

Next, we have to specify the output function ρ. For that we have to briefly return to A_{BS}: we recall that an arc $q \xrightarrow{a} q'$ in δ_{BS} is associated with a set of digrams $b\,c \in Dig\,(e'\ \dashv)$, where e' is the input numbered version of r.e. e, and $a \in \Sigma, b \in \Sigma_N$ and $c \in \Sigma_N \cup \{\ \dashv\ \}$. Notice that the digrams are such that the symbols b and c are in the sets $I_a\,(q)$ and $I\,(q')$, respectively.

Then, the corresponding value $\rho\,(q, a)$ of the output function is a set of strings over the alphabet M_N of numbered metasymbols. Such strings are precisely the substrings of the fragments of linearized syntax tree that may be enclosed between symbols b and c.

To illustrate, we look-ahead to the transducer in Fig. 3.26 on p. 175, the states of which (except for *start*) coincide with the states of the A_{BS} in Fig. 3.25. The

[17] The transducer model will be discussed at length in Chap. 5.

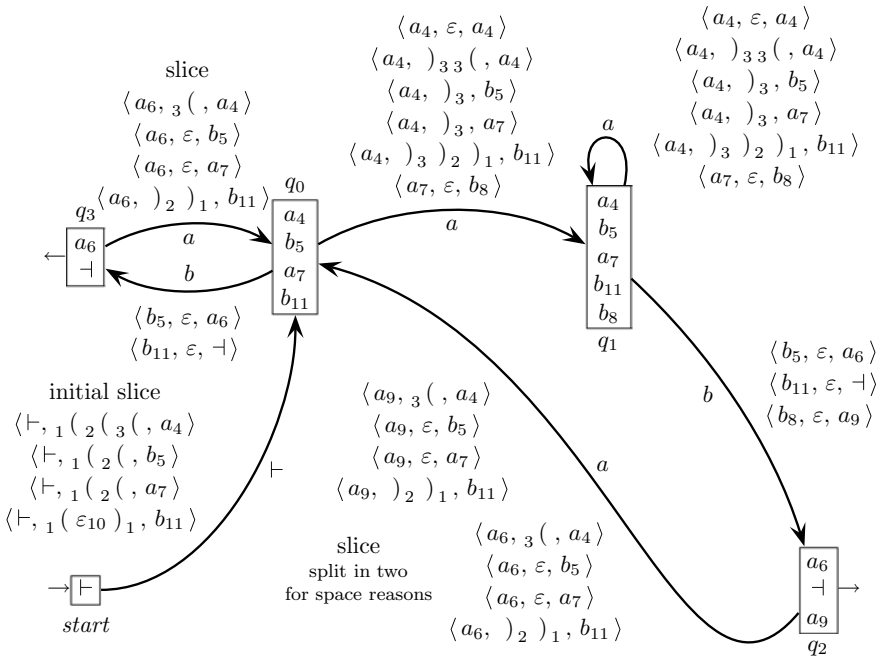


Fig. 3.26 Graph of the finite transducer A constructed by Algorithm 3.55 as *BSP* parser for the running example r.e. e (Example 3.45). The sets of initials Ini_{SE} and of followers Fol_{SE} are listed in Table 3.5. Each arc carries an input letter and the output slice, i.e., a set of 3-tuples (the curly brackets around each slice are omitted)

Table 3.4 Some values of the output function ρ of the Berry–Sethi parser (*BSP*)

Diagram	Metasymbolic string	3-tuple representation
$a_4\, a_4$	empty string	$\langle a_4\, ,\ \varepsilon\, ,\ a_4 \rangle$
$a_4\, a_4$	$)_3\, 3($	$\langle a_4\, ,\ )_3\, 3(\, ,\ a_4 \rangle$

output function ρ associated with the transition $q_0 \xrightarrow{a} q_1$ includes, among others, the following two strings of numbered metasymbols: the empty string ε and "$)_3\, 3($", which are enclosed inside the digram $a_4\, a_4$. We represent each such string as the middle component within a 3-tuple, having as first and last components the elements of the digram, as schematized in Table 3.4. Later on, a set of 3-tuples will be called a *slice*.

It helps to observe how the output function values are also visible in the local recognizer in Fig. 3.24 (top). The strings ε and "$)_3\, 3($" respectively correspond to the arc $8 \xrightarrow{a_4} 8$ and to the path $8 \xrightarrow{)_3} 12 \xrightarrow{3(} 4 \xrightarrow{a_4} 8$. Still in Fig. 3.26, for the same arc $q_0 \xrightarrow{a} q_1$ the output contains, associated with the digram $a_4\, b_{11}$, also the 3-tuple $\langle\, a_4,\ )_3\,)_2\,)_1,\ b_{11}\, \rangle$, which in the local recognizer corresponds to the path:

$$8 \xrightarrow{)_3} 12 \xrightarrow{)_2} 13 \xrightarrow{)_1} 7 \xrightarrow{b_{11}} 11$$

We stress that the output produced by any transducer transition is of bounded size; therefore the machine only needs a finite memory.

3.9.3.1 Segments of Linearized Syntax Tree

To make algorithmic the construction of the transducer, we have to precisely define how to split each *LST* into the segments that are emitted by the output function ρ at each step.

Consider a possibly empty phrase $x \in L(e)$ with $|x| = n \geq 0$. Let $\omega \in L(\breve{e} \dashv)$ be an *LST* such that $x = \mathit{flatten}(\omega)$. We have already observed that string ω, which we prefer to terminate by $\dashv$, can be factorized into what we now call the *decomposition into segments*:

$$\begin{aligned} \omega \dashv &= \zeta_1 \cdot \ldots \cdot \zeta_j \cdot \ldots \cdot \zeta_n \cdot \zeta_{n+1} \\ &= \underbrace{\mu_1 a_1}_{\zeta_1} \cdot \ldots \cdot \underbrace{\mu_j a_j}_{\zeta_j} \cdot \ldots \cdot \underbrace{\mu_n a_n}_{\zeta_n} \cdot \underbrace{\mu_{n+1} \dashv}_{\zeta_{n+1}} \quad \text{where } n \geq 0 \text{ and} \\ &\quad \forall 1 \leq j \leq n+1 \quad \mu_j \in M_N^* \text{ and } a_j \in \Sigma_N \qquad \dashv \text{ replaces } a_{n+1} \end{aligned} \tag{3.8}$$

Each term ζ_j is a *segment*, each term μ_j denotes a (possibly empty) string of numbered metasymbols, and each symbol a_j (obviously $\mathit{flatten}(a_j) \in \Sigma$) is called the *end symbol* of ζ_j. The end symbol $\dashv$ of the last segment is sometimes unnecessary, and we may drop it or leave it implicit, thus identifying ζ_{n+1} with μ_{n+1}. For every *LST* ω, the decomposition into segments is obviously unique.

Definition 3.51 (*segment*) A *segment* is a string of type $\zeta = \mu a_h$ or $\zeta = \mu \dashv$, with $\mu \in M_N^*$ and $a_h \in \Sigma_N$. Given an r.e. e and its m.r.e. $\breve{e}$, we define the set, denoted $SE(e)$, of all the segments that occur in some string of the marked language $L(\breve{e} \dashv)$ (notice the end of text), as follows ($n \geq 0$):

$$SE(e) = \left\{ \zeta_j \mid \ \zeta_1 \ldots \zeta_j \ldots \zeta_{n+1} \in L(\breve{e} \dashv) \text{ and } \zeta_j \text{ is a segment} \right\}$$

□

It is essential to observe that, for every non-problematic r.e. e, the set $SE(e)$ is finite. Therefore, given e, it is possible to pre-compute $SE(e)$ and to use it as building block in the construction of the Berry–Sethi parser. This approach is similar to the use of the symbols of the numbered input alphabet Σ_N in the definition of the sets of initials and followers in the context of on the classic *BS* recognizer.

Next, we transpose from strings of characters to sequences of segments the familiar concepts of sets of initials and followers, by using the segments instead of the atomic letters of the alphabet. Such a transposition is possible since the number of segments in the set $SE(e)$ is finite and can be viewed as a (larger) alphabet of symbols.

Table 3.5 Set of *initial* segments and all the sets of *follower* segments for the r.e. e of the running example (Example 3.53). The total number of segments is 17

$\check{e} = {}_1(\left[{}_2([{}_3([a_4]^+)_3 \mid b_5\, a_6 \mid a_7\, b_8\, a_9]^+)_2 \mid \varepsilon_{10} \right])_1\, b_{11}$

Segment set	Segments				
$Ini_{SE}(e)$	${}_1(\varepsilon_{10})_1\, b_{11}$	${}_1({}_2(a_7$	${}_1({}_2(b_5$	${}_1({}_2({}_3(a_4$	
$Fol_{SE}(e, a_4)$	a_4	$)_3\, a_7$	$)_3\, b_5$	$)_3\, {}_3(a_4$	$)_3\,)_2\,)_1\, b_{11}$
$Fol_{SE}(e, b_5)$	a_6				
$Fol_{SE}(e, a_6)$	a_7	b_5	${}_3(a_4$	$)_2\,)_1\, b_{11}$	
$Fol_{SE}(e, a_7)$	b_8				
$Fol_{SE}(e, b_8)$	a_9				
$Fol_{SE}(e, a_9)$	a_7	b_5	${}_3(a_4$	$)_2\,)_1\, b_{11}$	
$Fol_{SE}(e, b_{11})$	$\dashv$				

Definition 3.52 (*initial/follower segment*) Given an r.e. e, consider its decomposition into segments $\zeta_1 \ldots \zeta_{n+1} \in L(\check{e})\, \dashv$. The set of the *initial segments* of e is:

$$Ini_{SE}(e) = \{\, \zeta_1 \mid \zeta_1 \ldots \zeta_{n+1} \in L(\check{e}\, \dashv) \,\}$$

Let $\zeta_j = \mu_j\, a_h$, with $1 \leq j \leq n$, be a segment having $a_h \in \Sigma_N$ (not $\dashv$) as end symbol. The *set of the segments that follow* a_h, for short Fol_{SE}, is:

$$Fol_{SE}(e, a_h) = \left\{\, \zeta_{j+1} \mid \zeta_1 \ldots \zeta_j\, \zeta_{j+1} \ldots \zeta_{n+1} \in L(\check{e}\, \dashv) \text{ and } \zeta_j = \mu_j\, a_h \,\right\}$$

We also say that ζ_{j+1} is a *follower* of a_h and that $Fol_{SE}(e, a_h)$ is the *set of followers* of a_h. The set $Ini_{SE}(e)$ and all the sets $Fol_{SE}(e, a_h)$ for $a_h \in \Sigma_N$ are finite and computable. □

We illustrate the definition with an example.

Example 3.53 (*initial/follower segments*) Table 3.5 lists the set Ini_{SE} and all the sets Fol_{SE} of the m.r.e. $\check{e}$ of the running example. The sets can be calculated by scanning the m.r.e. from left to right, applying essentially the same rules for computing the local sets (Table 3.3 on p. 157). We notice that, if we erase all the numbered metasymbols M_N in Table 3.5, the initial and follower symbols coincide with those of the classic *BS* method for the same r.e. e. □

3.9.3.2 Generation of the Parser

We put together the previous intuitions and definitions into Algorithm 3.55, which constructs the *BSP* parser of an r.e. e. As anticipated, it is convenient to model the parser as a deterministic finite transducer, called A. To save effort, we reuse some definitions of the *BS* recognizer (evidenced by a subscript *BS*) in the specifications of the transducer A.

Definition 3.54 (*finite transducer for* BSP) Given an r.e. e over the input alphabet Σ, the deterministic finite transducer A is specified as follows:

- The input alphabet is $\Sigma \cup \{ \vdash \}$ and the initial state is *start*.
- The state set Q, transition function δ and final state set F are:

$$
\begin{aligned}
&Q = Q_{BS} \cup \{ \mathit{start} \}, \text{ with } I(\mathit{start}) = \{ \vdash \} \\
&\delta = \delta_{BS} \cup \left\{ \mathit{start} \xrightarrow{\vdash} q_0 \right\}, \text{ where } q_0 \text{ is the initial state of } A_{BS} \\
&F = F_{BS}, \text{ i.e., the final states of } A \text{ coincide with those of } A_{BS}.
\end{aligned}
$$

- For the output function ρ, the domain is the same as the domain of δ and the range is:

$$
O = \wp\left(\left(\Sigma_N \cup \{ \vdash \} \right) \times M_N^* \times \left(\Sigma_N \cup \{ \dashv \} \right) \right)
$$

  in words, the domain is a set of sets of 3-tuples (a set of 3-tuples is a *slice*)
- The output function $\rho \colon Q \times \left(\Sigma \cup \{ \vdash \} \right) \to O$ is defined as follows:

$$
\forall q, q' \in Q_{BS} \quad \forall a \in \Sigma \quad \left(q \xrightarrow{a} q' \right) \in \delta_{BS} \quad \forall b, c \in I_a(q), I(q') :
$$

$$
\begin{cases}
\rho(\mathit{start}, \vdash) = \{ \langle \vdash, \mu, c \rangle \mid \mu\, c \in \mathit{Ini}_{SE}(e) \} & \text{– initial slice} \\
\rho(q, a) = \{ \langle b, \mu, c \rangle \mid \mu\, c \in \mathit{Fol}_{SE}(e, b) \} & \text{– slice of } a\text{-arc}
\end{cases}
$$

where a is an input letter (unnumbered), symbols b and c are numbered and respectively belong to the alphabets $b \in \Sigma_N$ and $c \in \Sigma_N \cup \{ \dashv \}$. In addition symbol b is of class a, i.e., $a = \mathit{flatten}(b)$. □

Notice that the output range is defined as a the powerset of the Cartesian product of three sets. Therefore, the value of the output function ρ is a finite set of 3-tuples of the form already considered in Table 3.4 on p. 175. We call such a set a *slice*, since it constitutes a section of the directed acyclic graph *DAG* that will be introduced in Sect. 3.9.3.3 to represent all the *LST* of a phrase. The pseudo-code constructing the *BSP* transducer is in Algorithm 3.55.

Algorithm 3.55 (*Berry–Sethi parser algorithm—BSP*) The deterministic Berry–Sethi transducer is defined by:

Input: the sets Ini_{BSP} and Fol_{BSP} of a marked regular expression $\check{e}$

Output: parser $A = (\Sigma \cup \{\vdash\}, Q, start, \delta, \rho, F)$ of e as transducer

$I(start) := \{\vdash\}$ // create initial state *start*
$Q := \{start\}$ // initialize state set Q
$\delta := \emptyset$ // initialize transition function δ
$\rho := \emptyset$ // initialize output function ρ

$I(q_0) := \{b_j \mid \mu\, b_j \in Ini_{SE}(e)\}$ // create new state q_0
unmark state q_0 // state q_0 is unexamined
$SLICE := \{\langle \vdash, \mu, b_j \rangle \mid \mu\, b_j \in Ini_{SE}(e)\}$ // initialize slice
$Q := Q \cup \{q_0\}$ // update state set Q

$\delta := \delta \cup \{start \xrightarrow{\vdash} q_0\}$ // update transition function δ

$\rho := \rho \cup \{start \xrightarrow{SLICE} q_0\}$ // update output function ρ

mark state *start* // state *start* has been examined

while ∃ unmarked $q \in Q$ **do** //process each unexamined state q
  foreach $a \in \Sigma$ **do** // scan each input symbol a
    $I(q') := \emptyset$ // create new empty state q'
    unmark state q' // state q' is unexamined
    $SLICE := \emptyset$ // initialize 3-tuple slice
    foreach $a_i \in I_a(q)$ **do** // scan each a_i of class a in q
      // tag q' by all end symbols of an a_i-follower
      $I(q') := I(q') \cup \{b_j \mid \mu\, b_j \in Fol_{SE}(e, a_i)\}$
      // update 3-tuple slice of output function ρ
      $SLICE := SLICE \cup \{\langle a_i, \mu, b_j \rangle \mid \mu\, b_j \in Fol_{SE}(e, a_i)\}$
    if $I(q') \neq \emptyset$ **then** // if new state q' is not empty then
      if $q' \notin Q$ **then** // if q' is not in state set Q then
        $Q := Q \cup \{q'\}$ // update state set Q
      $\delta := \delta \cup \{q \xrightarrow{a} q'\}$ // update transition function δ
      $\rho := \rho \cup \{q \xrightarrow{SLICE} q'\}$ // update output function ρ
  mark state q // state q has been examined
$F := \{q \in Q \mid I(q)$ contains $\dashv\}$ // create final state set F

We explain Algorithm 3.55, starting from beginning. As customary, the state-transition and output functions δ and ρ are presented as labels of arcs in machine A graph. In the first steps, the initial state *start,* identified by the tag $\vdash$, is created, then

the state q_0 (it was the initial state of A_{BS}), and the arc $start \xrightarrow{\vdash} q_0$ with the output ρ defined by the initial segments $Ini_{SE}(e)$.

Then, the external `while` loop examines every state $q \in Q$ and marks it to avoid reexamining. For each state q, the outer `for` loop creates a new state q' as the target of an arc $q \xrightarrow{a} \bullet$ outgoing from q, with $a \in \Sigma$. The inner `for` loop examines every numbered symbol a_i of class a in $I_a(q)$, and it includes in the new set $I(q')$ all the end symbols b_j of the segments $Fol_{SE}(e, a_i)$. The value of the output function ρ for the same arc (variable *SLICE*) is computed using the metasymbolic part μ of the same segments. As a last step, the algorithm identifies as final the states that contain $\dashv$.

We remark that the *BSP* state-transition graph is almost identical to that of the classical Berry–Sethi recognizer, with the only addition of the initial state *start* and transition $start \xrightarrow{\vdash} q_0$ (with the initial slice). In fact, the *BSP* construction may be viewed as a generalization of the *BS* method.

Example 3.56 (*finite transducer representing BSP*) The finite transducer A for r.e. e in Fig. 3.26 has been computed by Algorithm 3.55, by using as parameters the sets of initials and followers Ini_{SE} and Fol_{SE} listed in Table 3.5. By construction, every *BSP* state is accessible from the initial state and is connected to a final state. □

Output Computed by *BSP*

A comparison with the *BS* recognizer A_{BS} in Fig. 3.25 on p. 174 confirms that, if we disregard the output function ρ and the initial state *start* of transducer A, the state-transition graphs of A and A_{BS} are identical. Moreover, for any pair of corresponding states of A and A_{BS}, the sets $I(\bullet)$ are identical. Therefore, the following relation holds between the languages recognized by the two automata: $L(A) = \vdash L(A_{BS})$; i.e., the two languages recognized just differ by the presence of the start-of-text.

Of course, the output function ρ makes the essential difference. We show on the example that the string output by transducer A for input x represents $LST(x)$.

Example 3.57 (*output of BSP*) We give the string $\vdash b$ as input to the transducer A in Fig. 3.26, and we compute the output $\rho(\vdash b)$:

$$\delta(start, \vdash b) = q_3$$

$$\rho(start, \vdash b) = \left\{ \begin{array}{c} \langle \vdash, {}_1({}_2({}_3(, a_4 \rangle \\ \langle \vdash, {}_1({}_2(, b_5 \rangle \\ \langle \vdash, {}_1({}_2(, a_7 \rangle \\ \langle \vdash, {}_1(\varepsilon_{10})_1, b_{11} \rangle \end{array} \right\} \cdot \left\{ \begin{array}{c} \langle b_5, \varepsilon, a_6 \rangle \\ \langle b_{11}, \varepsilon, \dashv \rangle \end{array} \right\}$$

slice of $start \xrightarrow{\vdash} q_0$ slice of $q_0 \xrightarrow{b} q_3$

String b is unambiguous and has just one *LST*, which is recognizable as the concatenation of the 3-tuples $\langle \vdash, {}_1(\varepsilon_{10})_1, b_{11} \rangle \cdot \langle b_{11}, \varepsilon, \dashv \rangle$, and represents the *LST*:

$_1(\ \varepsilon_{10}\ )_1\, b_{11}$. Notice that $\rho\,(\vdash b)$ includes also some 3-tuples, e.g., $\langle \vdash,\ _1(\ _2(,\ a_7\,\rangle$ and $\langle\, b_5,\ \varepsilon,\ a_6\,\rangle$, that do not contribute to any *LST* for the input b and seem unproductive.

In fact, every such 3-tuple will occur in the output of some longer phrase. Consider the input $\vdash b\,a\,b$, which has just one linearized tree:

$$LST\,(b\,a\,b) = {}_1(\ {}_2(\ b_5\ a_6\ )_2)_1\, b_{11}$$

The output is $\rho\,(start, \vdash b\,a\,b) = \rho\,(start, \vdash b) \cdot \rho\,(q_3,\ a\,b)$, and the 3-tuple $\langle\, b_5,\ \varepsilon,\ a_6\,\rangle$, previously useless for the input b, is now necessary. □

3.9.3.3 Construction of Linearized Syntax Trees

The output of the *BSP* transducer for input string x is a sequence of slices and can be interpreted as a *directed acyclic graph* (*DAG*), denoted by $DAG\,(x)$, acting as a compact representation of all the *LST* of x.

Each node of graph $DAG\,(x)$ corresponds to a pair $\langle\, slice\ number,\ 3\text{-}tuple\,\rangle$, where the first component is the ordinal position of a slice in the *DAG* and the second is a 3-tuple present in that slice. Each arc of the *DAG* is labeled by strings of numbered symbols and metasymbols, which are of the same type as the *segments* ζ_j occurring in an *LST*; see (3.8). On the *DAG* we show how to identify certain directed paths such that, by concatenating the labels on their arcs, one obtains all the *LST* of phrase x.

Example 3.58 (*DAG representing all the LST*) Consider Fig. 3.27, which is based on the accepting run of the *BSP* of Fig. 3.26 for the input $x = \vdash a\,a\,b$. The top part reports the transducer execution trace, including the states traversed and the output *slices* produced at each step. The middle part shows the *DAG* obtained from the execution trace above. The *DAG* contains 19 nodes, each one being identified by a pair $(slice\ number, 3\text{-}tuple)$, but for clarity we do not inscribe the slice number, which is instead represented by the column alignment. Two nodes in adjacent slices, i.e., in slice-i and slice-$(i+1)$, are connected by an arc if the third symbol in the 3-tuple of the left node is equal to the first symbol in the 3-tuple of the right node.

We define as *initial* and *final* a node that has the start-marker as first component or the end-marker as third component, respectively; final nodes are also marked by an exit arrow. A *DAG* path from an initial node to a final node is qualified as *recognizing*. By concatenating the middle components (of 3-tuples), i.e., the segments, of all the nodes on a recognizing path, we obtain a string over the alphabet Ω, which is an *LST* of the input string (disregarding the final $\dashv$).

In Fig. 3.26 there are two recognizing paths, evidenced by solid arcs and nodes; they spell out the two *LST* of phrase $a\,a\,b$. As a minor remark, on the exit arrow that marks a final node, the final segment is shown abbreviated without the $\dashv$. Thus, the segment sequence of the topmost recognizing path is:

$$\overbrace{{}_1(\ {}_2(\ {}_3(\ a_4}\ \overbrace{a_4}\ \overbrace{\ )_3\ )_2\ )_1\, b_{11}} \qquad \text{an } LST \text{ of phrase } a\,a\,b$$

To avoid clogging the drawing, only a few of the *DAG* arcs that do not contribute to any recognizing path are actually drawn (as dashed) in Fig. 3.27.

The bottom part of Fig. 3.27 depicts the marked syntax trees corresponding to the two *LST*. □

3.9.3.4 Extending *BSP* to Problematic Regular Expressions

It remains to explain the construction of *BSP* in the case of a *problematic* r.e., which by definition (p. 170) generates one or more problematic phrases x that have an infinite ambiguity degree; i.e., the cardinality of set $LST(x)$ is unbounded. We also know that such r.e. contains an iterated subexpression, $(\ f\ )^*$ or $(\ f\ )^+$, where the language of f is nullable.

Therefore, the segmented form of $LST(x)$ necessarily contains some segment of unbounded size. It follows that the *BSP* transducer computed by our Algorithm 3.55 on p. 179 would not be a finite-state machine, and the entire parser construction collapses.

To find a way out of this dead end, we study the *LST* set of a problematic phrase x and we argue that, in practice, not much is lost if we cut set $LST(x)$ by keeping just finitely many representative *LST* strings. Thus, we obtain a finite subset that we call *acyclic* and we denote by $LST_{acyclic}(x) \subseteq LST(x)$. After thus restoring the indispensable condition that all the segments are finite, we can use Algorithm 3.55 anew, for a problematic r.e. too.

Example 3.59 (*bounding the ambiguity degree*) In Example 3.49 on p. 170, the (problematic) r.e. $e_1 = (\ a\ \mid\ \varepsilon\ )^+$, with m.r.e. $\check{e}_1 = {}_1(\,[\,a_2\ \mid\ \varepsilon_3\,]^+\,)_1$, generates only problematic phrases, such as a. The set $LST(a)$ contains infinitely many *LST*, and a few of them are listed in Example 3.49.

In practice, we can scarcely imagine a situation requiring all the possible *LST* of such infinitely ambiguous phrases. Just a few *LST* suffice, which can (somewhat subjectively) represent all the possibilities. The remaining *LST* strings are simply removed from language $L(\check{e})$. A plausible criterion is to discard all the strings that contain *two or more* identically numbered copies of the empty string, i.e., a substring $\varepsilon_k \ldots \varepsilon_k$, without an input symbol of Σ_N in between. We call *cyclic* such strings to discard, and consequently *acyclic* those we choose to keep. Thus, we partition set $LST(a)$ into a finite set of representative *LST* and an infinite set of discarded *LST*:

$$\underbrace{{}_1(\,a_2\,)_1 \quad {}_1(\,\varepsilon_3\,a_2\,)_1 \quad {}_1(\,a_2\,\varepsilon_3\,)_1 \quad {}_1(\,\varepsilon_3\,a_2\,\varepsilon_3\,)_1}_{\text{finite set of representative } LST \text{ —acyclic } LST}$$

$$\underbrace{{}_1(\,\varepsilon_3\,\varepsilon_3\,a_2\,)_1 \quad {}_1(\,a_2\,\varepsilon_3\,\varepsilon_3\,)_1 \quad {}_1(\,\varepsilon_3\,\varepsilon_3\,a_2\,\varepsilon_3\,)_1 \quad \ldots}_{\text{infinite set of discarded } LST\text{—cyclic } LST}$$

Intuitively, our approach uses the acyclic *LST* set to construct the finite-state transducer. Before doing that, we observe the local recognizer of language $LST(e)$ and

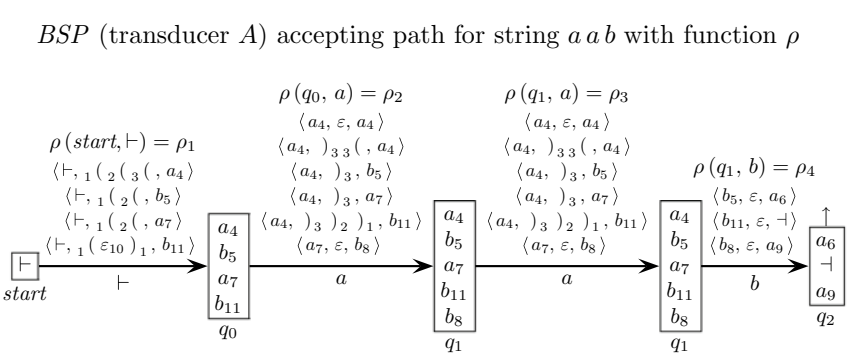


Fig. 3.27 *LST* construction for string *a a b* of Example 3.58. Accepting path of the *BSP A* (top). *DAG* of string *a a b* and the two recognizing paths (middle). The brackets and ordinal numbers of the 3-tuples are omitted. Marked and linearized syntax trees (bottom)

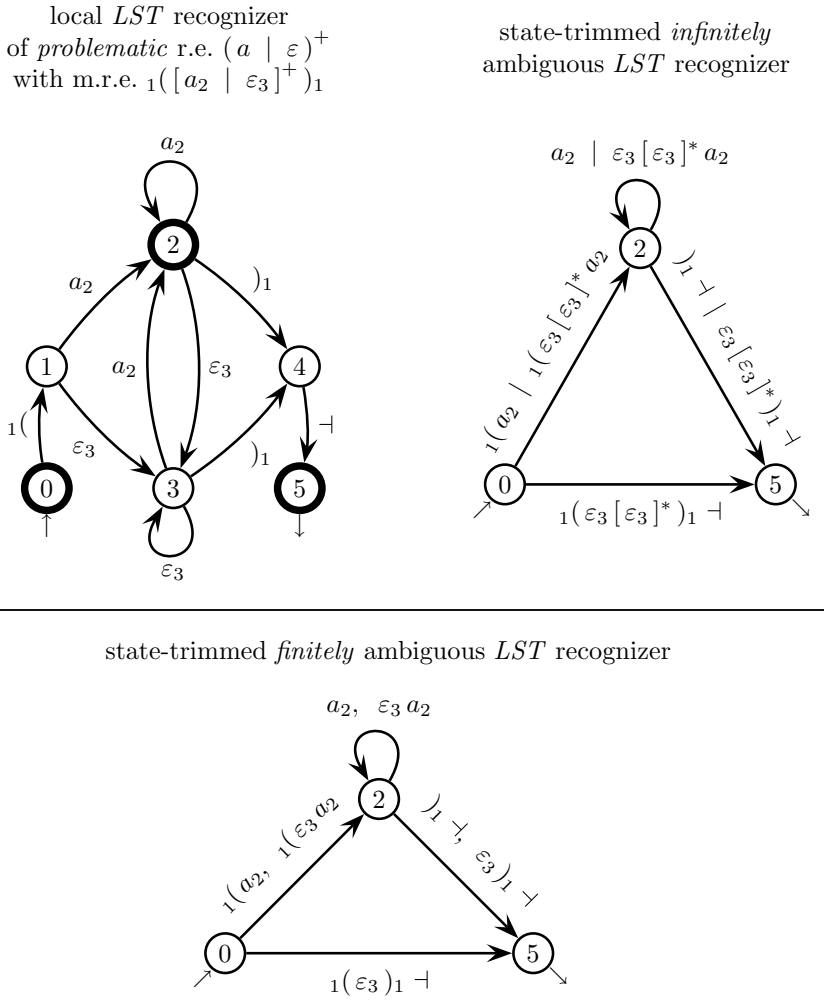


Fig. 3.28 Local *LST* recognizer for the problematic (infinitely ambiguous) r.e. e_1 of Example 3.59 (top left). The equivalent state-trimmed machine (top right). The same state-trimmed machine with ambiguity degree cut down to a finite value (bottom)

the state-trimmed one, shown in Fig. 3.28 (top). We compare with the recognizer of a non-problematic r.e. in Fig. 3.24 on p. 173 (top and middle), to bring out the essential difference. For the non-problematic case, every path that connects two thick nodes without traversing a thick node is acyclic. Therefore in the state-trimmed graph, every arc label is a regular language of finite cardinality.

On the other hand, for the problematic case, the acyclicity property falls for the local *LST* recognizer shown in Fig. 3.28 (top left). Here some paths from thick node to thick node are cyclic, e.g., path 0, 1, 3, 3, …, 3, 2. Therefore, when eliminating

the thin nodes while preserving equivalence, some arc labels of the state-trimmed machine in Fig. 3.28 (top right) contain the iteration operator and define an infinite language. See, e.g., r.e. $_1(a_2 \ | \ _1(\varepsilon_3 [\varepsilon_3]^* a_2$ on arc $0 \rightarrow 2$. □

As anticipated, we want to keep only a finite number of ambiguous *LST* for a given phrase. To this effect, for each pair of thick nodes we compute the set of paths that connect the first node to the second one and satisfy the following two constraints that ensure finiteness:

1. The path does not traverse any intermediate thick node.
2. The path includes at most one arc labeled with the same numbered empty string symbol ε_k.

The label of such preserved path is called an *acyclic segment* (*ASE*) of *LST*. The *ASE* are written on the label of the arc that connects two (formerly thick) nodes in the state-trimmed machine. Figure 3.28 (bottom) shows the resulting state-trimmed recognizer with ambiguity degree cut down to a finite value. Notice that, in accordance with constraint 2 above, the label of the cyclic path $2 \rightarrow 3 \rightarrow 3 \rightarrow \ldots \rightarrow 4 \rightarrow 5$ in Fig. 3.28 (top left) is not an *ASE*; therefore it is not attached to the arc $2 \rightarrow 5$ of the bottom graph in the same figure.

Definition 3.60 (*acyclic marked language and LST*) Let $ASE\,(e) \subseteq SE\,(e)$ be the set of all the *acyclic* segments of an r.e. e (of course such *ASE* is based on the m.r.e. $\check{e}$ of e). For r.e. e, we define the *acyclic marked language* $L_{acyclic}\,(\check{e})$ generated by the m.r.e. $\check{e}$, as:

$$L_{acyclic}\,(\check{e}) = \left\{ \zeta_1 \ldots \zeta_j \ldots \zeta_{n+1} \in L\,(\check{e}\,\dashv) \mid n \geq 0 \text{ and every } \zeta_j \in ASE\,(e) \right\}$$

For a phrase $x \in L\,(e)$, we define the set of *acyclic linearized syntax trees*, as:

$$LST_{acyclic}\,(x) = LST\,(x) \ \cap \ \left(L_{acyclic}\,(\check{e}) \setminus \{ \dashv \} \right)$$

The quotient "\" (Sect. 2.2.5) cancels the trailing end-of-text $\dashv$, and the intersection filters out any string that contains a non-acyclic segment. □

Notice that if an r.e. e is non-problematic, the two notions of marked language coincide: $L_{acyclic}\,(\check{e}) = L\,(\check{e})$. To construct the *BSP* parser of a problematic r.e., we proceed as for a non-problematic one, taking care to use the *ASE* set instead of the *SE* set. We compute the sets of initial segments and follower segments, which we call $Ini_{ASE}\,(e)$ and $Fol_{ASE}\,(e)$, respectively. At last, we construct the *BSP* finite transducer through the same Algorithm 3.55 on p. 179, using in the pseudo-code the sets Ini_{ASE} and Fol_{ASE} instead of Ini_{SE} and Fol_{SE}, respectively.

Example 3.61 (*BSP for problematic r.e.*) We illustrate the problematic case on r.e. e_1 of Example 3.59. The acyclic segment set *ASE*, and the various sets Ini_{ASE} and

Table 3.6 Set of acyclic *initial* segments and all the sets of acyclic *follower* segments for the problematic r.e. e_1 of Example 3.59 on p. 182

$\breve{e}_1 = {}_1([\, a_2 \mid \varepsilon_3 \,]^+)_1$

Segment set	Acyclic segments (total of 7 sorted as in Table 3.5)			
$ASE\,(e_1)$	$a_2 \quad \varepsilon_3\, a_2 \quad \varepsilon_3\,)_1 \dashv \quad {}_1(a_2 \quad {}_1(\varepsilon_3\, a_2 \quad {}_1(\varepsilon_3\,)_1 \dashv \quad)_1 \dashv$			
$Ini_{ASE}\,(e_1)$	${}_1(a_2$	${}_1(\varepsilon_3\, a_2$	${}_1(\varepsilon_3\,)_1 \dashv$	
$Fol_{ASE}\,(e_1,\ a_2)$	a_2	$\varepsilon_3\, a_2$	$\varepsilon_3\,)_1 \dashv$	$)_1 \dashv$

Fol_{ASE} are listed in Table 3.6. The whole procedure is carried out in Fig. 3.29: finite transducer A of e_1 (top), two execution traces (overlapped) for phrases ε and a (middle left), corresponding (also overlapped) two *DAG* (middle right), and acyclic *LST* (and *MST*) obtained (bottom), only one for ε (left) and four for a (right). The traces and *DAG* of strings ε and a overlap as ε is a prefix of a. An *LST* (at least) is eventually found for each problematic phrase ε and a. □

Notice however that phrase a has four acyclic *LST*, whereas phrase ε has only one and it appears to be non-ambiguous: in other words, the *BSP* fails to detect the ambiguity of ε. This is a minor flaw of our construction: since infinite ambiguity is cut down to finite, certain (somewhat pathological) phrases, though correctly recognized, are presented as if they were not ambiguous.

We hint to one remedy which requires a slight correction of the acyclic segment definition. Infinite ambiguity should be cut by discarding all the segments that contain *three or more* (instead of two or more) equally numbered empty string symbols ε_k without any input symbol in between, i.e., $\ldots \varepsilon_k \ldots \varepsilon_k \ldots \varepsilon_k \ldots$. In this way, phrase ε would be left with two *LST*, namely ${}_1(\varepsilon_3\,)_1$ as before, and ${}_1(\varepsilon_3\, \varepsilon_3\,)_1$, and its ambiguity would be discovered. We omit the simple adjustments of the *BSP* construction.

3.9.3.5 An operational condition for r.e. ambiguity

We return to the problem of deciding r.e. ambiguity, to present a decision method that can be easily applied to the *BSP* transducer graph. We recall that an r.e. e is ambiguous if some phrase $x \in L\,(e)$ can be derived in different ways that correspond to distinct syntax trees, *LST* or *MST*. This is equivalent to having two or more *LST* in the set $LST\,(x)$ (Definition 3.47). We show how to check the latter condition on the transducer graph, first for the non-problematic case.

Definition 3.62 (*operational ambiguity condition*) An r.e. e is ambiguous if its *BSP* transducer has a state q and an input symbol a, including $q = start$ and $a = \vdash$, such that (*i*) the output slice $\rho\,(q,\ a)$ contains two 3-tuples with an identical third component, i.e., $\langle\, a_h,\ \mu,\ c\,\rangle$ and $\langle\, a_k,\ \mu',\ c\,\rangle$, and (*ii*) the first components a_h and a_k of these 3-tuples are in the same class a. □

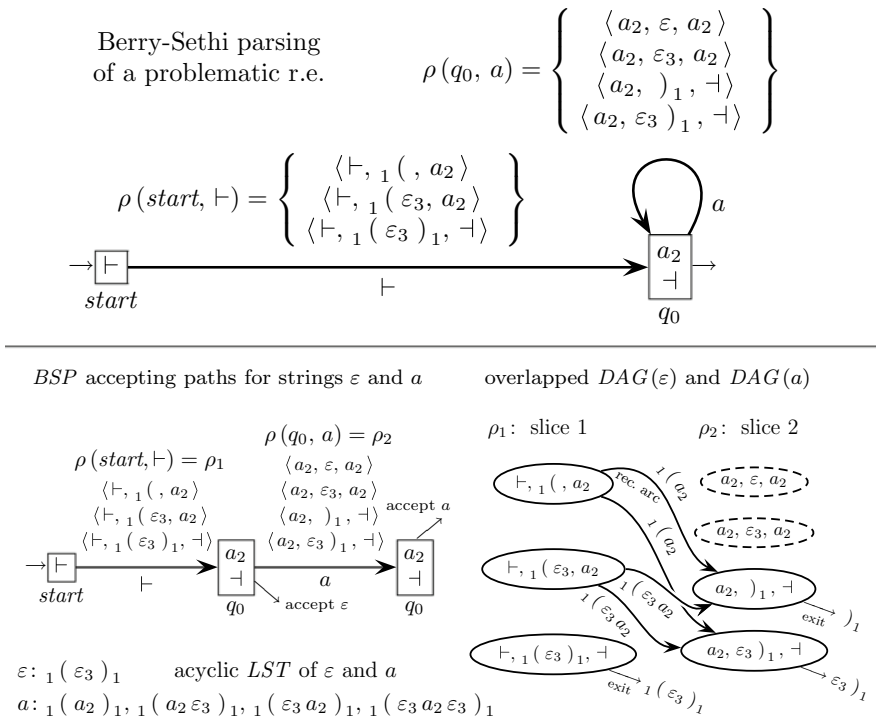


Fig. 3.29 Procedure for constructing the acyclic *LST* of the ambiguous strings ε and a for the problematic r.e. e_1 of Example 3.59. Notice that the ambiguity of ε is not preserved

To illustrate, in Fig. 3.26 the arc $q_0 \xrightarrow{a} q_1$ outputs a slice containing two 3-tuples with (*i*) the same end symbol $c = a_4$ and (*ii*) start symbols $a_h = a_k = a_4$. In Fig. 3.27, string $a\,a\,b$ exhibits the corresponding form of ambiguity. The *BSP* ambiguity condition (Definition 3.62) is also satisfied, still in the graph of Fig. 3.26, by the arc $q_2 \xrightarrow{a} q_0$ that outputs two 3-tuples with (*i*) the same end symbol $c = b_5$ and (*ii*) start symbols $a_h = a_6$ and $a_k = a_9$. For instance, this transition is executed in analyzing the ambiguous string $a\,b\,a\,b$.

To finish, we briefly discuss the problematic case. The *BSP* ambiguity condition (Definition 3.62) is valid for most problematic r.e. Yet, for certain such r.e. it may fail to detect ambiguity. A case is the formerly discussed r.e. e_1 of Example 3.59, though the failure there is only for the particular phrase ε. In other words, for a problematic r.e., within the current definition of acyclic segment (Sect. 3.9.3.4), the

BSP ambiguity condition is sufficient, yet not necessary in all cases. However, it could be slightly generalized so as to have it characterize ambiguity without any exception, as hinted at the end of Sect. 3.9.3.4.

3.10 Expressions with Complement and Intersection

Before leaving regular languages, we complete the study of the operations that were left suspended in Chap. 2: complement, intersection and set difference. This will allow us to extend regular expressions with such operations, to the purpose of writing more concise or expressive language specifications. By using the properties of finite automata, now we can state and prove a property (Property 3.63) anticipated in Chap. 1.

Property 3.63 (closure of REG under complement and intersection) *Let L and L' be regular languages. Their complement $\neg L$ and intersection $L \cap L'$ are regular languages.* □

Proof First we show how to build the recognizer of the complement language $\neg L = \Sigma^* \setminus L$. We assume that the recognizer M of language L is deterministic, with initial state q_0, state set Q, final state set F and transition function δ. The recognizer is constructed by Algorithm 3.64.

In the first step of Algorithm 3.64, we complete automaton M by adding the error or sink state p and the arcs to and from it, in order to make total the transition function δ of M. Then (steps two and three) machine M is complemented and the recognizer $\overline{M}$ of the complement language $\neg\, L\,(M)$ is obtained, i.e., $L\,(\overline{M}) = \neg\, L\,(M)$.

Notice that final and non-final states are interchanged from M to $\overline{M}$. To justify the construction, first observe that if a computation of automaton M accepts a string $x \in L\,(M)$, then the corresponding computation of automaton $\overline{M}$ terminates in a non-final state, so that $x \notin L\,(\overline{M})$. Second, if a computation of M does not accept a string y, two cases are possible: either the computation ends in a non-final state q or it ends in the sink p. In both cases the corresponding computation of $\overline{M}$ ends in a final state, meaning that string y is in the language $L\,(\overline{M})$. Thus we conclude that $L\,(\overline{M}) = \neg\, L\,(M)$.

Then, in order to show that the intersection of two regular languages is regular, it suffices to quote the well-known De Morgan identity:

$$L \cap L' = \neg\left(\neg L \cup \neg L'\right)$$

because, from knowing that the languages $\neg\, L$ and $\neg\, L'$ are regular, it follows that their union too is regular, as well as its complement. □

Algorithm 3.64 (*deterministic recognizer for complement*)

Input: A deterministic finite-state automaton M

Output: A deterministic finite-state automaton $\overline{M}$ complement of M

1 create a new state $p \notin Q$, the *sink*, and the state set of $\overline{M}$ is $Q \cup \{ p \}$

2 the transition function $\overline{\delta}$ of $\overline{M}$ is:

$\overline{\delta}(q, a) = \delta(q, a)$, where $\delta(q, a) \in Q$

$\overline{\delta}(q, a) = p$, where $\delta(q, a)$ is not defined

$\overline{\delta}(p, a) = p$, for every character $a \in \Sigma$

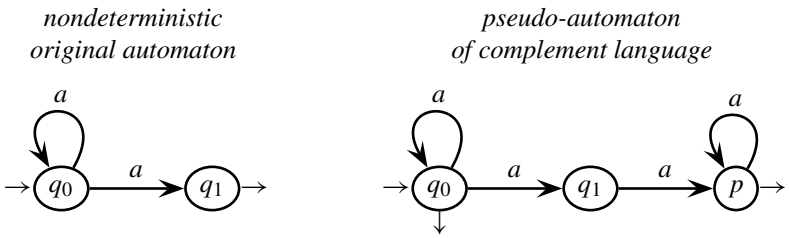
3 the final state set of $\overline{M}$ is $\overline{F} = (Q \setminus F) \cup \{ p \}$ □

As a corollary, the *set difference* of two regular languages L and L' is regular because of the identity:

$$L \setminus L' = L \cap \neg L'$$

Example 3.65 (*complement automaton*) Figure 3.30 shows three machines: the (deterministic) given one M, the intermediate one completed with sink and the (deterministic) recognizer $\overline{M}$ of the complement language. □

For the construction to work, the original machine has to be deterministic. Otherwise the language accepted by the constructed machine may be not disjoint from the original one. Such a fact fact would violate the characteristic property of complement, i.e., $L \cap \neg L = \emptyset$. See the following counterexample, where a pseudo-complement machine mistakenly accepts string a, which actually is in the original language:



Finally we mention that the complement construction may produce unclean or non-minimal machines. These however can always be cleaned through Example 3.5 on p. 124 and minimized as explained in Sect. 3.4.3.1 on p. 128.

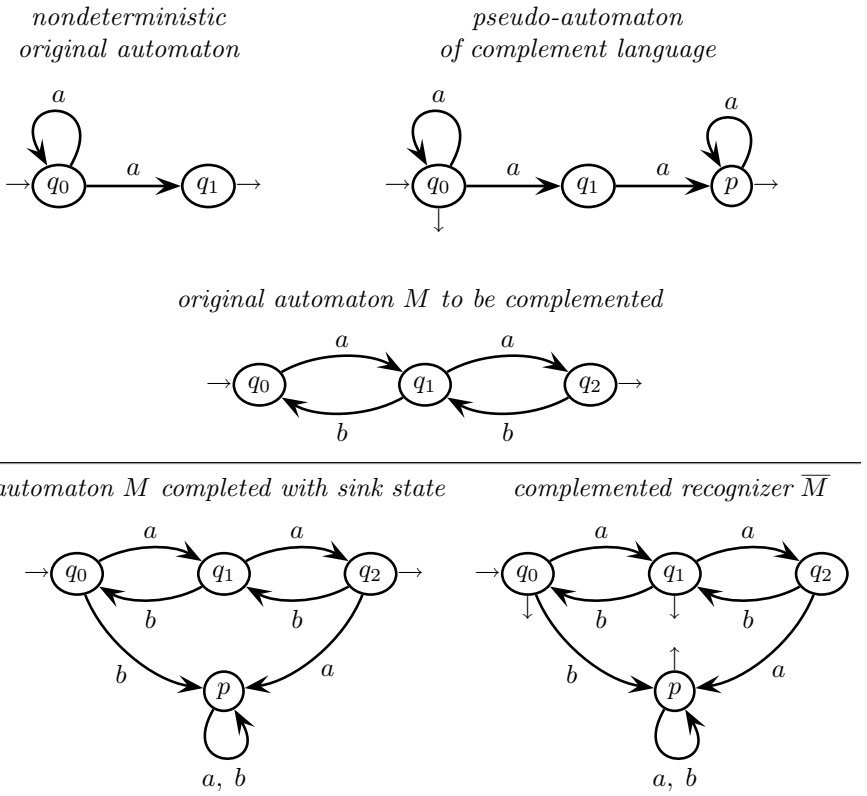


Fig. 3.30 Construction of the recognizer M of complement language $\neg L\,(M)$ (Example 3.65)

3.10.1 Product of Automata

A frequently used technique consists in simulating two (or more) machines by a single one that has as state set the cartesian product of the two state sets. We present this technique for the case of the recognizer of the intersection of two regular languages.

Incidentally, the closure proof of family *REG* under intersection, based on the De Morgan identity (p. 188), already gives a procedure for recognizing intersection: given two finite deterministic recognizers, first construct the recognizers of the complement languages, then the recognizer of their union (by the Thompson method on p. 148). From the latter (and after determinizing if needed), construct the complement machine, which is the desired result.

More directly, the intersection of two regular languages is accepted by the cartesian product of the given machines M' and M''. We assume the two machines to be free from spontaneous moves, yet not necessarily deterministic.

The *product machine* M has state set $Q' \times Q''$, i.e., the cartesian product of the two state sets. This means that each state is a pair $\langle q',\ q'' \rangle$, where the left (right)

component is a state of the first (second) machine. For such a pair or product state $\langle q', q'' \rangle$, we construct the outgoing arc:

$$\langle q', q'' \rangle \xrightarrow{a} \langle r', r'' \rangle$$

if, and only if, there exist the arcs $q' \xrightarrow{a} r'$ in M' and $q'' \xrightarrow{a} r''$ in M''. In other words, such an arc exists in M if, and only if, its projection on the left (respectively right) component exists in M' (respectively in M'').

The initial state set I of the product machine M is the product $I = I' \times I''$ of the initial state sets of the component machines. The final state set of M is the product of the final state sets of the components, i.e., $F = F' \times F''$.

To justify the construction correctness, consider any string x in the intersection language. Since string x is accepted by a computation of machine M' as well as by one of machine M'', it is also accepted by the one of machine M that traverses the state pairs respectively traversed by the two computations. Conversely, if string x is not in the intersection, at least one of the computations by M' or M'' does not reach a final state. Hence the computation of M does not reach a final state either, because the latter are pairs of final states.

Example 3.66 (*intersection and product machine—Sakarovitch* [12]) The recognizer M of the strings that contain as substrings both digrams $a\,b$ and $b\,a$ is naturally specified through the intersection of languages L' and L'':

$$L' = (a \mid b)^* \, a\,b \, (a \mid b)^* \quad L'' = (a \mid b)^* \, b\,a \, (a \mid b)^*$$

The cartesian product M of the recognizers M' and M'' of the component languages is in Fig. 3.31. As usual, the state pairs of the cartesian product that are not accessible from the initial state pair can be discarded. □

The cartesian product method can be exploited for operations different from intersection, such as union or exclusive union. In the case of union, it would be easy to modify the product machine construction to accept a string if at least one (instead of both as in the intersection) component machine accepts it. This would give us an alternative construction for the recognizer of the union of two languages. But the size of the product machine is typically much larger than that of the machine obtained through the Thompson method: in the worst case, its number of states is equal to the product of the two machine sizes, instead of the sum.

3.10.1.1 Extended and Restricted Regular Expression

A *regular expression* is *extended* if it uses other operators beyond the basic ones (union concatenation star and cross): complement, intersection and set difference. For instance, the strings that contain one or multiple occurrences of the digram $a\,a$ as a substring and do not end with the digram $b\,b$ are defined by the extended r.e.:

$$\big((a \mid b)^* \, a\,a \, (a \mid b)^*\big) \cap \neg \big((a \mid b)^* \, b\,b\big)$$

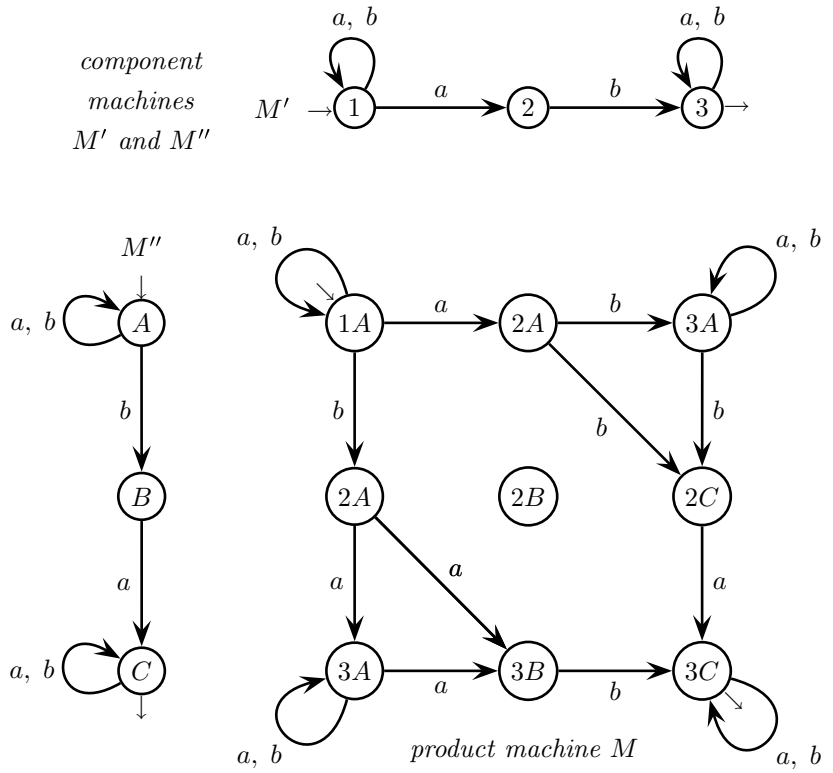


Fig. 3.31 Recognizers of component languages (top and left) and the product machine (bottom right) that recognizes their intersection (Example 3.66). State $2B$ can be removed

The next realistic example (Example 3.67) shows the practicality of using extended regular expressions for a greater expressivity.

Example 3.67 (*identifier*) Suppose that the valid identifiers, e.g., variable names, may contain letters $a \ldots z$, digits $0 \ldots 9$ (not in the first position) and one or more dashes "_". Dashes may not occur in the first nor in the last position, and consecutive dashes are forbidden. A sentence of this language is after_the_2nd_test.

The next extended regular expression prescribes that (*i*) every string starts with a letter, (*ii*) it does not contain consecutive dashes, and (*iii*) it does not end with a dash:

$$\underbrace{(a \ldots z)^+ \, (a \ldots z \mid 0 \ldots 9 \mid _)^*}_{(i)} \cap$$
$$\underbrace{\neg \left((a \ldots z \mid 0 \ldots 9 \mid _)^* \, __ \, (a \ldots z \mid 0 \ldots 9 \mid _)^* \right)}_{(ii)} \cap$$

$$\underbrace{\neg\,\big((a\,\ldots\,z\;\mid\;0\,\ldots\,9\;\mid\;_)^*\,_\big)}_{(iii)}$$

The r.e. above is extended with intersection and complement. □

When a language is specified through an extended r.e., of course we can construct its recognizer by applying the complement and cartesian product methods. Thus an equivalent non-extended r.e. can be obtained, if desired.

3.10.1.2 Star-Free Language

We know that adding complement and intersection operators to an r.e. does not enlarge the family of regular languages, because such operators can be eliminated and replaced by basic ones. On the other hand, removing star from the permitted operators causes a loss of generative capacity. The language family shrinks into a subfamily of family *REG*, variously named as *aperiodic* or *star-free* or *non-counting*. Since such a family is rarely considered in the realm of compilation, a short discussion suffices.

Consider the operator set comprising union, concatenation, intersection and complement. Starting with the terminal elements of the alphabet and the empty set $\emptyset$,[18] we can write a so-called *star-free* r.e. by using only these operators. Notice that having intersection and complement is essential to compensate somehow for the loss of star (and cross), otherwise just finite languages would be definable through star-free regular expressions.

A language is called *star-free* if there exists a star-free r.e. that defines it. First, observe that the universal language over an alphabet Σ is star-free, as it is the complement of the empty set $\emptyset$; i.e., it is defined by the star-free r.e. $\neg\,\emptyset = \Sigma^*$. Second, a useful subclass of star-free languages has already been studied without even knowing the term: indeed the local languages on p. 150 can be defined without star or cross. Recall that a local language is precisely defined by three local sets: initials, finals and permitted (or forbidden) digrams. Its specification can be directly mapped onto the intersection of three star-free languages, as follows.

Example 3.68 (*star-free r.e. of local and non-local languages*) The sentences of the local language $(\,a\,b\,c\,)^+$ (Example 3.27 on p. 150) start with a letter a, end with a letter c and do not contain any digram from $\{\,a\,a,\;a\,c,\;b\,a,\;b\,b,\;c\,b,\;c\,c\,\}$. The language is therefore specified by the star-free r.e.:

$$(\,a\,\neg\emptyset\,)\;\cap\;(\,\neg\emptyset\,c\,)\;\cap\;\Big(\neg\big(\neg\emptyset\,(\,a\,a\;\mid\;a\,c\;\mid\;b\,a\;\mid\;b\,b\;\mid\;c\,b\;\mid\;c\,c\,)\,\neg\emptyset\big)\Big)$$

[18] For the star-free family, we allow in an r.e. the empty set $\emptyset$ as a new metasymbol. This is necessary, e.g., to define the universal language as $\neg\,\emptyset$ (complement of $\emptyset$), since we may not use the Kleene star expression Σ^*.

Here is another example. Language $L_2 = a^* \, b \, a^*$ is star-free because it can be converted into the equivalent star-free r.e.:

$$\overbrace{\neg \, (\neg \emptyset \, b \, \neg \emptyset)}^{a^*} \; \overbrace{b}^{b} \; \overbrace{\neg \, (\neg \emptyset \, b \, \neg \emptyset)}^{a^*}$$

On the other hand, this language is not local as its local sets do not suffice for excluding spurious strings. Among the strings that start and end with a letter a or b and may contain digrams $\{ \, a \, a, \;\; a \, b, \;\; b \, a \, \}$, as prescribed by language L_2, unluckily we find string, say, $a \, b \, a \, b$, not belonging to L_2. □

The family of star-free languages is strictly included in the regular language family. In particular, it excludes the languages characterized by certain counting properties that justify the other name of "non-counting" often given to the family. An example is the regular language:

$$\left\{ \, x \in \{ \, a \;\; | \;\; b \, \}^* \;\; | \;\; \; | \, x \, |_a \text{ is an even number} \, \right\}$$

This language is accepted by a machine that has a circuit of length two in its graph, i.e., a modulo-2 counter (or flip-flop) of the letters a encountered. Such a language cannot be defined with an r.e without using star or cross.

An empirical observation is that, in the panorama of artificial and human languages, the operation of counting letters or substrings modulo some integer constant (in the intuitive sense of the previous example) is rarely needed, if ever. In other words, the string classification based on the congruence classes modulo some integer is usually uncorrelated with their being valid sentences or not. For reasons that may have to do with the organization of the human mind or perhaps with the robustness of noisy communication, none of the existing technical languages discriminates sentences from illegal strings on the basis of modulo counting properties. Indeed, it would be strange if a computer program were considered valid depending on the number of its instructions being, say, a multiple of three or not!

Therefore, in principle it would be enough to deal with the subfamily of aperiodic or non-counting regular languages, when modeling compilation and artificial languages. Yet, on one side star-free r.e. is often less readable than the basic ones, and on the other side in different fields of computer science, counting objects is of the uttermost importance. For instance, a most common digital component is the flip-flop or modulo-2 counter, which recognizes the language $(11)^*$, obviously not a star-free one.[19]

[19] For the theory of star-free languages we refer to McNaughton and Papert [18].

3.11 Summary of the Relations Between Regular Expressions, Grammars and Automata

On leaving the topic of regular languages and finite automata, it is convenient to recapitulate the relations and transformations between the various formal models associated with this language family.

Figure 3.32 represents, by means of a flow graph, the conversion methods back and forth from regular expressions and automata of different types. For instance, we read that algorithm *GMY*, explained on p. 159, converts an r.e. into an automaton devoid of spontaneous moves. In a similar way, Fig. 3.33 represents the direct correspondence between right-linear grammars and finite automata. The spontaneous transitions, i.e., ε-moves, correspond to the copy rules of grammars; both can be eliminated (as seen on p. 142 and p. 67). The relations between left-linear grammars and automata are not listed, because they are essentially similar to the right-linear case, thanks

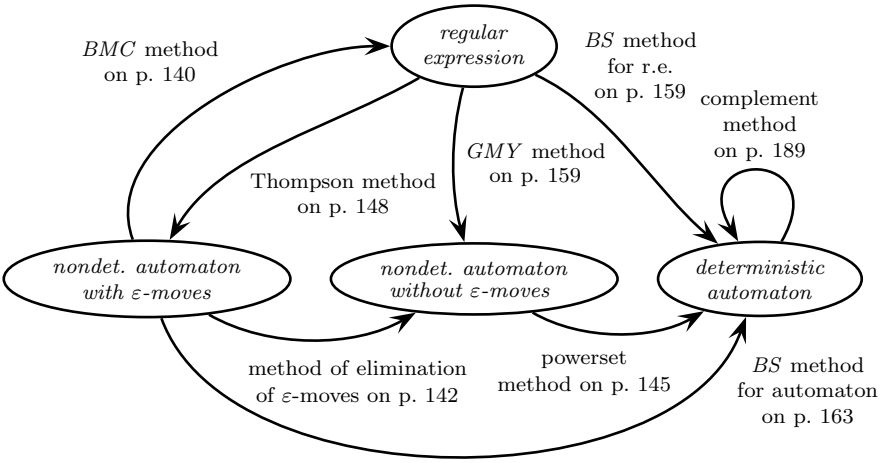


Fig. 3.32 Conversion methods between r.e. and (nondeterministic and deterministic) finite automata, with and without ε-moves

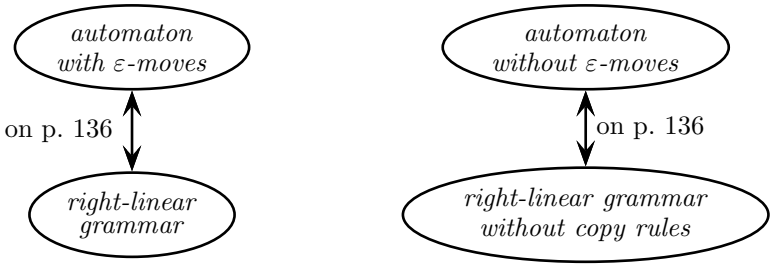


Fig. 3.33 Correspondence between right-linear grammars and finite automata, in general nondeterministic

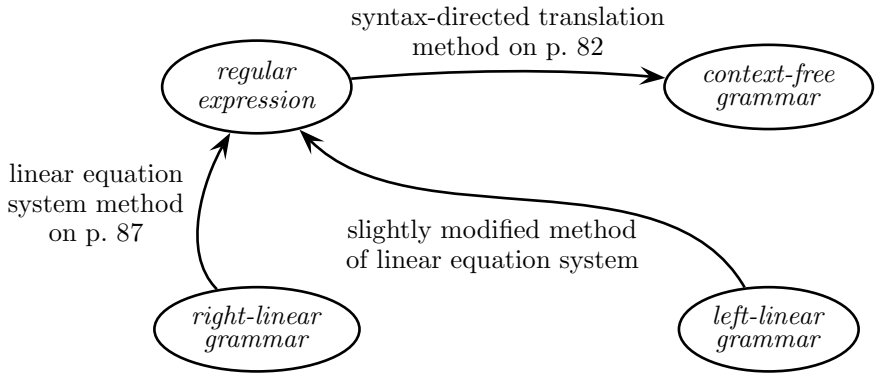


Fig. 3.34 Correspondence between regular expressions and grammars

to the left/right duality of grammar rules and the arrow-reversing transformation of automaton moves.

At last, Fig. 3.34 lists the relations between regular expressions and grammars. The three figures give evidence to the equivalence of the three models used for regular languages: regular expressions, unilinear grammars and finite automata. Finally, we recall that regular languages are a very restricted subset of the context-free ones, which are indispensable for defining artificial languages.

References

1. Bovet D, Crescenzi P (1994) Introduction to the theory of complexity. Prentice-Hall, Englewood Cliffs
2. Floyd RW, Beigel R (1994) The language of machines: an introduction to computability and formal languages. Computer Science Press, New York
3. Hopcroft J, Ullman J (1979) Introduction to automata theory, languages, and computation. Addison-Wesley, Massachusetts
4. Kozen D (2007) Theory of computation. Springer, London
5. McNaughton R (1982) Elementary computability, formal languages and automata. Prentice-Hall, Englewood Cliffs
6. Rich E (2008) Automata, computability, and complexity: theory and applications. Pearson Education, New York
7. Salomaa A (1973) Formal languages. Academic Press, New York
8. Hopcroft J, Ullman J (1969) Formal languages and their relation to automata. Addison-Wesley, Massachusetts
9. Harrison M (1978) Introduction to formal language theory. Addison Wesley, Massachusetts
10. Shallit J (2008) A second course in formal languages and automata theory, 1st edn. Cambridge University Press, New York
11. Rozenberg G, Salomaa A (eds) (1997) Handbook of formal languages, vol. 1: word, language, grammar. Springer, New York

12. Sakarovitch J (2009) Elements of automata theory. Cambridge University Press, Cambridge
13. Watson B (1994) A taxonomy of finite automata minimization algorithms, Report. Department of Mathematics and Computer Science, Eindhoven; Eindhoven University of Technology, Eindhoven, The Netherlands
14. Thompson K (1968) Regular expression search algorithm. Commun ACM 11(6):419–422
15. Berry G, Sethi R (1986) From regular expressions to deterministic automata. Theor Comput Sci 48(1):117–126
16. Berstel J, Pin JE (1996) Local languages and the Berry-Sethi algorithm. Theor Comput Sci 155(2):439–446
17. Frisch A, Cardelli L (2004) Greedy regular expression matching. In: Díaz J, Karhumäki J, Lepistö A, Sannella D (eds) ICALP. Springer, Berlin, pp 618–629
18. McNaughton R, Papert S (1971) Counter-free automata. The MIT Press, Cambridge



Pushdown Automata and Parsing

4

4.1 Introduction

The algorithms for recognizing whether a string is a legal sentence require more memory and time resources for context-free languages than for the regular ones. This chapter presents several algorithms, first as abstract automata with a pushdown memory stack, then as parsing[1] (or syntax analysis) procedures that produce the syntax tree of a sentence. We recall that for regular languages defined by unilinear grammars, parsing has little interest because their syntax structure is predeterminate (left or right-linear), and for languages defined by ambiguous regular expressions we have presented a parsing algorithm in Chap. 3.

Similarly to unilinear grammars, it is possible to match also the rules of a context-free grammar to the moves of an automaton which is not a pure finite-state machine since it also has a pushdown stack memory. In contrast to the finite-state case, such a pushdown machine can be made deterministic only for a subfamily of context-free languages known as *deterministic*, briefly *DET*. Moreover, the presence of two different memory devices, i.e., finite state and stack, introduces a variety of functioning modes and makes the theoretical study of such machines more involved.

The chapter starts with the essentials of pushdown automata and compares the general and deterministic cases in terms of expressivity and of closure under language operations. Two simplified types of deterministic machine are examined: the simple deterministic model and the input-driven one.

The chapter continues with several useful parsing algorithms, which altogether meet the requirements of compiler writing. It is traditional to classify such algorithms on two dimensions: deterministic versus nondeterministic, and pushdown-based versus tabular. We carefully develop the deterministic pushdown-based methods, which

[1] From Latin *pars, partis*, in the sense of dividing a sentence into its parts or constituents.

© Springer Nature Switzerland AG 2019

S. Crespi Reghizzi et al., *Formal Languages and Compilation*,
Texts in Computer Science, https://doi.org/10.1007/978-3-030-04879-2_4

are further divided into the classes of bottom-up (or shift-reduce) and top-down (or predictive) parsers, depending on the construction order of the syntax tree. The leading bottom-up $LR\,(k)$ method is first presented, then the top-down method based on the $LL\,(k)$ grammars, and lastly the bottom-up operator-precedence method, which is motivated by its suitability for parallel execution.

We show how to adjust a grammar to deterministic parsing and how to make a deterministic parser more selective by using a longer prospection. Subsequently we return to theory and we compare the language families that correspond to the various parsers considered.

Then we address the relevant problem of how to improve parsing performance on multiprocessor (multi-core) computers. To enable a parallel execution of parsing steps, the parser must be able to make decisions by examining a limited part of the source text, so that several workers can independently analyze different text parts. We present the parallel parsing algorithm *PAPAGENO* based on the efficient sequential operator-precedence parser (for unextended *BNF*), and we analyze its complexity. A discussion on the deterministic parsing methods ends this part.

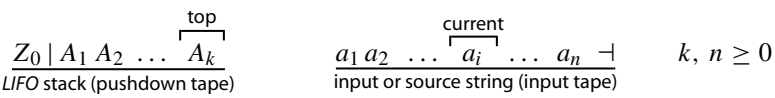
All the deterministic methods impose specific restrictions on the form of grammar rules. The last method considered, called Earley parser, is fully general and is able to cope with ambiguous and nondeterministic grammars, but it pays the price of nonlinear time complexity. It uses tables instead of a stack.

The chapter ends with a brief discussion on syntactic errors, error diagnosis and recovery in parsing, and incremental parsing.

It is important to observe that our presentation of classical parsing methods is novel and much improved over orthodox ones. Such methods, i.e., $LR\,(k)$, $LL\,(k)$ and Earley, have a long individual history behind them, which has left much tedious and irrelevant diversity that we sweep away in our unified presentation. Our approach is based on representing the grammar as a network of finite automata. Automata networks are a natural representation for grammars, especially when regular expressions occur in the rule right parts, i.e., for the *extended* context-free form or *EBNF* (p. 100) frequently used in the language reference manuals. Accordingly, the sequential parsers we present take such extended context-free grammars. Starting from the bottom-up deterministic methods, we are able to incrementally develop the top-down and the general parsing algorithms, respectively, as a specialization and a generalization of the bottom-up method.

4.2 Pushdown Automaton

Every compiler includes a recognition algorithm, which is essentially a finite automaton enriched with an auxiliary memory organized as a pushdown or *LIFO* stack of unbounded capacity. The stack stores the symbols $A_1 \ldots A_k$:



The input or source string $a_1 \ldots a_n$ is often delimited on the right by an end-marker $\dashv$. The following three operations apply to a stack:

pushing	operation *push* (B) inserts symbol B on top, i.e., onto the right of symbol A_k; several push operations *push* (B_1), *push* (B_2), ..., *push* (B_n) can be combined in one command *push* $(B_1\ B_2 \ldots B_m)$ that inserts $B_1\ B_2 \ldots B_m$, with B_m on top $(m \geq 1)$
popping	operation *pop*, if the stack is not empty, removes the top symbol A_k $(k \geq 1)$
emptiness test	predicate *empty* is true if, and only if, $k = 0$ (no symbols in the stack)

Sometimes it is convenient to imagine that a special symbol is painted on the stack bottom, i.e., on the left of symbol A_1. Such a symbol is denoted Z_0 and is termed *bottom*: it can be read, that is, its presence on the stack top can be checked, yet it may not be pushed or popped. The presence of Z_0 on the stack top means that the stack is empty.

The machine reads the source characters $a_1 \ldots a_n$ from left to right by means of a reading head. The character a_i under the reading head is termed *current* $(1 \leq i \leq n)$. At each instant the machine *configuration* is specified by the remaining portion of the input string left to be read, the current state, and the stack contents. With a move an automaton can:

- read the current character, and either shift the reading head or perform a move (called spontaneous) without reading
- read and pop the top symbol, or read the bottom Z_0 if the stack is empty
- compute the next state from the current values of the state, character and top-of-stack symbol
- push one or more symbols (or even none) onto the stack

4.2.1 From Grammar to Pushdown Automaton

We show how the grammar rules can be interpreted as instructions of a nondeterministic pushdown machine that recognizes the language. The machine is so simple as to work without any internal state, and it uses only the stack for memory. Intuitively, the machine operation is *predictive* or goal-oriented: the stack serves as an agenda of predicted future actions. The stack stores nonterminal and terminal grammar symbols. If the stack contains symbols $A_1 \ldots A_k$ (from bottom to top), the machine first executes the action prescribed by A_k, then the action for A_{k-1}, and so on until the last action for A_1. The action for A_k has to recognize if, starting from the current character a_i, the source string contains a substring w that derives from A_k. If so, the action will eventually shift the reading head of $\mid w \mid$ positions. Naturally enough, the goal may recursively spawn subgoals if, for recognizing the derivation from A_k, it is

Table 4.1 Correspondence between grammar rules and moves of a nondeterministic pushdown machine with one state. Symbols *cc* and *top* represent the current input character and the stack top symbol, respectively

#	Grammar rule	Automaton move	Comment
1	$A \to B\, A_1 \ldots A_m$ with $m \geq 0$	If $top = A$ then *pop*; *push* $(A_m \ldots A_1 B)$	To recognize A, orderly recognize $B\, A_1 \ldots A_m$
2	$A \to b\, A_1 \ldots A_m$ with $m \geq 0$	If $cc = b$ and $top = A$ then *pop*; *push* $(A_m \ldots A_1)$; *shift* reading head	Character b is expected as next and is read, so it remains to orderly recognize $A_1 \ldots A_m$
3	$A \to \varepsilon$	If $top = A$ then *pop*	The empty string deriving from A is recognized
4	For any character $b \in \Sigma$	if $cc = b$ and $top = b$ then *pop*; *shift* reading head	Character b is expected as next and is read
5	Acceptance condition	If $cc = \dashv$ and the stack is *empty* then accept; *halt*	The input string has been entirely scanned, and the agenda has no goals

necessary to recognize other nonterminal symbols. The initial goal is the grammar axiom: the machine task is to recognize if the source string derives from the axiom.

Algorithm 4.1 (*from grammar to nondeterministic one-state pushdown machine*) Given a grammar $G = (V, \Sigma, P, S)$, Table 4.1 explicates the correspondence between rules and moves. Letter b denotes a terminal, letters A and B denote nonterminals, and letter A_i can be any symbol. The rule form shapes the move. For rules of form 2, the right-hand side starts with a terminal and the move is triggered on reading it. Instead, rules of form 1 and 3 give rise to spontaneous moves that do not check the current character. Move 4 checks that a terminal surfacing on the stack top (having been previously pushed by a move of type 1 or 2) matches the current character. Lastly, move 5 accepts the string if the stack is empty upon reading the end-marker.

Initially the stack contains only the bottom symbol Z_0 and the grammar axiom S, and the reading head is on the first input character. At each step, the automaton chooses a move (not deterministically), which is defined in the current configuration, and executes it. The machine recognizes the string if there exists a computation that ends with move 5. Accordingly, we say that this machine model recognizes a sentence by *empty stack*. □

Surprisingly enough, this automaton never changes state and the stack is the only memory it has. Later on, we will be obliged to introduce states in order to make deterministic the machine behavior.

Table 4.2 Pushdown machine moves for the grammar rules of Example 4.2

#	Grammar rule	Automaton move
1	$S \to a\ S$	If $cc = a$ and $top = S$ then *pop*; *push* (S); *shift*
2	$S \to A$	If $top = S$ then *pop*; *push* (A)
3	$A \to a\ A\ b$	If $cc = a$ and $top = A$ then *pop*; *push* $(b\ A)$; *shift*
4	$A \to a\ b$	If $cc = a$ and $top = A$ then *pop*; *push* (b); *shift*
5		If $cc = b$ and $top = b$ then *pop*; *shift*
6		If $cc = \dashv$ and the stack is *empty* then accept; *halt*

Example 4.2 (*from a grammar to a pushdown automaton*) The moves of the recognizer of the language L below:

$$L = \left\{ a^n\ b^m \mid \quad n \geq m \geq 1 \right\}$$

are listed in Table 4.2 next to the grammar rules. The choice between moves 1 and 2 is not deterministic, since move 2 may be taken also when character a is current; similarly for choosing between moves 3 and 4. □

It is easy to see that this machine accepts a string if, and only if, the grammar generates it. In fact, for each accepting computation there exists a corresponding derivation and conversely; in other words, the automaton simulates the leftmost derivations of the grammar. For instance, the derivation:

$$S \Rightarrow A \Rightarrow a\ A\ b \Rightarrow a\ a\ b\ b$$

mirrors the successful trace in Fig. 4.1 (left). Yet Algorithm 4.1 does not have any apriori information about which of the possible derivations will succeed, if any at all, and has to explore all the possibilities, including the computations that end in error as the one traced (right). Moreover, the source string is accepted by different computations if, and only if, it is generated by different left derivations, i.e., if it is ambiguous for the grammar.

With some thought, we may see that the mapping of Table 4.1 on p. 202 is one to one, thus it can be applied the other way round, to transform the moves of a pushdown machine (of the model considered) into the rules of an equivalent grammar. This remark allows us to state an important theoretical fact, which links grammars and pushdown automata.

Property 4.3 (CF languages and one-state pushdown automata) *The family of context-free languages coincides with the family of the languages accepted by a*

stack	input x
$Z_0 \mid S$	$a\,a\,b\,b\,\dashv$
$\cdots \mid A$	$a\,a\,b\,b\,\dashv$
$\cdots \mid b\,A$	$a\,b\,b\,\dashv$
$\cdots \mid b\,b$	$b\,b\,\dashv$
$\cdots \mid b$	$b\,\dashv$
$Z_0 \mid$	$\dashv$

stack	input x
$Z_0 \mid S$	$a\,a\,b\,b\,\dashv$
$\cdots \mid S$	$a\,b\,b\,\dashv$
$\cdots \mid S$	$b\,b\,\dashv$
$Z_0 \mid A$	$b\,b\,\dashv$
error – cannot move	

Fig. 4.1 Two computations: accepting (left) and rejecting by error (right)

nondeterministic pushdown machine that recognizes by empty stack and has only one state. □

We stress that the mapping above from pushdown automaton to grammar does not work when the automaton has two or more states; other methods will be developed for that case.

It may appear that in little space and without effort we have already reached the objective of the chapter: to obtain a procedure for building the recognizer of a language defined by a grammar. Unfortunately, the automaton is nondeterministic and in the worst case it is forced to explore all the computation paths, with a time complexity non-polynomial with respect to the source string length; more efficient algorithms are wanted for practical application.

4.2.1.1 Computational Complexity of Pushdown Automata

We compute a worst-case upper bound to the number of steps that are needed to recognize a string with the previous pushdown machine. For simplicity, we consider a grammar G in the Greibach normal form (p. 78), which features rules starting with a terminal and not containing other terminals. Therefore, the machine constructed by Algorithm 4.1 is free from spontaneous moves (types 1 and 3 of Table 4.1 on p. 202) and it never pushes a terminal character onto the stack.

For a string x of length n, the derivation $S \overset{+}{\Rightarrow} x$ has exactly n steps, if it exists. The same number of moves is performed by the automaton to recognize string x. Let K be the maximum of the number of alternative rules $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$, for any nonterminal A. At each step of a leftmost derivation, the leftmost nonterminal, say A, is rewritten by choosing one out of $k \leq K$ alternatives. It follows that the number of possible derivations of length n is at most K^n. Since in the worst case the algorithm is forced to compute all the derivations before finding the accepting one or declaring failure, the time complexity for recognition is exponential in n.

However, this result is overly pessimistic. At the end of this chapter, a clever algorithm for string recognition in polynomial time will be described, which uses a linked list data structure instead of a LIFO stack.

4.2.2 Definition of Pushdown Automaton

We are going to define several pushdown machine models. In order to expedite the presentation, we trust the reader to adapt to the present context the analogous concepts already seen for finite automata. A pushdown automaton M is defined by seven items:

Q	a (non-empty) finite *set of states* of the control unit
Σ	an *input alphabet*
Γ	a *stack (or memory) alphabet*
δ	a *transition function*
$q_0 \in Q$	an *initial state*
$Z_0 \in \Gamma$	an *initial stack symbol*
$F \subseteq Q$	a *set of final states*

In general such a machine is nondeterministic. The domain and range of the transition function δ are expressed by Cartesian products:

domain	$Q \times (\Sigma \cup \{ \varepsilon \}) \times \Gamma$
range	$\wp (Q \times \Gamma^*)$, i.e., the *powerset* of the set $Q \times \Gamma^*$

The moves, i.e., the values of function δ, are classified as *reading* and *spontaneous*, as follows:

reading move

$$\delta (q,\ a,\ A) = \{ (p_1,\ \gamma_1),\ (p_2,\ \gamma_2),\ \ldots,\ (p_n,\ \gamma_n) \}$$

with $q,\ a,\ A$ resp. in $Q,\ \Sigma,\ \Gamma$, and $p_i,\ \gamma_i$ resp. in $Q,\ \Gamma^*$ for $1 \leq i \leq n$.
The machine, in state q with A on stack top, reads character a and enters one of the states p_i after performing these operations: *pop* and *push* (γ_i).

spontaneous move

$$\delta (q,\ \varepsilon,\ A) = \{ (p_1,\ \gamma_1),\ (p_2,\ \gamma_2),\ \ldots,\ (p_n,\ \gamma_n) \}$$

with the same stipulations as before.
The machine, in state q with A on stack top and without reading an input character, enters one of the states p_i after performing these operations: *pop* and *push* (γ_i).

Notes: The choice of the i-th action out of n possibilities is not deterministic; the reading head automatically shifts forward on input; the top symbol is always popped; and the string pushed onto the stack may be empty. Although any move performs a pop that erases the top symbol, the same symbol can be pushed anew by the same move, if the computation needs to keep it on stack. Thus, the move $\delta\,(q,\ a,\ A) = \{\,(p, \mathrm{A})\,\}$ checks the presence of A on stack top and lets the stack as it is.

From the definition, it is clear that the machine behavior can be nondeterministic for two causes, to be carefully studied later: (*i*) the range of the transition function comprises a set of alternative actions and (*ii*) the machine may have the alternative between a reading and a spontaneous move.

We say that a pushdown automaton free from spontaneous moves has the *real-time property*.

Short Notation

Certain patterns of moves which frequently occur can be conveniently shortened with the following abbreviations. If, for all stack symbols in $A \in \Gamma$ (including the initial stack symbol Z_0), the move $\delta\,(q,\ a,\ A) = \{\,(p,\ A\,\gamma)\,\}$ is defined, we may represent all such moves by the new notation:

$$\delta\,(q,\ a,\ \varepsilon) = \{\,(p,\ \gamma)\,\}$$

and in particular by $\delta\,(q,\ a,\ \varepsilon) = \{\,(p,\ \varepsilon)\,\}$ if γ is the empty string. In the latter case, the only effect of the move is to change state, and the move can be simply represented by an arc $q \xrightarrow{a} p$ as in a finite automaton.

Thus, having the empty string as third argument of δ indicates that the move is performed whatever the top symbol, which is left untouched. The meaning of a spontaneous move of the form $\delta\,(q,\ \varepsilon,\ \varepsilon) = \{\,(p,\ \gamma)\,\}$ is similar.

The *instantaneous configuration* of a machine M is a 3-tuple:

$$(q,\ y,\ \eta) \in Q \times \Sigma^* \times \Gamma^+$$

which specifies:

- q the current state
- y the rest (suffix) of the source string x, to be read
- η the stack content

The *initial* configuration of machine M is $(q_0,\ x,\ Z_0)$ or $(q_0,\ x\ \dashv,\ Z_0)$, if the end-marker is there.

By applying a move, a transition from a configuration to another occurs, to be denoted as $(q,\ y,\ \eta) \mapsto (p,\ z,\ \lambda)$. A computation is a chain of zero or more transitions, denoted by $\overset{*}{\mapsto}$. As customary, a cross instead of a star, i.e., $\overset{+}{\mapsto}$, denotes a computation with at least one transition.

Depending on the move, the following transitions are possible:

current conf.	*next conf.*	*applied move*	*move type*
$\Big(q,\ a\,z,\ \eta\,A \Big)$	$(\ p,\ z,\ \eta\,\gamma\ )$	$\delta\,(q,\ a,\ A) = \{\ (p,\ \gamma),\ \dots\ \}$	reading
$\Big(q,\ a\,z,\ \eta\,A \Big)$	$(\ p,\ a\,z,\ \eta\,\gamma\ )$	$\delta\,(q,\ \varepsilon,\ A) = \{\ (p,\ \gamma),\ \dots\ \}$	spontaneous

A string x is *recognized* (or accepted) by *final state* if there exists a computation that entirely reads the string and terminates in a final state:

$$(\ q_0,\ x,\ Z_0\ ) \overset{*}{\mapsto} (\ q,\ \varepsilon,\ \lambda\ ) \qquad q \text{ is a final state and } \lambda \in \Gamma^*$$

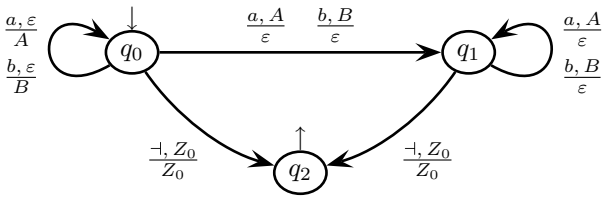
The language recognized by machine M is the set of all accepted strings.

Notice that when the machine recognizes and halts, the stack contains some string λ not further specified, since the recognition modality is by final state; in particular, string λ is not necessarily empty.

4.2.2.1 State-Transition Diagram for Pushdown Automata

The transition function of a pushdown automaton can be visualized as a state-transition graph, although its readability is somewhat lessened by the need to specify stack operations. This is shown in the next example.

Example 4.4 (*language and automaton of palindromes*) The language $L = \left\{\ u\,u^R \mid\ \ u \in \{\ a,\ b\ \}^*\ \right\}$ of the even-length palindromes (p. 32) is accepted by final state by the pushdown recognizer below:



The stack alphabet has three symbols: the bottom Z_0, and the symbols A and B indicating that a character a or b was read, respectively. For instance, arc $q_0 \xrightarrow{\frac{a,\,\varepsilon}{A}} q_0$ denotes any reading move $(q_0,\ X\,A) \in \delta\,(q_0,\ a,\ X)$, with $X \in \Gamma$.

The machine behaves nondeterministically in the state q_0 on reading an input character a with symbol A on stack top, as it may stay in q_0 and push A, or it may go to state q_1, pop A and push nothing; similarly with b and B.

stack	input x	state
$\boxed{Z_0\vert}$	$a\,a\,\dashv$	q_0
$\boxed{\cdots\vert\,A}$	$a\,\dashv$	q_0
$\boxed{\cdots\vert\,A\,A}$	$\dashv$	q_0

failure: no move is defined,
i.e., $\delta\,(q_0,\,\dashv,\,A) = \emptyset$

stack	input x	state
$\boxed{Z_0\vert}$	$a\,a\,\dashv$	q_0
$\boxed{\cdots\vert\,A}$	$a\,\dashv$	q_0
$\boxed{Z_0\vert}$	$\dashv$	q_1
$\boxed{Z_0\vert}$	finished	q_2

recognition by final state

Fig. 4.2 Two computations of the automaton of Example 4.4 for the input string $x = a\,a$

In Fig. 4.2 we trace two computations, among others possible, for string $x = a\,a$. Since the computation (right) entirely reads string $a\,a$ and reaches a final state, the string is accepted. Another example is the empty string ε, recognized by the move corresponding to the arc from q_0 to q_2. □

4.2.2.2 Varieties of Pushdown Automata

It is worth noticing that the pushdown machine of the definition differs from the one derived from a grammar through the mapping of Algorithm 4.1 on p. 202 in the following two aspects: to recognize a string, it performs state transitions and checks if the current state is final. These and other differences are discussed next.

Accepting Modes

Two different ways of deciding acceptance when a computation ends have been encountered so far: when the machine enters a final state or when the stack is empty. The former mode by *final state* disregards the stack content, whereas the latter by *empty stack* disregards the current machine state.

The two modes can also be combined into recognition by *final state and empty stack*. A natural question to ask is whether these and other acceptance modes are equivalent.

Property 4.5 (acceptance modes) *For the family of nondeterministic pushdown automata, the three acceptance modes below:*

- *by empty stack*
- *by final state*
- *combined (empty stack and final state)*

have the same capacity with respect to language recognition. □

The statement says that any of the three acceptance modes can be simulated by any other. In fact, if the automaton recognizes by final state, it can be easily modified by adding new states so that it recognizes by empty stack. Simply, when the original machine enters a final state, the second machine enters a new state and it stays there as it empties the stack by spontaneous moves until the bottom symbol pops up.

Vice versa, to convert an automaton recognizing by empty stack to the final state mode, do the following:

- whenever the stack becomes empty, add a new final state f and a move leading to it
- on performing a move to state q, when the stack of the original machine ceases to be empty, let the second machine move from state f to state q

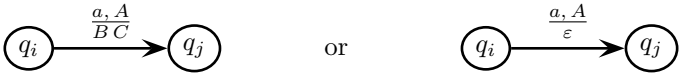
Similar considerations could be made for the third acceptance mode.

4.2.3 One Family for Context-Free Languages and Pushdown Automata

We are going to show that the language accepted by a pushdown machine that uses also the states is context-free. Combined with Property 4.3 (p. 203), that every context-free language can be recognized by a pushdown machine, this leads to the next central property of context-free languages, analogous to the characterization of the regular languages by finite automata.

Property 4.6 (CF and pushdown automata) *The family* CF *of context-free languages coincides with the family of the languages recognized by pushdown automata.* □

Proof Let $L = L(M)$ be the language recognized by a pushdown machine M. To simplify the construction, presented as Algorithm 4.7 below, we assume that machine M has only one final state, it accepts only if the stack is empty, and each transition has either form:



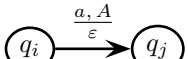
where a is a terminal or the empty string, and A, B and C are stack symbols. Thus a move pushes either two symbols or none. The initial stack symbol and state are Z and q_0, respectively. Symbol Z plays similarly to the stack bottom Z_0, except that it may be popped at the end. In this way the machine accepts by empty stack and final state. It turns out that the assumptions above do not reduce the generality of the machine. □

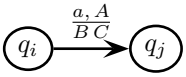
Algorithm 4.7 (*from PDA to grammar*) We construct a grammar G equivalent to machine M. The construction may produce useless nonterminals and rules that later can be removed by cleaning. In the resulting grammar, the axiom is S and all the other nonterminal symbols are formed by a 3-tuple containing two states q_i, q_j and a stack symbol A of M, written as $\langle q_i,\ A,\ q_j \rangle$.

Grammar rules are constructed in such a way that each computation represents a leftmost derivation. The old construction for stateless automata (Algorithm 4.1 on p. 202) creates a nonterminal for each stack symbol. Yet now we have to take care of the states as well. To this end, each stack symbol A is associated with multiple nonterminals marked with two states that have the following meaning. String z derives from nonterminal $\langle q_i,\ A,\ q_j \rangle$ if, and only if, the automaton starting in the state q_i with symbol A on the stack top performs a computation that reads string z, enters state q_j and deletes symbol A from the stack. According to this principle, the grammar rules that rewrite the axiom have the form:

$$S \to \langle q_0,\ Z,\ q_f \rangle$$

where Z is the initial stack symbol, and q_0 and q_f are the initial and final states, respectively. The other grammar rules are obtained as next specified:

1. Moves of the form 

where the two states may coincide and a may be empty, are converted into rule $\langle q_i,\ A,\ q_j \rangle \to a$.
2. Moves of the form 

where the two states may coincide, are converted into the set of rules:

$$\{\ \langle q_i,\ A,\ q_x \rangle \to a \langle q_j,\ C,\ q_y \rangle \langle q_y,\ B,\ q_x \rangle \mid \ \forall \text{ states } q_x \text{ and } q_y \text{ of } M \}$$

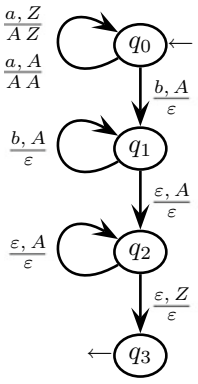
We omit the correctness proof [2] of the construction, and we go to an example.

Example 4.8 (*grammar equivalent to pushdown machine*) The language L:

$$L = \{\ a^n\, b^m \mid \ n > m \geq 1 \}$$

is accepted by the pushdown automaton M in Fig. 4.3 (which does not use the short notation introduced on p. 206). At the beginning, the stack content is Z. The automaton upon reading a character a stores it as a symbol A on the stack. Then, it pops a stack symbol for each character b. At the end it checks that at least one symbol A is left and empties the stack, including the initial symbol Z.

[2] See for instance [1–4].



grammar G

$$\langle q_0,\ Z,\ q_3 \rangle \to a\ \langle q_0,\ A,\ q_2 \rangle\ \langle q_2,\ Z,\ q_3 \rangle$$
$$\langle q_0,\ A,\ q_1 \rangle \to a\ \langle q_0,\ A,\ q_1 \rangle\ \langle q_1,\ A,\ q_1 \rangle$$
$$\langle q_0,\ A,\ q_1 \rangle \to b$$
$$\langle q_0,\ A,\ q_2 \rangle \to a\ \langle q_0,\ A,\ q_1 \rangle\ \langle q_1,\ A,\ q_2 \rangle$$
$$\langle q_0,\ A,\ q_2 \rangle \to a\ \langle q_0,\ A,\ q_2 \rangle\ \langle q_2,\ A,\ q_2 \rangle$$
$$\langle q_0,\ A,\ q_2 \rangle \to \varepsilon$$
$$\langle q_2,\ Z,\ q_3 \rangle \to \varepsilon$$
$$\langle q_1,\ A,\ q_1 \rangle \to b$$
$$\langle q_1,\ A,\ q_2 \rangle \to \varepsilon$$

Fig. 4.3 Pushdown automaton M (left) and equivalent grammar G (right) for Example 4.8. The axiom is $\langle q_0,\ Z,\ q_3 \rangle$

The grammar rules are listed next to the automaton: notice that the axiom is $\langle q_0,\ Z,\ q_3 \rangle$. We do not list the useless rules created by step 2 of the construction above, such as the rule:

$$\langle q_0,\ A,\ q_1 \rangle \to a\ \langle q_0,\ A,\ q_3 \rangle\ \langle q_3,\ A,\ q_1 \rangle$$

that contains the undefined nonterminal $\langle q_0,\ A,\ q_3 \rangle$. To understand the mapping between the two models, it helps to compare the following computation and the leftmost derivation, both of five steps:

$$(q_0,\ a\,a\,b,\ Z) \overset{5}{\mapsto} (q_3,\ \varepsilon,\ \varepsilon) \qquad \langle q_0,\ Z,\ q_3 \rangle \overset{5}{\Rightarrow} a\,a\,b$$

The computation and corresponding derivation tree with numbered steps are in Fig. 4.4. The next properties hold at each step:

- the string prefix read by the machine and the terminal prefix generated by the derivation are identical
- the stack content is the reversal of the string obtained by concatenating the middle symbols (A and Z) of each 3-tuple in the derived string
- any two consecutive 3-tuples in the derived string are chained together by the identity of the states, which are marked by equal arrows as below:

$$\langle q_0,\ Z,\ q_3 \rangle \overset{+}{\Rightarrow} a\,a\,\langle q_0,\ A,\ \overset{\downarrow}{q_1} \rangle \langle \overset{\downarrow}{q_1},\ A,\ \overset{\Downarrow}{q_2} \rangle \langle \overset{\Downarrow}{q_2},\ Z,\ q_3 \rangle$$

Such a stepwise correspondence between transition and derivation ensures that the two models define the same language. □

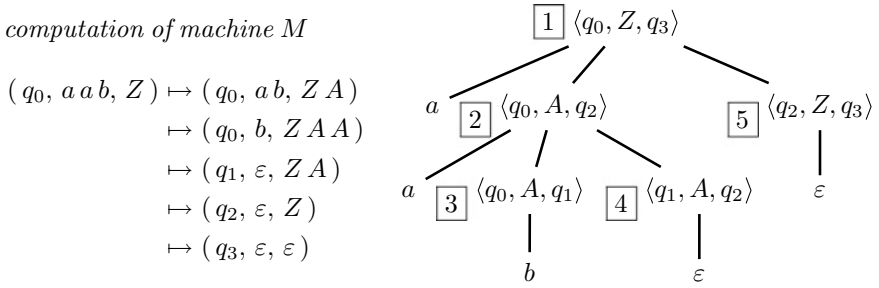


Fig. 4.4 A computation of machine M (left) and the corresponding tree for Example 4.8 (right)

4.2.3.1 Real-Time Pushdown Automata

An automaton works in *real time* if at each step it reads an input character, i.e., if it does not involve any spontaneous moves. This definition applies to finite-state machines and pushdown machines, both deterministic and nondeterministic. In particular, a pushdown automaton has the *real-time property* if the transition function δ, as defined in Sect. 4.2.2 on p. 205, is such that for all the states $q \in Q$ and for all the stack symbols $A \in \Gamma$, the value of $\delta\,(q,\ \varepsilon,\ A)$ is undefined; in other words, the domain of the transition function is $Q \times \Sigma \times \Gamma$ instead of $Q \times (\ \Sigma\ \cup\ \{\ \varepsilon\ \}) \times \Gamma$.

For finite automata, we already know that real-time automata are as powerful as those that have spontaneous moves, since the latter can be always eliminated. Next we show that a similar property holds for nondeterministic pushdown machines.

Property 4.9 (real-time PDA) *For every context-free language there exists a nondeterministic pushdown machine that has the real-time property and recognizes the language.* □

To prove Property 4.9, consider a language L that we may assume, without any loss of generality, to be defined by a grammar G in the real-time normal form (p. 78). Recall that in this form each rule starts with a terminal, thus it has the form 2 of Table 4.1 on p. 202: $A \to b\,A_1\ \ldots\ A_m$, with $m \geq 0$ and $b \in \Sigma$. Therefore empty rules $A \to \varepsilon$ are excluded (we disregard the axiomatic rule $S \to \varepsilon$ needed if the empty string is in L). It follows that the automaton, constructed from such a grammar by Algorithm 4.1, shifts the input head with every move, i.e., it has the real-time property.

Moreover, for any input string x, each computation stops as soon as the automaton reads the last character, after executing exactly $|\ x\ |$ steps (of course the machine may stop at an earlier time and reject the string if the transition function is undefined). Such a behavior of an abstract machine is also known as *online* computation.

Example 4.10 (*real-time pushdown automaton*) The language L below:

$$L = \{\, a^+ b^n c^n \mid \ m, n \geq 1 \,\} \cup \{\, a^m b^+ d^m \mid \ m, n \geq 1 \,\}$$

is defined by the following grammar, where each rule starts with a terminal character:

$$\begin{cases} S \to a\,X \\ X \to a\,X \mid b\,Y\,c \mid b\,c \\ Y \to b\,Y\,c \mid b\,c \end{cases} \qquad \begin{cases} S \to a\,W\,d \\ W \to a\,W\,d \mid b\,Z \mid b \\ Z \to b\,Z \mid b \end{cases}$$

For brevity, instead of listing the moves of the real-time pushdown automaton M that would be built by Algorithm 4.1, we describe in words how a more abstract equivalent real-time machine works. We prefer to write language L as:

$$L = \{\, a^m b^n c^n \mid \ m, n \geq 1 \,\} \cup \{\, a^m b^n d^m \mid \ m, n \geq 1 \,\}$$

The automaton starts reading string a^m and, nondeterministically, either (*i*) it does not push anything onto the stack or (*ii*) it pushes m symbols, say, string A^m. Then, in the case (*i*), machine M reads string b^n and pushes, say, string B^n onto the stack, subsequently it reads string c^m and removes one symbol B for each letter c, and finally it recognizes by empty stack. Similarly, in the case (*ii*), machine M reads string b^n without storing anything onto stack, then it reads string d^m and removes one symbol A for each letter d, and finally it recognizes by empty stack. Clearly, automaton M has the real-time property and is nondeterministic. □

The question whether it is possible to obtain an equivalent deterministic real-time machine is postponed to Sect. 4.3.4.

4.2.4 Intersection of Regular and Context-Free Languages

As an illustration of previous results, we prove the useful property, stated in Chap. 2 in Table 2.9 on p. 95 that the intersection of a context-free and a regular language is context-free. Take a grammar G and a finite automaton A. We explain how to construct a pushdown automaton M that recognizes the intersection language $L(G) \cap L(A)$.

First, starting from grammar G we construct by Algorithm 4.1 on p. 202, a one-state pushdown automaton N that recognizes language $L(G)$ by empty stack. Then we construct a Cartesian product machine M, to simulate the respective computations of the component machines N and A. The construction of M is essentially the same explained for two finite machines (p. 190), with the difference of the presence of a

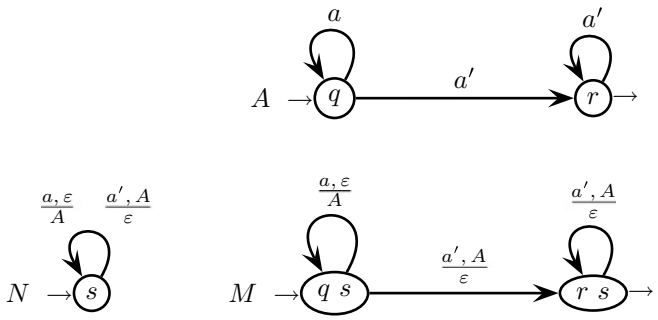


Fig. 4.5 Product machine for the intersection in Example 4.11 of the Dyck language and the regular language $a^* a'^+$ over alphabet $\{ a, a' \}$

stack. The state set of M is the Cartesian product of the state sets of machines N and A, which in this case has the same cardinality as the latter state set. Machine M performs the same operations on the stack as machine N does. For M, recognition is by final state and empty stack. The final states of M are those that contain a final state of the finite machine A. Notice that the product machine is deterministic if both component machines are so.

It is easy to see that a computation of M empties the stack and enters a final state, i.e., it recognizes a string, if, and only if, the string is accepted by empty stack by N and is accepted also by A, which reaches a final state. It follows that machine M accepts the intersection of the two languages.

Example 4.11 (*intersection of a context-free and a regular language*) We want to intersect the Dyck language and the regular language $a^* a'^+$, both over alphabet $\{ a, a' \}$ (as in Example 2.87 on p. 96), thus resulting in the language $\{ a^n a'^n \mid n \geq 1 \}$.

It is straightforward to imagine a one-state pushdown machine N that accepts the Dyck language by empty stack. Machine N, the finite automaton A and the resulting product machine M are depicted in Fig. 4.5. Clearly the resulting machine simulates both component machines step by step. For instance, the arc from $\{ q, s \}$ to $\{ r, s \}$ that reads a' manipulates the stack exactly as automaton N does (it pops symbol A) and simulates the same state transition of automaton A from state q to state r. □

In general, the pushdown automaton N may have more than one state. Then, the product machine state set contains a number of states that is at most the product of the number of states of machines N and A, since, of course, some states of the Cartesian product may be useless.

4.3 Deterministic Pushdown Automata and Languages

It is important to further the study of the deterministic recognizers and corresponding languages, because they are widely adopted in compilers thanks to their computational efficiency. When we observe a pushdown machine (as defined on p. 205), we may find three nondeterministic situations, namely the uncertainty between:

1. reading moves, if, for a state q, a character a and a stack symbol A, the transition function δ has two or more values, i.e., $|\,\delta\,(q,\ a,\ A)\,| > 1$
2. a spontaneous move and a reading move, if both moves $\delta\,(q,\ \varepsilon,\ A)$ and $\delta\,(q,\ a,\ A)$ are defined
3. spontaneous moves, if for some state q and symbol A, the function $\delta\,(q,\ \varepsilon,\ A)$ has two or more values, i.e., $|\,\delta\,(q,\ \varepsilon,\ A)\,| > 1$

If none of the three forms occurs in the transition function, the pushdown machine is *deterministic*. The language recognized by a deterministic pushdown machine is called (context-free) *deterministic*, and the family of such languages is named DET.

Example 4.12 (*forms of nondeterminism*) The one-state recognizer of language $L = \{\, a^n\, b^m \mid \ n \geq m > 0 \,\}$, shown in Table 4.2 on p. 203, is nondeterministic of form 1:

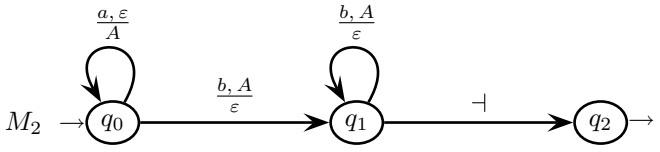
$$\delta\,(q_0,\ a,\ A) = \{\ (q_0,\ b),\ (q_0,\ b\,A)\ \}$$

and also of form 2:

$$\delta\,(q_0,\ \varepsilon,\ S) = \{\ (q_0,\ A)\ \} \qquad\qquad \delta\,(q_0,\ a,\ S) = \{\ (q_0,\ S)\ \}$$

The same language L is accepted by the deterministic machine M_2:

$$M_2 = \big(\ \{\, q_0,\ q_1,\ q_2\,\},\ \{\, a,\ b\,\},\ \{\, A,\ Z_0\,\},\ \delta,\ q_0,\ Z_0,\ \{\, q_2\,\}\ \big)$$



In the state q_0, machine M_2 reads character a and pushes symbol A without reading the stack, i.e., by applying move $\frac{a,\,Z_0}{Z_0\,A}$ or $\frac{a,\,A}{A\,A}$. Upon reading the first character b, it pops an A and moves to state q_1. Then, it pops an A for each further character b. If there is an excess of letters b with respect to letters a, the computation ends in error. Upon reading the terminator $\dashv$, the machine goes to final state q_2 without reading the stack top symbol A or Z_0. □

Table 4.3 Closure properties of language family *DET*, in relation to *CF* and *REG*

Operation	New property	Known property
Reversal	$D^R \notin DET$	$D^R \in CF$
Union	$D_1 \cup D_2 \notin DET,\ D \cup R \in DET$	$D_1 \cup D_2 \in CF$
Complement	$\neg D \in DET$	$\neg L \notin CF$
Intersection	$D \cap R \in DET$	$D_1 \cap D_2 \notin CF$
Concatenation	$D_1 \cdot D_2 \notin DET,\ D \cdot R \in DET$	$D_1 \cdot D_2 \in CF$
Star	$D^* \notin DET$	$D^* \in CF$

Although we were able to find a deterministic pushdown machine for the language of the example, this is impossible for other context-free languages, in other words, the family *DET* is a proper subfamily of *CF*, as we will see.

4.3.1 Closure Properties of Deterministic Languages

Deterministic languages are a proper subclass of context-free languages and have specific properties. Starting from the known properties (in Table 2.9 on p. 95), we list the properties of deterministic languages in Table 4.3. We symbolize a language belonging to the families *CF*, *DET* and *REG* with L, D and R, respectively. Next we argue for the listed properties and support them by examples.[3]

reversal

The language L below:

$$L = \{\, a^n\, b^n\, e \,\} \cup \{\, a^n\, b^{2n}\, d \,\} \qquad \text{for } n \geq 1$$

satisfies the two equalities $|\, x \,|_a = |\, x \,|_b$ if the sentence ends with letter e, and $2|\, x \,|_a = |\, x \,|_b$ if it ends with letter d. Language L is not deterministic, but the reversed language L^R is so. In fact, it suffices to read the first character to decide which equality has to be checked.

union

Example 4.15 on p. 219 shows that the union of deterministic languages (as L' and L'' obviously are) is in general nondeterministic. From the De Morgan identity it follows that $D \cup R = \neg\,(\,\neg D \cap \neg R\,)$, which is a deterministic language, for the following reasons: the complement of a deterministic or regular language

[3] It may be superfluous to recall that a statement such as $D^R \notin DET$ means that there exists some language D such that D^R is not deterministic.

is, respectively, deterministic or regular, and the intersection of a deterministic language and a regular one is deterministic (discussed below).

complement

The complement of a deterministic language is deterministic. The proof (similar to the one for regular languages on p. 189) constructs the recognizer of the complement by creating a new sink state and interchanging final and non-final states.[4] Alternatively, Heilbrunner method [5] transforms the *LR* (1) grammar of a deterministic language into the grammar of its complement. It follows that if a context-free language has a non-context-free one as complement, it cannot be deterministic.

intersection with a regular language

The machine constructed by the Cartesian product of a deterministic pushdown machine and a deterministic finite automaton is deterministic. To show that the intersection of two deterministic languages may go out of *DET*, recall that the language with three equal exponents (p. 93) is not context-free. Yet it can be defined by the intersection of two languages, both deterministic:

$$\left\{ a^n b^n c^n \mid \ n \geq 0 \right\} = \left\{ a^n b^n c^* \mid \ n \geq 0 \right\} \cap \left\{ a^* b^n c^n \mid \ n \geq 0 \right\}$$

concatenation and star

When two deterministic languages are concatenated, it may happen that a deterministic pushdown machine cannot localize the frontier between the strings of the first and second languages. Therefore it is unable to decide the point for switching from the former to the latter transition function. As an example, take the languages L_1 and L_2:

$$L_1 = a^+ \cup \left\{ a^i b^i \mid \ i \geq 1 \right\} \qquad L_2 = c^+ \cup \left\{ b^i c^i \mid \ i \geq 1 \right\}$$

and notice that their concatenation $L_1 \cdot L_2$:

$$L_1 \cdot L_2 = \left\{ a^j b^k c^l \mid \ j = k \text{ or } k = l \text{ with } j, k, l \geq 1 \right\} \cup a^+ c^+$$

is the disjoint union of the inherently ambiguous language L of Example 2.58 on p. 62 and a regular language we may disregard. Since any sentence in L has two distinct syntax trees, any pushdown machine that recognizes the sentence necessarily performs two distinct computations and cannot be deterministic. The situation for the star of a deterministic language is similar.

concatenation with a regular language

The recognizer of $D \cdot R$ can be constructed by *cascade composition* of a deterministic pushdown machine and a deterministic finite automaton. More precisely, when the pushdown machine enters the final state that recognizes a sentence of D, the new machine moves to the initial state of the finite recognizers of R and simulates its computation until the end.

[4] See for instance [1,2].

On the other hand, Table 4.3 says that the basic operations of regular expressions may spoil determinism when applied to a deterministic language. This creates some difficulty to language designers: when two existing technical languages are united, the result is not granted to be deterministic (it may even be ambiguous). The same danger threatens the concatenation and the star or cross of deterministic languages. In practice, the designer should check that, after transforming the language under development, determinism is not lost.

It is worth mentioning another important difference between *DET* and *CF*. While the equivalence of two *CF* grammars or pushdown automata is undecidable, an algorithm exists for checking if two deterministic automata are equivalent.[5]

4.3.2 Nondeterministic Languages

In this section we reason about the context-free languages that cannot be accepted by a deterministic pushdown automaton, starting from the next fundamental inclusion property.

Property 4.13 (family inclusion) *The family* DET *of deterministic languages is strictly included in the family* CF *of context-free languages.* □

Proof The statement follows from two known facts. First, the inclusion $DET \subseteq CF$ is obvious since a deterministic pushdown automaton is a special case of the nondeterministic one. Second, it holds $DET \neq CF$ because certain closure properties (Table 4.3) differentiate a family from the other. □

This completes the proof, but it is worthwhile exhibiting a few typical nondeterministic context-free languages in order to evidence some language paradigms that ought to be carefully avoided by language designers who strive for compiler efficiency.

Lemma of Double Service

A valuable technique for proving that a context-free language is not deterministic is based on the analysis of the sentences that are prefix of each other. Let D be a deterministic language, let $x \in D$ be a sentence, and suppose there exists another sentence $y \in D$ that is a prefix of x, i.e., it holds $x = y\,z$, where the strings x, y and z may be empty.

Now, we define another language called the *double service* of language D, by inserting a new terminal—the sharp sign ♯—between strings y and z:

$$ds(D) = \{\, y \sharp z \mid \; y \in D \;\wedge\; z \in \Sigma^* \;\wedge\; y\,z \in D \,\}$$

[5] Senizergues algorithm [6] is quite complex and is not presented in this book.

For instance, the double service of the (finite) language $F = \{ a,\ a\,b,\ b\,b \}$ is:

$$ds\,(F) = \{ a\,\sharp\,b,\ a\,\sharp,\ a\,b\,\sharp,\ b\,b\,\sharp \}$$

Clearly, the original sentences are terminated by $\sharp$ and may be followed or not by a suffix. It follows that $D\,\sharp \subseteq ds\,(D)$ and in general the containment is strict (as in the preceding example).

Lemma 4.14 (double service) *If language D is deterministic, then its double service language $ds\,(D)$ is deterministic as well.* □

Proof We are going to transform a deterministic recognizer M of language D into a deterministic one M' of the double service language of D. To simplify the construction,[6] we assume that automaton M functions *online* (p. 212). This means that as it scans the input string, it can at once decide if the scanned string is a sentence. Now, consider the computation of M that accepts string y. If string y is followed by a sharp sign $\sharp$, then the new machine M' reads the sharp and accepts if the input is finished, because $y\,\sharp \in ds\,(D)$ if $y \in D$. Otherwise, the computation of M' proceeds deterministically and scans string z exactly as machine M would do to recognize string $y\,z$. □

The reason for the curious name "double service" of the language is now clear: the automaton simultaneously performs two services inasmuch as it has to recognize a prefix and a longer string. Lemma 4.14 has the practical implication that if the double service of a *CF* language L is not *CF* (and hence not D), then the language L itself is not deterministic.

Example 4.15 (*nondeterministic union of deterministic languages*) The language L below:

$$L = \left\{ a^n\,b^n \mid \ n \geq 1 \right\} \cup \left\{ a^n\,b^{2n} \mid \ n \geq 1 \right\} = L' \cup L''$$

which is the union of two deterministic languages L' and L'', is not deterministic. Intuitively, an automaton for L has to read a number of letters a and store the number on the stack. Then, if the input string is in L', e.g., $a\,a\,b\,b$, it must pop an a upon reading a letter b. But if the input string is in L'', e.g., $a\,a\,b\,b\,b\,b$, it must read two letters b before popping one a. As the machine does not know which the correct choice is (for that it should count the number of b while examining an unbounded substring), it is obliged to carry on both computations nondeterministically.

More rigorously, assume by contradiction that language L is deterministic. Then by Lemma 4.14 also its double service language $ds\,(L)$ would be deterministic, and

[6] See Floyd and Beigel [7] for further reading.

from the closure property of *DET* (p. 216) under intersection with regular languages, the language L_R:

$$L_R = ds(L) \cap \left(a^+ b^+ \sharp b^+\right)$$

is deterministic, too. But language L_R is not context-free, hence certainly not a deterministic one, because its sentences have the form $a^i b^i \sharp b^i$, for $i \geq 1$, with three equal exponents (p. 93), a well-known non-context-free case. □

The next example applies the same method to the basic paradigm of palindromic strings.

Example 4.16 (*palindromes and nondeterminism*) The language L of palindromes is defined by grammar:

$$S \to a\,S\,a \mid b\,S\,b \mid a \mid b \mid \varepsilon$$

To prove that language L is not deterministic, we intersect its double service language $ds(L)$ with a regular one, with the aim of obtaining a language that is not in *CF*. Consider the language L_R:

$$L_R = ds(L) \cap \left(a^* b\, a^* \sharp b\, a^*\right)$$

A string of the form $a^i\, b\, a^j \sharp b\, a^k$ is in L_R if, and only if, condition $j = i$ and $k = i$ holds. But then this language L_R is again the well-known non-*CF* one with three equal exponents. □

4.3.3 Determinism and Language Unambiguity

If a language is accepted by a deterministic automaton, every sentence is recognized with exactly one computation and it is provable that the language can be generated by a non-ambiguous grammar.

The construction of Algorithm 4.7 on p. 210 produces a grammar equivalent to a pushdown automaton, which simulates computations by means of derivations: the grammar generates a sentence with a leftmost derivation if, and only if, the machine performs a computation that accepts the sentence. It follows that two distinct derivations (of the same sentence) correspond to distinct computations, which on a deterministic machine necessarily scan different strings.

Property 4.17 (grammar unambiguity condition) *Let M be a deterministic pushdown machine. Then the corresponding grammar of $L(M)$ obtained with Algorithm 4.7 on p. 210 is not ambiguous.* □

Yet, of course other grammars of $L(M)$ may be ambiguous.

A consequence is that any inherently ambiguous context-free language L is nondeterministic, i.e., cannot be recognized by a deterministic pushdown machine. Suppose by contradiction that language L is deterministic, then the preceding property states that a non-ambiguous equivalent grammar does exist. This contradicts the very definition of inherent ambiguity (p. 62) that every grammar of the language is ambiguous. The discussion above confirms what has been already said about the irrelevance of inherently ambiguous languages for technical applications.

Example 4.18 (*inherently ambiguous language and nondeterminism*) The inherently ambiguous language of Example 2.58 (p. 62) is the union of two languages:

$$L_A = \left\{ a^i\, b^i\, c^* \mid \ \ i \geq 0 \right\} \cup \left\{ a^*\, b^i\, c^i \mid \ \ i \geq 0 \right\} = L_1 \cup L_2$$

which are both deterministic (by the way, this is another proof that family *DET* is not closed under union). Intuitively, to recognize language L_A, different strategies must be adopted for the strings of the two languages. For language L_1 the letters a are pushed onto the stack and popped from it upon reading b. The same happens for language L_2, but for letters b and c. Therefore, any sentence belonging to both languages is accepted by two different computations and the automaton is nondeterministic. □

The notion of ambiguity applies to any automaton type. An *automaton* is *ambiguous* if it recognizes a sentence by two distinct computations. We observe that the determinism condition is more stringent than the absence of ambiguity: the family of deterministic pushdown automata is strictly contained in the one of non-ambiguous pushdown automata. To clarify the statement, we show two examples.

Example 4.19 (*relation between nondeterminism and unambiguity*) The language L_A of the previous example (Example 4.18) is nondeterministic and even ambiguous, because certain sentences are necessarily accepted by different computations. On the other hand, the language L_I of Example 4.15 on p. 219:

$$L_I = \left\{ a^n\, b^n \mid \ \ n \geq 1 \right\} \cup \left\{ a^n\, b^{2n} \mid \ \ n \geq 1 \right\} = L' \cup L''$$

though nondeterministic (as argued there) is unambiguous. In fact, each sublanguage L' and L'' is easily defined by an unambiguous grammar, and the union is still unambiguous because the components are disjoint. The two recognition strategies described in Example 4.18 are implemented by distinct computations, but at most one of them may succeed for any given string. □

4.3.4 Subclasses of Deterministic Pushdown Automata and Languages

By definition, the family *DET* is associated with the most general type of deterministic pushdown machine, the one featuring several states and using final states for

acceptance. Various limitations on the internal states and acceptance modes, which do not have any consequence in the nondeterministic case, cause a restriction of the language family recognized by a deterministic machine. The main limitations[7] are briefly mentioned:

automaton with only one state
: Acceptance is necessarily by empty stack, and it is less powerful than recognition by final state.

limitation on the number of states
: Any language family obtained by limiting the number of states is more restricted than *DET*. A similar loss is caused if a limit is imposed on just the number of final states or, more generally, on the number of final configurations of the machine.

real-time functioning
: A machine with the real-time property (Sect. 4.2.3.1) may not have spontaneous moves, yet certain deterministic context-free languages need them to be recognized by means of a deterministic pushdown automaton. We illustrate with the language L of Example 4.10 on p. 213:

$$L = \left\{ a^m b^n c^n \mid \ m, n \geq 1 \right\} \cup \left\{ a^m b^n d^m \mid \ m, n \geq 1 \right\} \tag{4.1}$$

A nondeterministic real-time machine recognizing language L was there described, yet a simple reasoning suffices to prove that a deterministic machine M_D recognizing L with the real-time property does not exist.

We observe that the two sublanguages in (4.1) have a common prefix of unbounded length of the form $a^m b^n$. After processing such a prefix for an input string x, any machine M_D needs to have recorded in the stack a string of symbols, say $A^m B^n$, which remember the values m and n, otherwise it would be later impossible to check at least one of the identities $|x|_b = |x|_c$ or $|x|_a = |x|_d$. Anyway, to check the latter identity, the machine must first perform n spontaneous moves to clear the stack from symbols B. Therefore, the machine violates the real-time property. Notice however that the machine is deterministic, because the next input character c or d determines the move to be performed.

In many practical situations, technical languages are designed so as to be deterministic. For instance, this is the case for almost all the programming languages and for the families *XML*, *HTML*, etc, of markup languages. Different approaches exist for ensuring that a language is deterministic, by imposing some conditions on the language or on the grammar thereof. Depending on the condition, one obtains a different subfamily of *DET*.

Two simple cases are next described to introduce the topic. Others, much more important for the applications, will be obtained with the conditions $LL(k)$, $LR(k)$ and operator precedence.

[7] For further reading, see [1,8].

4.3.4.1 Simple Deterministic Languages

A grammar is called *simple deterministic* if it satisfies the next conditions:

1. every rule right part starts with a terminal character, hence empty rules and the rules starting with a nonterminal symbol are excluded.
2. for any nonterminal A, there do not exist any alternatives that start with the same character:

$$\nexists\ (A \to a\,\alpha \mid a\,\beta) \qquad \text{with } a \in \Sigma,\ \alpha,\ \beta \in (\Sigma \cup V)^* \text{ and } \alpha \neq \beta$$

An example of a simple deterministic grammar is: $S \to a\,S\,b \mid c$.

Clearly, if we construct the pushdown machine from a simple deterministic grammar by means of Algorithm 4.1 on p. 202, we obtain a deterministic machine. Moreover, this machine consumes a character with each move, i.e., it works in real time, which is consistent with such grammar rules being in the real-time normal form (p. 78). We hasten to say that anyway the simple deterministic condition is too inconvenient for a practical use.

4.3.4.2 Parenthesis Languages and Input-Driven Languages

It is easy to check that the parenthesized languages (introduced in Chap. 2 on p. 47), generated by parenthesized grammars, are deterministic. We assume that in such a grammar any two rules differ in their right-hand sides, i.e., the grammar is *invertible*, or that the grammar is *distinctly parenthesized* (p. 50); either assumption ensures that the grammar is unambiguous.

Any sentence generated by a parenthesized grammar has a bracketed structure that marks the start and end of the right part of each rule used in the derivation. A simple recognition algorithm scans the input string, and it localizes a substring that does not contain any parentheses and is enclosed between two matching parentheses. Then the set of the right parts of the grammar rules is searched for such a parenthesized substring. If none of such parts matches, the input string is rejected. Otherwise the substring is reduced to the corresponding nonterminal, i.e., the left part of the matching rule, thus producing a shorter input string to be recognized. Then the algorithm resumes scanning the new input string in the same way, and at last it recognizes and halts when the string is reduced to the axiom. It would not be difficult to encode the algorithm by means of a deterministic pushdown machine.

If a grammar defines a nondeterministic language or if it is ambiguous, parenthesizing the grammar (see Definition 2.45 on p. 50) removes both defects. For instance, the language of the even-length palindromes is changed into a deterministic one when its grammar $S \to a\,S\,a \mid b\,S\,b \mid \varepsilon$ (on p. 220) is parenthesized (here we use letters a and b to tag brackets) as follows:

$$S \to \underbrace{`_a(\,'\,a\,S\,a\,`)_a\,'}_{\text{alternative } a} \mid \underbrace{`_b(\,'\,b\,S\,b\,`)_b\,'}_{\text{alternative } b} \mid \underbrace{`(\,'\ `)\,'}_{\text{center}}$$

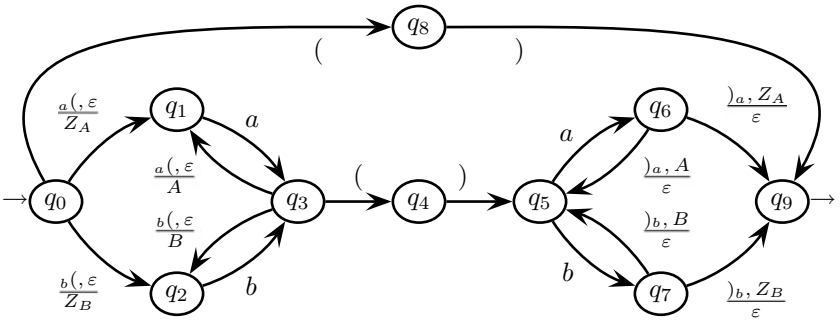


Fig. 4.6 The pushdown automaton for the language of even length palindromes. This machines qualifies as input-driven, see Definition 4.20 below

Now, the midpoint of a sentence such as $a\ b\ b\ b\ b\ a$ is marked by the substring "( )" in the parenthesized version "$_a(\ a\ _b(\ b\ _b(\ b\ (\ )\ b\ )_b\ b\ )_b\ a\ )_a$". The corresponding deterministic pushdown machine is shown in Fig. 4.6; notice that it has the real-time property.

We recall that move $\frac{a(,\varepsilon}{A}$ (or initially $\frac{a(,\varepsilon}{Z_A}$) reads terminal "$_a($" without reading or popping anything from stack and pushes symbol A (or initially Z_A) onto stack. Moreover, when a move does not perform either a pop or a push on reading a terminal, say b, we simply denote the move as $q_2 \xrightarrow{b} q_3$ instead of $q_2 \xrightarrow{\frac{b,\varepsilon}{\varepsilon}} q_3$.

By carefully observing Fig. 4.6, we discover a close association between terminals and move types:

- *push operations*, also known as *calls*, are performed for terminals "$_a($" and "$_b($"
- *pop operations*, also known as *returns*, are performed for terminals "$)_a$" and "$)_b$"
- *internal operations*, i.e., exclusively on the state, are performed for terminals a, b, "(" and ")"

When such an association exists, each terminal symbol commands one and only one move type. It is convenient to group together the terminals that command the same move type, thus partitioning the terminal alphabet Σ into three disjoint subsets, respectively, named Σ_{call}, Σ_{return} and $\Sigma_{internal}$. Their values for the example of Fig. 4.6 are:

- $\Sigma_{call} = \{\ `\ _a(\ ',\ `\ _b(\ '\ \}$ contains all the terminals commanding *push*
- $\Sigma_{return} = \{\ `\ )_a\ ',\ `\ )_b\ '\ \}$ contains all the terminals commanding *pop*
- $\Sigma_{internal} = \{\ a,\ b,\ `(\ ',\ `)\ '\ \}$ contains all the terminals commanding an *internal* (also called *neutral*) move

A pushdown automaton that features such a 3-partition of the terminal alphabet is appropriately named *input-driven* (*ID*), or also *visibly pushdown*, where both names suggest that the current input character determines which action has to be undertaken

on the stack. Next we formally define the input-driven property for a pushdown machine.[8]

Definition 4.20 (*input-driven deterministic pushdown machine*) Let M be a deterministic pushdown automaton with input alphabet Σ, stack alphabet Γ and state set Q. Machine M has the *input-driven* (*ID*) *property* if alphabet Σ is the disjoint union of three alphabets Σ_{call}, Σ_{return} and $\Sigma_{internal}$, such that the state-transition function δ has this form:

#	*domain*	*range*	*move description*
1	$Q \times \Sigma_{call}$	$Q \times \Gamma$	a *push* move does not read (so does not check either) the top-of -stack symbol, and writes one or more symbols onto the stack
2	$Q \times \Sigma_{return} \times \Gamma$	Q	a *pop* move reads (and so checks) the top-of-stack symbol, then deletes it; but if the top symbol is Z_0, i.e., the stack is empty, the move proceeds without deleting Z_0
3	$Q \times \Sigma_{internal}$	Q	an *internal* move just ignores the stack, i.e., it does not read or write on it

Machine M recognizes the input string if, by starting with an empty stack, it reaches a final state when the input has been entirely scanned. A *language* recognized by an *ID* machine is called *input-driven*. □

Notice that spontaneous moves are excluded. Given a pushdown machine, it is immediate to verify the property *ID* and to split the alphabet into the three subsets. In particular, any deterministic finite automaton is clearly identical to an *ID* machine with the trivial 3-partition $\Sigma = \Sigma_{internal}$. Thus, family *REG* is strictly included in family *ID*.

Since the moves of an *ID* machine are more constrained than those of a generic deterministic real-time pushdown machine, it is not surprising that some deterministic real-time *CF* languages cannot be recognized by an *ID* machine. A situation preventing a language from being *ID* occurs when the same terminal symbol is associated with different move types, e.g., push and pop. The deterministic language:

$$\left\{ a^m b^m c^n a^n \mid \ m, n \geq 1 \right\}$$

[8] For simplicity, we consider deterministic pushdown machines, but the property *ID* can be defined for the nondeterministic case as well. It is noteworthy that the families of the languages, respectively, recognized by deterministic *ID* machines and by nondeterministic ones, coincide. In other words, every nondeterministic *ID* automaton can be always transformed into an equivalent deterministic *ID* machine in a way similar to the determinization of a finite-state machine. For more information, see [9,10].

is not of type *ID* because letter a acts as a call symbol in the prefix $a^m\, b^m$, as the machine must store the count of letters a to check that the number of letters b is equal, whereas it acts as a return symbol in the suffix $c^n\, a^n$. Notice that both prefix and suffix are separately of type *ID*, yet their concatenation is not. Moreover, the union of the prefix and suffix sets, i.e., $\{\ a^m\, b^m \mid \ \ m \geq 1\ \} \cup \{\ c^n\, a^n \mid \ \ n \geq 1\ \}$, presents the same conflict, thus proving that the union of two *ID* languages in general is not input-driven.

On the other hand, concatenation and union preserve the input-driven property if the two languages combined have the same 3-partitioned alphabet. We can check this statement for the following languages L_1 and L_2:

$$L_1 = \left\{\ a^m\, b^m \mid \ \ m \geq 1\ \right\} \qquad L_2 = \left\{\ a^n\, c^n \mid \ \ n \geq 1\ \right\} \tag{4.2}$$

Clearly, letter a triggers a push move, and letters b and c a pop move, therefore the 3-partitions of L_1 and L_2 are identical:

$$\Sigma_{call} = \{\ a\ \} \qquad \Sigma_{return} = \{\ b,\ c\ \} \qquad \Sigma_{internal} = \emptyset \tag{4.3}$$

For both concatenation $L_1 \cdot L_2$ and union $L_1 \cup L_2$, we leave it to the reader to combine the pushdown automata of the two languages into one, and to check that the combined one has the *ID* property.

This example prompts another remark, if we, respectively, view letter a as an open bracket, and letters b and c as closed ones: in an *ID* language the same open bracket can match different closed brackets. This breaks the traditional rule of a parenthesis language that the open and closed parenthesis types are in a one-to-one relation.

Now, we consider a language composition case that looks more problematic for the *ID* property. We concatenate the language L_1 of line (4.2) above, with language $L_3 = a^+\, b$, which is recognized by the finite automaton having the transitions:

$$q_0 \xrightarrow{a} q_1 \qquad q_1 \xrightarrow{a} q_1 \qquad q_1 \xrightarrow{b} q_2 \qquad \text{with state } q_2 \text{ final}$$

Clearly, according to Definition 4.20 (case 3) all the moves are internal, i.e., letters a and b are in $\Sigma_{internal}$, thus clashing with the assignment of a to class Σ_{call} in language L_1. But it would be wrong to conclude that language $L_1 \cdot L_3$ violates the *ID* condition! The pushdown machine for $L_1 \cdot L_3$, shown in Fig. 4.7, has the same 3-partition as in (4.3). Notice that all symbols C pushed onto the stack by the moves that follow state q_3 are useless, because there is no need to count the number of letters a in a^+; their presence in the stack is just an artifact of assigning letter a to subset Σ_{call}.

Another lesson from the above example is that, for each regular language, we can arbitrarily choose a 3-partition and construct an *ID* machine accepting the language. For instance, if we choose the trivial 3-partition $\Sigma = \Sigma_{return}$, all the machine moves are of type pop, thus the stack stays empty; such a machine would then work as a finite automaton accepting the same language.

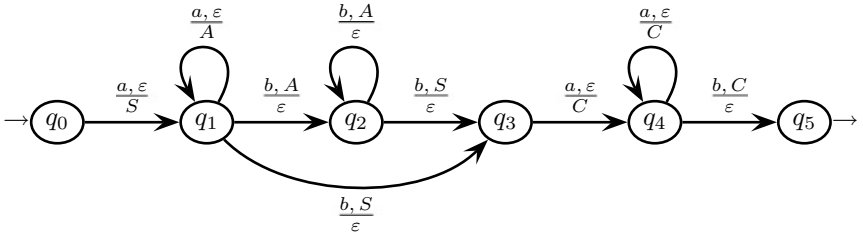


Fig. 4.7 The automaton of type *ID* that recognizes language $\{ a^m b^m \mid \ m \geq 1 \} \cdot a^+ b$

Table 4.4 Properties of the input-driven family *ID* and of the subfamilies ID_{part} characterized by the same alphabet partition, along with the known properties of family *DET*

Operation	Property of ID and ID_{part}	Property of DET
Reversal	$I^R \in ID$	$D^R \notin DET$
Union	$I'_{part} \cup I''_{part} \in ID_{part}$	$D_1 \cup D_2 \notin DET$
Complement	$\neg I_{part} \in ID_{part}$	$\neg D \in DET$
Intersection	$I'_{part} \cap I''_{part} \in ID_{part}$	$D_1 \cap D_2 \notin CF$
Concatenation	$I'_{part} \cdot I''_{part} \in ID_{part}$	$D_1 \cdot D_2 \notin DET$
Star	$I^*_{part} \in ID_{part}$	$D^* \notin DET$

4.3.4.3 Properties of Input-Driven Languages

Input-driven languages are quite remarkable for their mathematical properties. To start, notice that the family *ID* of input-driven languages is strictly included in the family *DET* and strictly contains family *REG*. Then, within the family *ID*, we focus on the set of the languages characterized by the same 3-partition of the terminal alphabet, and we call each such set a *partitioned ID subfamily*, to be denoted by ID_{part}. The closure properties of (sub)family *ID* and ID_{part} are listed in Table 4.4.[9] We symbolize a language belonging to family *ID* and to the subfamily with a given 3-partition as I and I_{part}, respectively. For comparison, we also list the known properties of family *DET*.

Notice that every subfamily ID_{part} has the same closure properties as the regular languages. In particular, closure under intersection, which follows from closure under union and complement, and from the De Morgan identity, implies the decidability of the emptiness question: is it true that $I'_{part} \cap I''_{part} = \emptyset$? On the contrary, for unrestricted deterministic languages the same question is undecidable. Having a decision algorithm for intersection emptiness is useful for a class of important appli-

[9] For a justification of the properties we have not discussed, we refer the reader to [9–11].

cations (out of scope for this book), such as the formal verification of a hardware and software system modeled by means of an automaton.[10]

Coming back to the compiler business, a relevant question is whether family *ID* is a good model for defining artificial and technical languages. The answer is yes and no. Truly, some data definition languages such as the markup languages *XML* and *HTML*, or the *Javascript Object Notation JSON*, admit a grammar of type *ID*. This is not surprising, if we consider that their sentences essentially consist of nested and concatenated parenthetical constructs, and that we have introduced the *ID* machines as recognizers for parenthesis languages. Web documents and semi-structured databases are often encoded in *XML*. Distinct opening and closing marks are used to delimit the document parts, to allow an efficient recognition and transformation thereof. Therefore, the *XML* family is deterministic. The *XML* grammar model, technically known as *Document Type Definition* or *DTD*, is similar to the parenthesis grammar model, but it uses various regular expression operators in the grammar rules, as a distinguishing feature.[11]

Example 4.21 (*input-driven model of language JSON*) Language *JSON* was designed as a compact replacement of *XML* for representing nested data structures. It is closer to the input-driven model, as it uses only two bracket pairs: square and curly. We construct a deterministic *ID* recognizer for a large fragment of *JSON*, in a modular way.

Consider the *EBNF* grammar of *JSON* in Example 2.96 on p. 102. For brevity, we drop terminals num and bool, which do not need a stack for their recognition, and we schematize nonterminal STRING as a terminal s, for the same reason. A 3-partition of the alphabet is:

$$\Sigma_{call} = \{\ \text{`\{'},\ \text{`[' }\}\quad \Sigma_{return} = \{\ \text{`}\}\text{'},\ \text{`]' }\}\quad \Sigma_{internal} = \{\ \mathsf{s},\ \text{`:'},\ \text{`,' }\}$$

(terminals are quoted to avoid confusion). By repeatedly substituting grammar rules, we obtain a grammar with one rule (V stands for VALUE):

$$V \to \mathsf{s} \mid \underbrace{\text{`[' }\left(V\ (,V)^*\right) \mid \varepsilon\ \text{`]'}}_{\text{alternative ARRAY}} \mid \underbrace{\text{`\{' }\left(\mathsf{s}{:}V\ (,\mathsf{s}{:}V)^*\right) \mid \varepsilon\ \text{`}\}\text{'}}_{\text{alternative OBJECT}} \tag{4.4}$$

In rule (4.4), we consider the ARRAY alternative and the terminal rule $V \to \mathsf{s}$ for a string. The case of the OBJECT alternative is similar and is omitted. Figure 4.8 shows the deterministic *ID* machine for arrays. Stack symbols A and Z (initial removable symbol) count and match square brackets. An array element can be a string, an array, or a comma-separated list thereof, or even nothing. Recognition is by final state (in

[10] For an introduction to system model checking, see [12,13].

[11] Such *DTD* grammars, also named regular tree grammars, generate context-free languages, but they differ from context-free grammars in several ways; see [14,15] for more information.

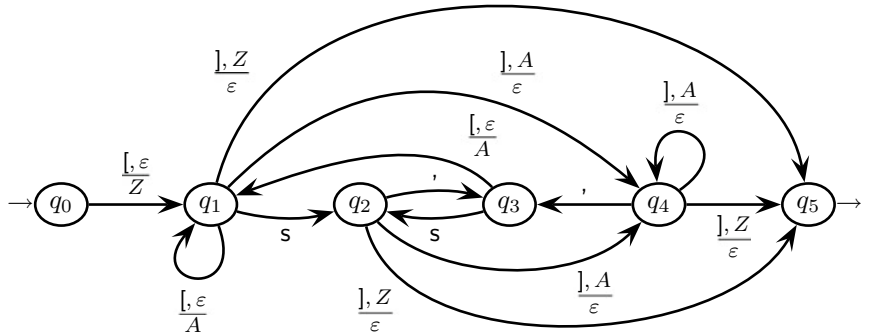


Fig. 4.8 State-transition graph of a deterministic *ID* machine for *JSON* arrays, defined by rule (4.4)

q_5). Table 4.5 tabulates the computation for the input [s, [s, [ ], s]], which specifies a 3-level array. Since language *JSON* has a parenthesis structure, from the trace of the *ID* machine we can construct the abstract tree of the input string. In a trace, we pair the push and pop operations that work on the same stack symbol, we create a tree

Table 4.5 Recognition of input [s, [s, [ ], s]] by the *ID* machine of Fig. 4.8

Machine configuration			Machine action		
State	Input	Stack	Input	Stack	Move type
q_0	[s, [s, [ ], s]]	$Z_0\|$	Initial configuration		
q_0	[s, [s, [ ], s]]	$Z_0\|$	[	*push Z*	Push
q_1	s, [s, [ ], s]]	$\cdots\|Z$	s	—	Internal
q_2	, [s, [ ], s]]	$\cdots\|Z$	,	—	Internal
q_3	[s, [ ], s]]	$\cdots\|Z$	[	*Push A*	Push
q_1	s, [ ], s]]	$\cdots\|ZA$	s	—	Internal
q_2	, [ ], s]]	$\cdots\|ZA$	,	—	Internal
q_3	[ ], s]]	$\cdots\|ZA$	[	*Push A*	Push
q_1	], s]]	$\cdots\|ZAA$	]	*Pop A*	Pop
q_4	, s]]	$\cdots\|ZA$	,	—	Internal
q_3	s]]	$\cdots\|ZA$	s	—	Internal
q_2	]]	$\cdots\|ZA$	]	*Pop A*	Pop
q_4	]	$\cdots\|Z$	]	*Pop Z*	Pop
q_5	Finished	$Z_0\|$	Final configuration		

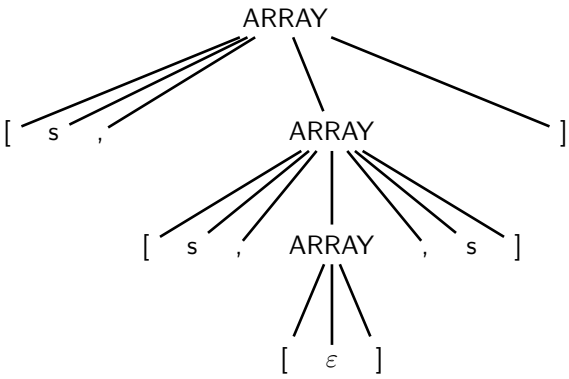


Fig. 4.9 *JSON* abstract syntax tree of the input [s, [s, [ ], s]] constructed from the computation trace in Table 4.5

node for each pair, and we append to the node (from left to right) as its siblings the input characters and the inner nodes in between. Figure 4.9 shows an abstract tree for *JSON*. It would not be difficult to automate such a tree construction procedure, thus a parser from our *JSON ID* pushdown recognizer. Yet its relation to a specific grammar, e.g., the *EBNF* one in Example 2.96, is rather indirect. Better methods will be explained in the next sections, to directly construct a parser starting from a *CF* grammar. □

We will return to language *JSON* in Sect. 4.8, to see how to automatically obtain a deterministic (and even parallel) parser for *JSON* starting from a grammar of the operator-precedence type. Such grammars define a language family intermediate between *ID* and *DET*.

On the other hand, the constructs of most programming languages, such as *Java*, are more complex than those of data representation languages and are beyond the capability of *ID* languages. Differently said, parenthesizing a grammar to make it input-driven is too drastic a remedy to be acceptable by programmers and language users. Nevertheless, the idea that in a language certain terminal characters may act as opening parentheses and certain others as closing ones is present in all the artificial languages.

4.4 Syntax Analysis: Top-Down and Bottom-Up Constructions

The rest of the chapter explains the difference between top-down and bottom-up sentence recognition, and systematically covers the practical parsing algorithms used by compilers, also when grammars are extended with regular expressions.

Consider a grammar G. If a source string is in the language $L(G)$, a syntax analyzer or parser scans the string and computes a derivation or syntax tree; otherwise it stops and prints the configuration where an error was detected (diagnosis); afterward it may resume parsing and skip the substrings contaminated by the error (error recovering), in order to offer as much diagnostic help as possible with a single scan

of the source string. Thus an analyzer is simply a recognizer capable of recording a string derivation and possibly of offering error treatment. Therefore, when a push-down machine performs a move corresponding to a grammar rule, it has to save the rule label into some kind of data structure. Upon termination, such a data structure will represent the syntax tree.

If the source string is ambiguous, the result of the analysis is a set of trees, also called a *tree forest*. In that case the parser may decide to stop as soon as it finds a derivation, or to exhaustively produce all the derivations.

We know (p. 46) that the same syntax tree corresponds to many derivations, notably the leftmost and rightmost ones, as well as to less relevant others. Depending on the derivation being leftmost or rightmost, and on the construction order, we obtain two most important parser classes:

top-down analysis
: Constructs the leftmost derivation by starting from the axiom, i.e., the tree root, and growing the tree toward the leaves. Each algorithm step corresponds to a derivation step.

bottom-up analysis
: Constructs the rightmost derivation, but in the reversed order, i.e., from the leaves to the root of the tree. Each step corresponds to a reduction.

The complete algorithms are described in the next sections. Here their functioning is best introduced through an example.

Example 4.22 (*tree visit orders*) Consider the grammar below:

$$\begin{cases} 1\colon\ S \to a\,S\,A\,B \\ 3\colon\ A \to b\,A \\ 5\colon\ B \to c\,B \end{cases} \qquad \begin{cases} 2\colon\ S \to b \\ 4\colon\ A \to a \\ 6\colon\ B \to a \end{cases}$$

and the sentence $a^2\,b^3\,a^4$. The orders corresponding to the top-down and bottom-up visits are in Fig. 4.10. In each node the framed number gives the visit order and the nonterminal subscript indicates the grammar rule applied.

A top-down analyzer grafts the rule right part under the node representing the corresponding left part, which is a nonterminal. If the right part contains terminal symbols, these have to orderly match the characters of the source string. The procedure terminates when all the nonterminal symbols have been matched to terminal ones or to empty strings.

On the other hand, starting with the source text, a bottom-up analyzer reduces to a nonterminal node a source substring that matches a rule right part, as it finds it in a left-to-right scan. After reducing, it scans again the modified text to find the next reduction, until the text is reduced to the axiom. □

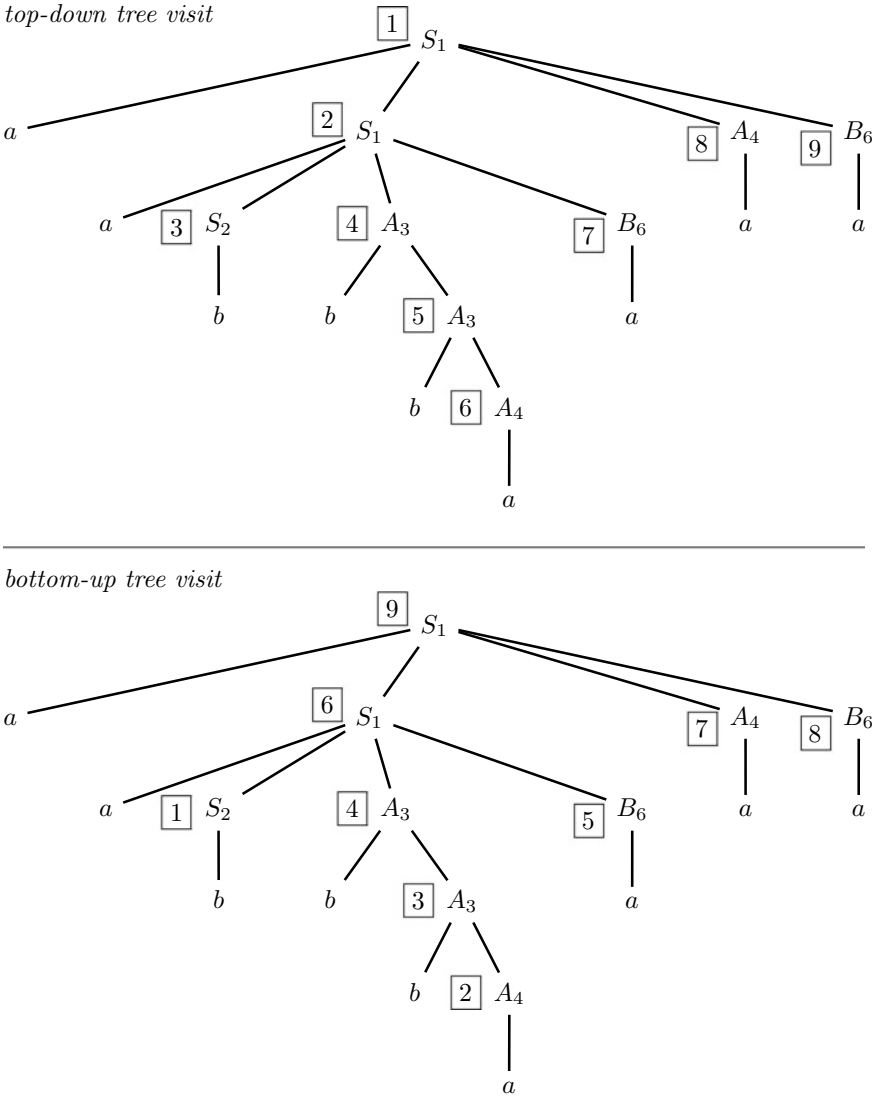


Fig. 4.10 Left-to-right, top-down and bottom-up visits of a syntax tree for Example 4.22

The dichotomy top-down/bottom-up can be made to apply to extended *BNF* grammars, too, and we postpone the discussion to later sections.

In principle, syntax analysis would work as well from right to left, by scanning the reversed source string; in reality all the existing languages are designed for left-to-right processing, as it happens in the natural language where the reading direction matches the uttering order by a speaker. Moreover, reversing the scanning direction may cause loss of language determinism, because family *DET* is not closed under reversal (p. 216).

Finally, the diffusion of parallel processing techniques motivates the study of new algorithms that would parse a long text—such as a complex web page—by means of parallel processes, each one acting on a subtext.

4.5 Grammar as Network of Finite Automata

We are going to represent a grammar as a network of finite machines. This form has several advantages: it offers a pictorial representation, gives evidence to the similarities across different parsing algorithms, allows to directly handle grammars with regular expressions, and maps quite nicely on recursive descent parser implementations.

In a grammar, each nonterminal is the left part of one or more alternatives. On the other hand, if a grammar G is in the *extended context-free form* (*EBNF*) on p. 100, a rule right part may contain the union operator, which makes it possible to define each nonterminal by just one rule $A \to \alpha$, where α is an r.e. over the alphabet of terminals and nonterminals. The r.e. α defines a regular language, the finite recognizer M_A of which is easily constructed through the methods of Chap. 3.

In the trivial case when the string α contains just terminal symbols, the automaton M_A recognizes the language $L_A(G)$ generated by grammar G starting from nonterminal A. But since in general the right part α includes nonterminal symbols, we next examine what to do with an arc labeled by a nonterminal B. This case can be thought of as the invocation of another automaton, namely the one associated with rule $B \to \beta$. Notice that symbol B may coincide with A, thus causing the invocation to be recursive.

In this chapter, to avoid confusion we call such finite automata as "machines", and we reserve the term "automaton" for the pushdown automaton that accepts the language $L(G)$. It is convenient, though not strictly necessary, for the machines to be deterministic; if not, they can be made deterministic through the methods of Chap. 3.

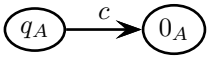
Definition 4.23 (*recursive net of finite deterministic machines*)

- Let Σ and $V = \{ S, A, B, \ldots \}$ be, respectively, the terminal and nonterminal alphabets, and let S be the axiom, of an *EBNF* grammar G.
- For each nonterminal A there is exactly one (extended) grammar rule $A \to \alpha$ and the rule right part α is an r.e. over alphabet $\Sigma \cup V$.
- We denote the grammar rules by $S \to \sigma$, $A \to \alpha$, $B \to \beta$, ... Symbols R_S, R_A, R_B, ... denote the regular languages over alphabet $\Sigma \cup V$, defined by the r.e. σ, α, β, ..., respectively.
- Symbols M_S, M_A, M_B, ... are the names of the (finite deterministic) machines that accept the corresponding regular languages R_S, R_A, ... The collection of all such machines, i.e., the *net*, is denoted by symbol $\mathcal{M}$.
- To prevent confusion, the names of the states of any two machines are made disjoint, say, by appending the machine name as a subscript. The state set of a

machine M_A is denoted $Q_A = \{\ 0_A,\ \ldots,\ q_A,\ \ldots\ \}$, its only initial state is 0_A and its final state set is $F_A \subseteq Q_A$. The *state set* Q of a *net* $\mathcal{M}$ is the union of all the states of the component machines:

$$Q = \bigcup_{M_A \in \mathcal{M}} Q_A$$

The transition function of all the machines, i.e., of the whole net, will be denoted by the same name δ as for the individual machines, at no risk of confusion since the machine state sets are all mutually disjoint.

- For a state q_A, the symbol $R\,(M_A,\ q_A)$, or for brevity $R\,(q_A)$, denotes the regular language over alphabet $\Sigma \cup V$ that is accepted by machine M_A starting from q_A. For the initial state, we have $R\,(0_A) \equiv R_A$.
- It is convenient to stipulate that every machine M_A may not have any arc like 

($c \in \Sigma \cup V$) which enters the initial state 0_A. Such a normalization ensures that the initial state may not be visited twice, i.e., it is not recirculated or reentered, within a computation that does not leave machine M_A. □

We stress that any rule $A \to \alpha$ is faithfully represented by machine M_A.[12] Therefore, a machine net $\mathcal{M} = \{\ M_S,\ M_A,\ \ldots\ \}$ is essentially a notational variant of a grammar, and we may go on using already known concepts such as derivation and reduction. In particular, the terminal language $L\,(\mathcal{M})$ (over alphabet Σ) defined (or recognized) by the machine net coincides with the language generated by the grammar, i.e., $L\,(G)$.

We need to extend the previous definition to the terminal language defined by a generic machine M_A, but starting from any state. For any state q_A, not necessarily initial, we define the following language:

$$L\,(M_A,\ q_A) = L\,(q_A) = \left\{\ y \in \Sigma^* \mid\ \ \eta \in R\,(q_A)\ \wedge\ \eta \stackrel{*}{\Rightarrow} y\ \right\}$$

The formula above contains a string η over terminals and nonterminals, accepted by machine M_A when starting in the state q_A. The derivations originating from η produce all the terminal strings of language $L\,(q_A)$.

In particular, from previous stipulations it follows that:

$$L\,(M_A,\ 0_A) = L\,(0_A) \equiv L_A\,(G)$$

and for the axiom it is:

$$L\,(M_S,\ 0_S) = L\,(0_S) = L\,(\mathcal{M}) \equiv L\,(G)$$

[12] Of course two different but equivalent r.e. are represented by machines that are equivalent or identical.

Example 4.24 (*machine net for arithmetic expressions*) The *EBNF* grammar below has alphabet $\{+,\ -,\ \times,\ /,\ a,\ \text{`(`},\ \text{`)`}\}$, three nonterminals E, T and F (axiom E), and therefore three rules:

$$\begin{cases} E \ \to\ [+\ |\ -]\, T\, \big((+\ |\ -)\, T\big)^* \\ T \ \to\ F\, \big((\times\ |\ /)\, F\big)^* \\ F \ \to\ a\ |\ \text{`(`}\, E\, \text{`)`} \end{cases}$$

Figure 4.11 shows the machines of net $\mathcal{M}$.

We list a few languages defined by the net and the component machines, to illustrate the definitions:

$$\begin{aligned} R\,(M_E,\, 0_E) &= R\,(M_E) = [+\ |\ -]\, T\, \big((+\ |\ -)\, T\big)^* \\ R\,(M_T,\, 1_T) &= R\,(1_T) = \big((\times\ |\ /)\, F\big)^* \\ L\,(M_E,\, 0_E) &= L\,(0_E) = L_E\,(G) = L\,(G) = L\,(\mathcal{M}) \\ &= \{\, a,\ a + a,\ a \times (a - a),\ \ldots \,\} \\ L\,(M_F,\, 0_F) &= L\,(0_F) = L_F\,(G) = \{\, a,\ (a),\ (-a + a),\ \ldots \,\} \\ L\,(M_F,\, 2_F) &= L\,(2_F) = \{\, \text{`)`} \,\} \end{aligned}$$

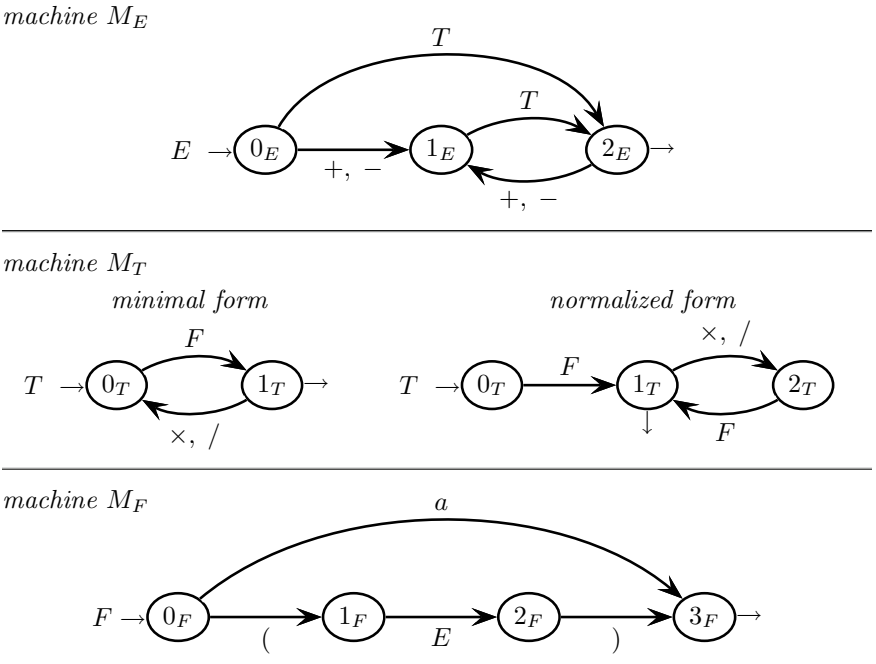


Fig. 4.11 Machine net of Example 4.25; machine M_T is shown in the minimal form (left) and after normalizing its initial state (right)

We show two examples of derivations (Chap. 2 on p. 46) and the corresponding reductions:

derivations

$$T - T \Rightarrow F \times F \times F - T \qquad \text{and} \qquad T - T \overset{2}{\Rightarrow} a \times F \times F - T$$

reductions

$F \times F \times F - T$ reduces to $T - T$ denoted $F \times F \times F - T \rightsquigarrow T - T$

$$a \times F \times F - T \overset{2}{\rightsquigarrow} T - T$$

□

Example 4.25 (*running example*) The grammar G and machine net $\mathcal{M}$ are shown in Fig. 4.12. The language can be viewed as obtained from the Dyck language by allowing a character a to replace a well-parenthesized substring; alternatively, it can be viewed as a set of simple expressions, with terms a, a concatenation operation with the operator left implicit, and parenthesized subexpressions. All the machines in $\mathcal{M}$ are deterministic and their initial states are not reentered. Most features needed to exercise different aspects of parsing are present: self-nesting, iteration, branching, multiple final states and a nullable nonterminal.

To show how it works, we list some of the languages defined by the machine net, along with their aliases:

$$\begin{aligned} R(M_E, 0_E) &= R(0_E) = T^* \\ R(M_T, 1_T) &= R(1_T) = E) \\ L(M_E, 0_E) &= L(0_E) = L(G) = L(\mathcal{M}) \\ &= \{\, \varepsilon,\ a,\ a\,a,\ (),\ a\,a\,a,\ (a),\ a\,(),\ ()\,a,\ ()\,(),\ \ldots \} \\ L(M_T, 0_T) &= L(0_T) = L_T(G) = \{\, a,\ (),\ (a),\ (a\,a),\ (()),\ \ldots \} \end{aligned}$$

□

To identify the machine states, an alternative convention, quite used for *BNF* (non-extended) grammars, relies on *marked grammar rules*. For instance, the states of machine M_T in Fig. 4.12 which represents the alternatives $T \to$ '(' E ')' | a, have the following aliases:

$$\begin{aligned} 0_T &\equiv T \to \bullet\, a \mid \bullet\,(E) & \qquad 2_T &\equiv T \to (E \bullet) \\ 1_T &\equiv T \to (\bullet\, E) & \qquad 3_T &\equiv T \to a \bullet \mid (E) \bullet \end{aligned} \tag{4.5}$$

where the bullet $\bullet$ is a character not in the terminal alphabet Σ.

EBNF grammar G (axiom E)

$$\Sigma = \{\ a,\ `(',\ `)'\ \} \qquad V = \{\ E,\ T\ \} \qquad P = \begin{cases} E \to T^* \\ T \to `(' \ E \ `)' \ \mid \ a \end{cases}$$

machine net $\mathcal{M}$

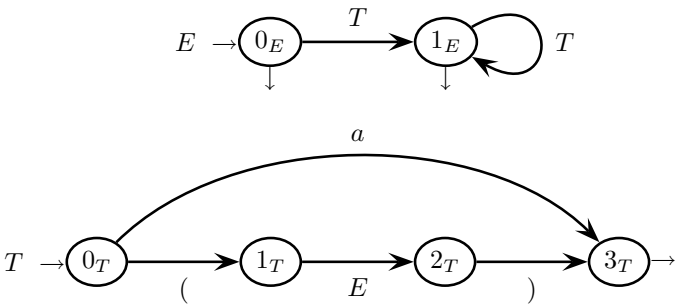


Fig. 4.12 *EBNF* grammar G and net $\mathcal{M}$ of the running example (Example 4.25)

4.5.1 Syntax Charts

As a short intermission, we cite the use of machine nets for the documentation of technical languages. In many branches of engineering it is customary to use graphics for technical documentation in addition to textual documents, which in our case are grammars and regular expressions. The so-called *syntax charts* are frequently included in the language reference manuals, as a pictorial representation of extended context-free grammars, and under the name of transition networks they are used in computational linguistics to represent the grammars of natural languages. The popularity of syntax charts derives from their readability as well as from the fact that this representation serves two purposes at once: to document the language and to describe the control flow of parsing procedures, as we will see.

Actually syntax charts differ from machine nets only with respect to their graphic style and naming. First, every finite machine graph is converted to the dual graph or chart by interchanging arcs and nodes. The nodes of a chart are the symbols (terminal and non-) occurring in the right part of a rule $A \to \alpha$. In a chart the arcs are not labeled; this means the states are not distinguished by a name, but only by their position in the graph. The chart has one entry point tagged by a dart with the chart nonterminal A, and one exit point similarly tagged by a dart, but corresponding to one or more final states. Different graphic styles are found in the manuals to visually differentiate the two classes of symbols, i.e., terminal and nonterminal.

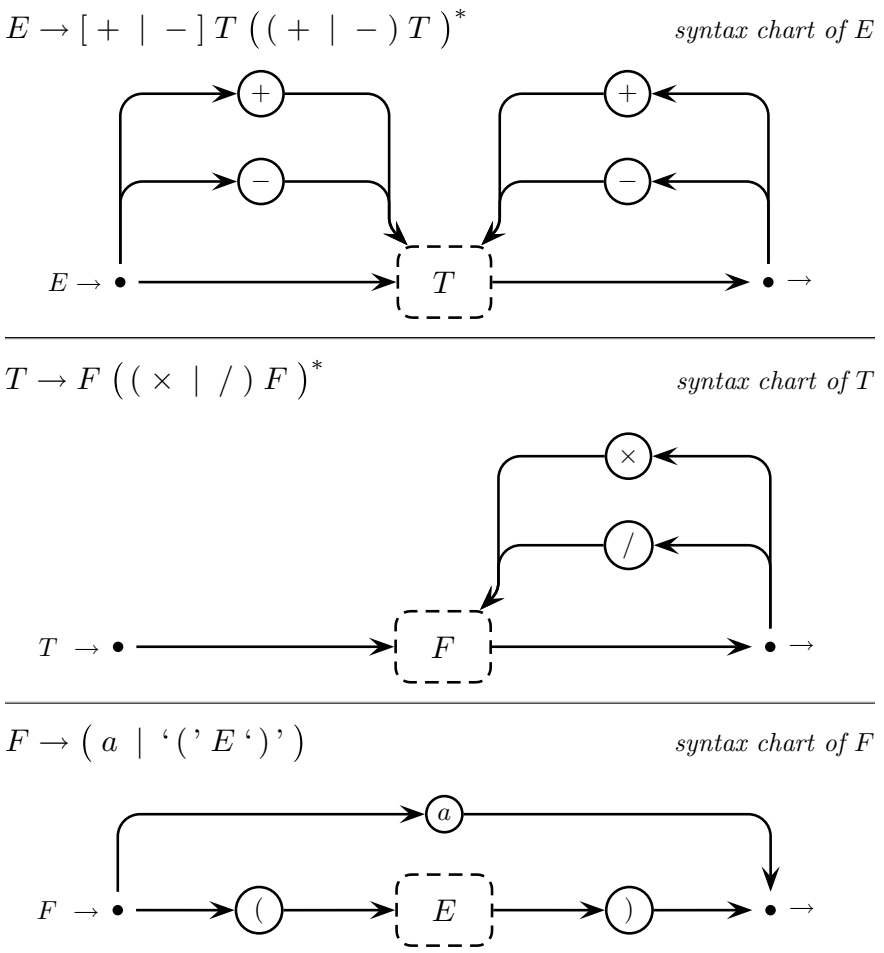


Fig. 4.13 Syntax charts equivalent to the machine net of arithmetic expressions in Fig. 4.11 on p. 235

Example 4.26 (*syntax charts of arithmetic expressions—Example* 4.24) The machines of the net (Fig. 4.11 on p. 235) are redrawn as syntax charts in Fig. 4.13. The nodes are grammar symbols with dashed and solid shapes around for nonterminals and terminals, respectively. The machine states are not drawn as graph nodes nor do they bear a label. Syntax charts are an equivalent representation of a grammar or machine net. □

In traversing a chart from entry to exit, we obtain any possible right part of the corresponding syntax rule. For instance, a few traversing paths (some with loops) in

the chart of nonterminal E are the following:

$$T \qquad +\,T \qquad T\,+\,T \qquad -\,T\,+\,T\,-\,T \qquad \ldots$$

It should be clear how to construct the syntax charts of a given *EBNF* grammar: by means of known methods, first construct the finite recognizer of each r.e. occurring as the right part of a grammar rule, then adjust the machine graphic representation to the desired convention.

4.5.2 Derivation for Machine Nets

For *EBNF* grammars and machine nets, the previous definition of derivation, which models a rule such as $E \to T^*$ as an infinite set of *BNF* alternative rules $E \to \varepsilon \mid T \mid T\,T \mid \ldots$, has shortcomings because a derivation step, e.g., $E \Rightarrow T\,T\,T$, replaces a nonterminal E by a string of possibly *unbounded* length; thus a multi-step computation inside a machine is equated to just one derivation step. For parsing applications, a more analytical definition is needed to split such a large step into a series of state transitions.

We recall that a *BNF* grammar is *right-linear* (*RL*) (see Definition 2.74 on p. 85) if every rule has the form $A \to u\,B$ or $A \to \varepsilon$, where $u \in \Sigma^*$ and $B \in V$ (B may coincide with A). Every finite-state machine can be represented by an equivalent *RL* grammar (see Sect. 3.44 on p. 129) that has machine states as nonterminal symbols.

4.5.2.1 Right-Linearized Grammar

For each machine $M_A \in \mathcal{M}$, it is straightforward to write the *RL* grammar equivalent with respect to the regular language $R\,(M_A,\ 0_A) \subseteq (\,\Sigma\ \cup\ V\,)^*$, to be denoted as $\hat{G}_A$. The nonterminals of $\hat{G}_A$ are the states of Q_A, so its axiom is 0_A. If an arc $p_A \xrightarrow{X} r_A$ is in the transition function δ of M_A, in $\hat{G}_A$ there exists a rule $p_A \to X\,r_A$ (for any terminal or nonterminal symbol $X \in \Sigma\ \cup\ V$), and if p_A is a final state of M_A, the empty rule $p_A \to \varepsilon$ exists.

Notice that X may be a nonterminal B, therefore a rule of $\hat{G}_A$ may have the form $p_A \to B\,r_A$, which is still *RL* since the first (leftmost) symbol of the right part is viewed as a "terminal" symbol for grammar $\hat{G}_A$. With this provision, the identity $L\left(\hat{G}_A\right) = R\,(M_A,\ 0_A)$ clearly holds.

Next, for every *RL* grammar of the net, we replace by 0_B every symbol $B \in V$ occurring in a rule such as $p_A \to B\,r_A$, thus obtaining rules of form $p_A \to 0_B\,r_A$. The resulting *BNF* grammar is denoted $\hat{G}$ and is named the *right-linearized grammar* of the net: it has terminal alphabet Σ, nonterminal set Q and axiom 0_S.

The rule right parts have length zero or two, and they may contain two nonterminal symbols, thus the grammar is *not RL*. Obviously, grammars $\hat{G}$ and G are equivalent, i.e., they both generate language $L\,(G)$.

Example 4.27 (*right-linearized grammar $\hat{G}$ of the running example*)

$$\hat{G}_E \begin{cases} 0_E \to 0_T\, 1_E \mid \varepsilon \\ 1_E \to 0_T\, 1_E \mid \varepsilon \end{cases} \qquad \hat{G}_T \begin{cases} 0_T \to a\, 3_T \mid \text{`('}\, 1_T \\ 1_T \to 0_E\, 2_T \\ 2_T \to \text{`)'}\, 3_T \\ 3_T \to \varepsilon \end{cases}$$

As said, in the right-linearized grammars we choose to name nonterminals by their alias states. An instance of non-*RL* rule is $1_T \to 0_E\, 2_T$. □

By using the right-linearized grammar $\hat{G}$ instead of G, we obtain derivations the steps of which are elementary state transitions instead of multi-step transitions on a machine. An example should suffice to clarify.

Example 4.28 (*derivation in a right-linearized grammar*) For grammar G (in Fig. 4.12 on p. 237) the "classical" leftmost derivation:

$$E \underset{G}{\Rightarrow} T\, T \underset{G}{\Rightarrow} a\, T \underset{G}{\Rightarrow} a\, (\, E\,) \underset{G}{\Rightarrow} a\, (\,) \tag{4.6}$$

is expanded into the series of truly atomic derivation steps of the right-linearized grammar $\hat{G}$:

$$\begin{array}{lllll} 0_E & \underset{\hat{G}}{\Rightarrow} 0_T\, 1_E & \underset{\hat{G}}{\Rightarrow} a\, 1_E & \underset{\hat{G}}{\Rightarrow} a\, 0_T\, 1_E \\ & \underset{\hat{G}}{\Rightarrow} a\, (\, 1_T\, 1_E & \underset{\hat{G}}{\Rightarrow} a\, (\, 0_E\, 2_T\, 1_E & \underset{\hat{G}}{\Rightarrow} a\, (\, \varepsilon\, 2_T\, 1_E \\ & \underset{\hat{G}}{\Rightarrow} a\, (\,)\, 3_T\, 1_E & \underset{\hat{G}}{\Rightarrow} a\, (\,)\, \varepsilon\, 1_E & \underset{\hat{G}}{\Rightarrow} a\, (\,)\, \varepsilon \\ & = a\, (\,) \end{array} \tag{4.7}$$

which show how it works. □

We may also consider reductions, such as $a\, (\, 1_T\, 1_E \rightsquigarrow a\, 0_T\, 1_E$.

4.5.2.2 Call Sites and Machine Activation

An arc $q_A \xrightarrow{B} r_A$ with nonterminal label B is also named a *call site* for machine M_B, and its destination r_A is the corresponding *return state*. Parsing can be viewed as a process that upon reaching a call site activates the specified machine. The activated machine performs scanning operations and further calls on its own graph, until it reaches a final state, where it performs a reduction and finally goes back to the return state. The axiomatic machine M_S is initially activated by the program that invokes the parser.

When we observe a left-linearized derivation, we see that at every step the nonterminal suffix of the derived string contains the current state of the active machine,

followed by the return states of the suspended ones. Such states are ordered from right to left according to the machine activation sequence.

Example 4.29 (*derivation and machine return point*) In derivation (4.7), observe $0_E \stackrel{*}{\Rightarrow} a\,(\,0_E\,2_T\,1_E$: machine M_E is active and its current state is 0_E; previously, machine M_T was suspended and will resume in state 2_T; an earlier activation of M_E was also suspended and will resume in state 1_E. □

4.5.3 Initial and Look-Ahead Characters

The deterministic parsers to be presented implement a pushdown automaton (*PDA*). To choose the next move, such parsers use the stack contents (actually just the top symbol), the automaton state and the next input character, which is the *token* or lexeme returned by the lexical analyzer. To make the choice deterministic, they rely on the information obtained by preprocessing the grammar, which consists of the sets of the tokens that may be encountered for each possible move. For instance, suppose in the current state p_A there are two outgoing arcs:

$$q_A \stackrel{B}{\longleftarrow} p_A \stackrel{C}{\longrightarrow} r_A$$

To choose the arc to follow, it helps to know which tokens may start languages $L\,(0_B)$ and $L\,(0_C)$. Such token sets are named *initial*.

Moreover, another situation occurs in bottom-up parsing, when the *PDA* completely scans the right part α of a rule $A \to \alpha$ and thus reaches the final state f_A of machine M_A. Then, to confirm that the reduction is due, it helps to know which tokens may follow nonterminal A in a derivation. Concretely, consider the machine net in Fig. 4.14:

After scanning substring $a\,b$, the *PDA* can be either in state f_A or in state f_B, and the reduction to be applied can be either $a\,b \rightsquigarrow A$ or $b \rightsquigarrow B$. How do we choose? By checking whether the next input token is character c or d, respectively. To enable the *PDA* to perform such checks, we pre-calculate the token sets and we assume that the strings are always followed by the end-of-text token. Nonterminal A can be followed by $\{\ c,\ \dashv\ \}$ and nonterminal B by $\{\ d\ \}$. Such sets of terminals are named *follow* sets.

Furthermore, the information provided by the follow sets can be made more accurate. We observe that nonterminal A cannot be followed by terminal c if the input string is $e\,a\,b$. A sharper definition should take into account the previous computation, i.e., the *PDA* configuration. Thus the exact sets, to be named *look-ahead* sets, associated with each configuration, are as follows: set $\{\ c\ \}$ for nonterminal A with configuration 1_S and set $\{\ \dashv\ \}$ for nonterminal A with configuration f_S; while for the nonterminal B with configuration 3_S (the only one existing), the look-ahead set is $\{\ d\ \}$.

The sets of initial and look-ahead tokens are now formalized by means of procedural definitions.

EBNF grammar

$$S \to A\,c \mid a\,B\,d \mid e\,A \qquad A \to a\,b \qquad B \to b$$

machine net

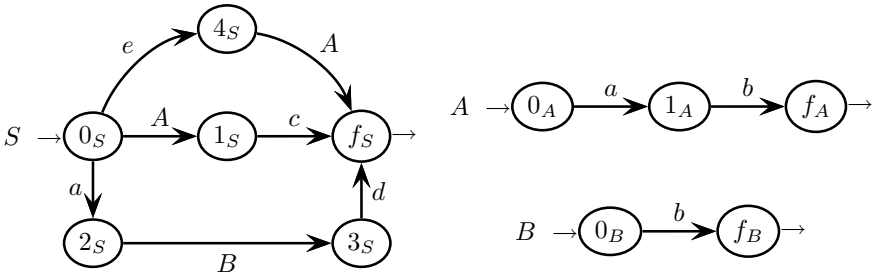


Fig. 4.14 Grammar and network for illustrating the computation of follow sets

Initials

We need to define the set of the *initial* characters for the strings recognized starting from a given state.

Definition 4.30 (*set Ini of initials*) The set $Ini \subseteq \Sigma$ of a state q_A is:

$$Ini\,(q_A) = Ini\left(L\,(q_A)\right) = \left\{\, a \in \Sigma \mid \;\; a\;\Sigma^* \;\cap\; L\,(q_A) \neq \emptyset \,\right\}$$

□

Notice that set *Ini* may not contain the null string ε. A set $Ini\,(q_A)$ is empty if, and only if, it holds $L\,(q_A) = \{\,\varepsilon\,\}$, i.e., the empty string is the only one generated starting from state q_A.

Let symbol a be a terminal, symbols A and B be possibly identical nonterminals, and take two states q_A and r_A of the same machine M_A. Set *Ini* is computed by applying the following logical clauses until a fixed point is reached.

It holds $a \in Ini\,(q_A)$ if, and only if, one or more of the three following conditions are met:

1. $\exists$ arc $q_A \xrightarrow{a} r_A$
2. $\exists$ arc $q_A \xrightarrow{B} r_A \;\wedge\; a \in Ini\,(0_B)$, excluding the case $0_A \xrightarrow{A} r_A$
3. $\exists$ arc $q_A \xrightarrow{B} r_A \;\wedge\; L\,(0_B)$ is nullable $\wedge\; a \in Ini\,(r_A)$

The restriction in 2 excludes an immediate left recursion, which does not add anything new. For the running example: $Ini\,(0_E) = Ini\,(0_T) = \{\,a,\;\text{`(`}\,\}$.

Look-Ahead

As said, a set of look-ahead characters is associated with each *PDA* parser configuration. Without going into details, we assume that such a configuration is encoded by one or more machine states. Remember that many such states are possibly entered by the parser after processing some text. In what follows, we consider the look-ahead set of each state separately.

A pair $\langle\, state,\ token \,\rangle$ is named a *candidate* (also known as *item*). More precisely, a candidate is a pair $\langle\, q_B,\ a \,\rangle$ in $Q \times \left(\Sigma \cup \{ \dashv \} \right)$. The intended meaning is that token a is a legal look-ahead for the current activation of machine M_B in state q_B. When parsing begins, the initial state of the axiomatic machine M_S is encoded by candidate $\langle\, 0_S,\ \dashv \rangle$, which says that the end-of-text character $\dashv$ is expected when the entire input is reduced to the axiom S. To calculate the candidates for a given grammar or machine net, we use a function traditionally named *closure*.

Definition 4.31 (*closure function*) Let C be a set of candidates. The closure of C is the function defined by applying the following clause (4.8):

$$\begin{aligned} &closure\,(C) = C \\ &\langle\, 0_B,\ b \,\rangle \in closure\,(C) \quad \text{if} \begin{cases} \exists \text{ candidate } \langle\, q,\ a \,\rangle \in C & \textbf{and} \\ \exists \text{ arc } q \xrightarrow{B} r \text{ in } \mathcal{M} & \textbf{and} \\ b \in Ini\left(L\,(r) \cdot a \right) \end{cases} \end{aligned} \tag{4.8}$$

until a fixed point is reached. □

Thus, function *closure* computes the set of the machines that are reached from a given state q through one or more invocations, without any intervening state transition. For each reachable machine M_B, represented by the initial state 0_B, the function returns also any input character b that can legally occur when the machine terminates; such a character is part of the look-ahead set. Notice the effect of term $Ini\left(L\,(r) \cdot a \right)$ when state r is final, i.e., when $\varepsilon \in L\,(r)$ holds: character a is included in the set.

When clause (4.8) terminates, the closure of C contains a set of candidates, with some of them possibly associated with the same state q, as follows:

$$\{\, \langle\, q,\ a_1 \,\rangle,\ \ldots,\ \langle\, q,\ a_k \,\rangle \,\}$$

For conciseness we group together such candidates and we write:

$$\left\langle\, q,\ \{\, a_1,\ \ldots,\ a_k \,\} \,\right\rangle$$

instead of $\{\, \langle\, q,\ a_1 \,\rangle,\ \ldots,\ \langle\, q,\ a_k \,\rangle \,\}$. The collection $\{\, a_1,\ a_2,\ \ldots,\ a_k \,\}$ is termed the *look-ahead set* of state q in the closure of C. By construction the look-ahead set of a state is never empty. To simplify the notation of singleton look-ahead, we write $\langle\, q,\ b \,\rangle$ instead of $\left\langle\, q,\ \{\, b \,\} \,\right\rangle$.

Example 4.32 (*closure computation*) We list a few values computed by the closure function for the grammar of Example 4.25 (Fig. 4.12 on p. 237):

C	*function closure* C		
$\langle 0_E, \dashv \rangle$	$\langle 0_E, \dashv \rangle$	$\langle 0_T, \{ a, \text{`(`}, \dashv \} \rangle$	
$\langle 1_T, \dashv \rangle$	$\langle 1_T, \dashv \rangle$	$\langle 0_E, \text{`)`} \rangle$	$\langle 0_T, \{ a, \text{`(`}, \text{`)`} \} \rangle$

For the top row we explain how the closure of candidate $\langle 0_E, \dashv \rangle$ is obtained. First, it trivially contains itself. Second, since arc $0_E \xrightarrow{T} 1_E$ calls machine M_T with initial state 0_T, by clause (4.8) the closure contains the candidate with state 0_T and with look-ahead set computed as $Ini\left(L\,(1_E)\,\dashv\right)$. Since language $L\,(1_E)$ is nullable, character $\dashv$ is in the set, as well as the initials of $L\,(1_E)$, i.e., characters a and `(`. Since no nonterminal arc departs from 0_T, the fixed point is now reached, hence a further application of closure would be unproductive.

In the bottom row the result contains $\langle 1_T, \dashv \rangle$ by reflexivity, and $\langle 0_E, \text{`)`} \rangle$ because arc $1_T \xrightarrow{E} 2_T$ calls machine M_E and $Ini\left(L\,(2_T)\right) = \{ \text{`)`} \}$. The result contains also $\langle 0_T, \{ a, \text{`(`}, \text{`)`} \} \rangle$ because arc $0_E \xrightarrow{T} 1_E$ calls machine M_T, language $L\,(1_E)$ is nullable and it holds $\{ a, \text{`(`}, \text{`)`} \} = Ini\left(L\,(1_E)\,\text{`)`}\right)$. Three applications of clause (4.8) are needed to reach the fixed point. □

The closure function is a basic component for constructing the *ELR* (1) parsers to be next presented.

4.6 Bottom-Up Deterministic Analysis

We outline the construction of the deterministic bottom-up algorithms that is widely applied to automatically implement parsers for a given grammar. The formal condition that allows such a parser to work was defined and named *LR* (k) by its inventor D. E. Knuth [16]. The parameter $k \geq 0$ specifies how many consecutive characters are inspected to let the parser move deterministically. Since we consider *EBNF* grammars and we assume to be $k = 1$ (in accordance with common use), the condition is referred to as *ELR* (1). The language family accepted by the parsers of type *LR* (1) (or *ELR* (1)) is exactly the family *DET* of deterministic context-free languages; more of that in Sect. 4.9.

The *ELR* (1) parsers implement a deterministic automaton equipped with a pushdown stack and with a set of internal states, to be named *macro-states* (for short m-states) to prevent confusion with the machine net states. An m-state consists of a set of candidates (on p. 243). The automaton performs a series of moves of two types. A *shift* move reads an incoming character, i.e., token or lexeme, and applies a state-transition function to compute the next m-state; then the token and the next

m-state are pushed onto the stack. A *reduction* (or *reduce*) move applies as soon as the sequence of topmost stack symbols matches the sequence of character labels of a recognizing path in a machine M_A, upon the condition that the current token is present in the current look-ahead set.

The first condition for determinism is that, in every parser configuration, if a shift move is permitted then a reduction move is impossible. The second is that in every configuration at most one reduction move be possible.

A reduction move is also in charge of growing the fragment of the syntax tree that the parser has so far computed. More precisely, in general the tree is a *forest*, since multiple subtrees may be still disconnected from the root that will be eventually created.

To update the stack, a reduction move performs several actions: it pops the matched topmost part (the so-called *handle*) of the stack, then it pushes onto the stack the nonterminal symbol recognized and the next m-state. Notice that a reduction does not consume the current token, which will be checked and consumed by the subsequent shift move that scans the input.

The *PDA* accepts the input string if the last move reduces the stack to the initial configuration and the input has been entirely scanned. The latter condition can be expressed by saying that the special end-marker character $\dashv$ is the current input token.

Since the parser construction is quite complex to understand in its full generality, we start from a few simple cases.

4.6.1 From Finite Recognizers to Bottom-Up Parser

To start from known ground, we view an r.e. e as an *EBNF* grammar that has only one nonterminal and one rule $S \to e$. The methods of Chap. 3, especially that by Berry and Sethi, construct a finite deterministic automaton that recognizes the regular language $L(S) = L(e)$. Next, we slightly generalize such grammar by introducing several nonterminals, while still remaining within the domain of regular languages. Hierarchical lists (Chap. 2 p. 29) are a natural example, where at each level a different nonterminal specified by an r.e. generates a list.

Example 4.33 (*bottom-up parser of hierarchical lists*) A sentence is a list of lists, with the outer list S using letter s as separator, and with the inner lists I using letter a as element and letter t as separator as well as end-marker. All such lists are not empty. The *EBNF* grammar G_1 and machine net $\mathcal{M}_1$ are shown in Fig. 4.15. Although language $L(G_1)$ is regular, the methods of Chap. 3 do not allow to construct a recognizer, unless we eliminate nonterminal I to derive a one rule grammar, which we do not intend to do. Instead we construct a deterministic *PDA* directly from the network. To find the information to be kept in a *PDA* state, consider the input string $a\ t\ a\ t\ s\ a\ t\ \dashv$. The automaton is in the initial state 0_S, whence an I-labeled arc originates: this signifies the *PDA* has to recognize a string deriving from nonterminal I. Therefore the automaton enters the initial state of machine M_I without reading

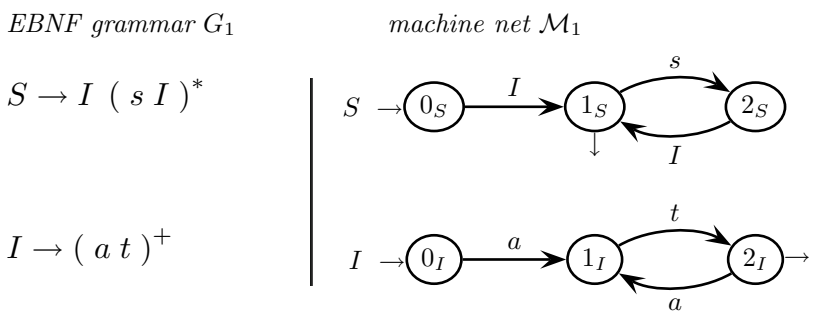


Fig. 4.15 Grammar G_1 and network $\mathcal{M}_1$ for Example 4.33 (two-level list)

any character, i.e., by a spontaneous transition. As conventional with spontaneous moves, we say that the *PDA* is in one of the states $\{\ 0_S,\ 0_I\ \}$ and we name this set a macro-state (m-state). The reader should notice the similarity between the concepts of m-state and of accessible state subset of a finite nondeterministic machine (Chap. 3 on p. 145).

The *PDA* stores the current m-state $\{\ 0_S,\ 0_I\ \}$ on the stack for reentering it in the future, after a series of transitions. Then, it performs a *shift* operation associated with the transition $0_I \xrightarrow{a} 1_I$ of machine M_I. The operation shifts the current token a and pushes the next m-state $\{\ 1_I\ \}$ onto the stack. Then, after transition $1_I \xrightarrow{t} 2_I$, the stack top is $\{\ 2_I\ \}$ and the current token is t.

These and the subsequent moves until recognition are listed in Table 4.6. At each step, the first row shows the input tape and the second one shows the stack contents; the already scanned prefix of the input is shaded; since the stack bottom $\{\ 0_S,\ 0_I\ \}$ does not change, it is represented only once in the first column; steps 1 and 14 perform the initialization and the acceptance test, respectively; the other steps perform shift or reduction as commented aside.

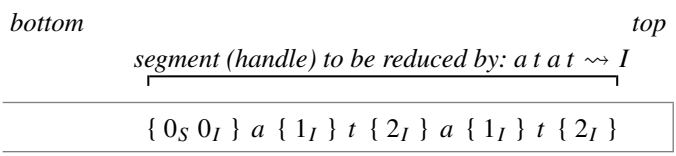
Returning to the parsing process, two situations require additional information to be kept in the m-states for the process to be deterministic. The first occurs when there is a choice between a *shift* and a *reduction*, known as potential *shift-reduce conflict*; and the second, when the *PDA* has decided for a reduction but the handle still has to be identified, known as potential *reduce-reduce* conflict.

The first time needing to choose between shift (of a) or reduction (to I), occurs at step 4, since the m-state $\{\ 2_I\ \}$ contains the final state of machine M_I. To arbitrate the conflict, the *PDA* makes use of *look-ahead set* (defined on p. 243), which is the set of tokens that may follow the reduction under consideration. The look-ahead set of m-state $\{\ 2_I\ \}$ contains the tokens s and $\dashv$, and since the next token a differs from them, the *PDA* rules out a reduction to I and shifts token a. After one more shift on t, the next shift/reduce choice occurs at step 6; the reduction to I is selected because the current token s is in the look-ahead set of state 2_I. To locate the handle, the *PDA* looks for a topmost stack segment that is delimited on the left by a stack symbol containing the initial state of machine M_I. In this example there is just the following

Table 4.6 Parsing trace of string $x = a\,t\,a\,t\,s\,a\,t\ \dashv$ (Example 4.15 on p. 246)

stack bottom	string to be parsed (with end-marker) and stack contents								effect after	#
	1	2	3	4	5	6	7			
	a	t	a	t	s	a	t	$\dashv$	initialization of the stack	1
	a	t	a	t	s	a	t	$\dashv$	terminal shift on a: $0_I \xrightarrow{a} 1_I$	2
	a $\{1_I\}$									
	a	t	a	t	s	a	t	$\dashv$	terminal shift on t: $1_I \xrightarrow{t} 2_I$	3
	a $\{1_I\}$	t $\{2_I\}$								
	a	t	a	t	s	a	t	$\dashv$	(do not reduce)	4
	a $\{1_I\}$	t $\{2_I\}$	a $\{1_I\}$						terminal shift on a: $2_I \xrightarrow{a} 1_I$	
	a	t	a	t	s	a	t	$\dashv$	terminal shift on t: $1_I \xrightarrow{t} 2_I$	5
	a $\{1_I\}$	t $\{2_I\}$	a $\{1_I\}$	t $\{2_I\}$						
	I				s	a	t	$\dashv$	since s follows I reduce $a\,t\,a\,t \rightsquigarrow I$	6
$\{0_S\,0_I\}$	I				s	a	t	$\dashv$	nonterminal shift on I: $0_S \xrightarrow{I} 1_S$	7
	I $\{1_S\}$									
	I				s	a	t	$\dashv$	(do not reduce)	8
	I $\{1_S\}$				s $\{2_S\,0_I\}$				terminal shift on s: $1_S \xrightarrow{s} 2_S$	
	I				s	a	t	$\dashv$	terminal shift on a: $0_I \xrightarrow{a} 1_I$	9
	I $\{1_S\}$				s $\{2_S\,0_I\}$	a $\{1_I\}$				
	I				s	a	t	$\dashv$	terminal shift on t: $1_I \xrightarrow{t} 2_I$	10
	I $\{1_S\}$				s $\{2_S\,0_I\}$	a $\{1_I\}$	t $\{2_I\}$			
	I				s	I		$\dashv$	since $\dashv$ follows I reduce $a\,t \rightsquigarrow I$	11
	I $\{1_S\}$				s $\{2_S\,0_I\}$					
	I				s	I		$\dashv$	nonterminal shift on I: $2_S \xrightarrow{I} 1_S$	12
	I $\{1_S\}$				s $\{2_S\,0_I\}$	I $\{1_S\}$				
	S							$\dashv$	since $\dashv$ follows I reduce $I\,s\,I \rightsquigarrow S$	13
	the input string is reduced to the axiom S							$\dashv$	stop and accept	14
	the stack contains only the axiom $\{0_S\,0_I\}$									

possibility:



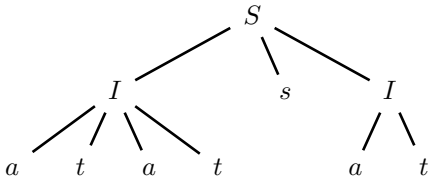
But more complex situations occur in other languages.

The reduction $a\ t\ a\ t \rightsquigarrow I$ denotes a subtree, the first part of the syntax tree produced by the parser. To make the stack contents consistent after reducing, the *PDA* performs a *nonterminal shift*, which is a sort of shift move where a nonterminal is shifted, instead of a terminal. The justification is that a string that derives from nonterminal I has been found, hence parsing resumes from m-state $\{\ 0_S,\ 0_I\ \}$ and applies the corresponding transition $0_S \xrightarrow{I} 1_S$ of machine M_S.

In Table 4.6, whenever a reduction to a nonterminal, say I, is performed, the covered input substring is replaced by I. In this way, the nonterminal shift move that follows a reduction appears to read such a nonterminal as if it were an input symbol.

In the steps 8–9, after shifting separator s, a new inner list (deriving from I) is expected in the state 2_S; this causes the initial state 0_I of M_I to be united with 2_S to form a new m-state. A new situation occurs in the steps 10–12. Since $\dashv$ belongs to the look-ahead set of state 2_I, the reduction $\ldots \rightsquigarrow I$ is chosen; furthermore, the topmost stack segment congruent with a string in the language $L\ (0_I)$ is found to be string $a\ t$ (disregarding the intervening m-states in the stack).

Step 12 shifts nonterminal I and produces a stack where a reduction is applied to nonterminal S. In the last *PDA* configuration, the input has been entirely consumed and the stack contains only the initial m-state. Therefore, the parser stops and accepts the string as valid, with the syntax tree below:



which has been grown bottom-up by the sequence of reduction moves. □

Unfortunately, finding the correct reduction may be harder than in the previous example. The next example shows the need for more information in the m-states, to enable the algorithm to uniquely identify the handle.

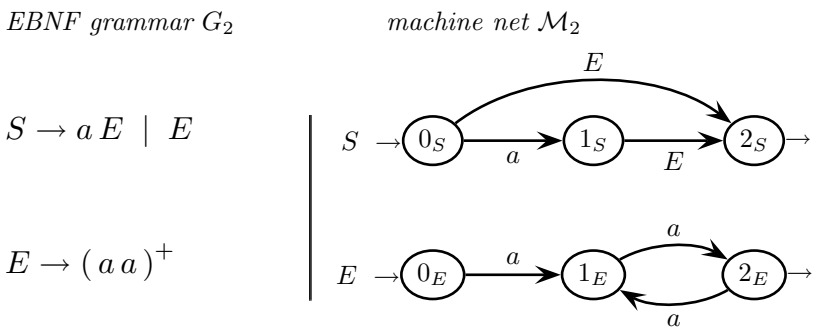


Fig. 4.16 Grammar G_2 and network $\mathcal{M}_2$ for Example 4.34

Example 4.34 (*grammar requiring more information to locate handles*) In this contrived example, machine M_S of net $\mathcal{M}_2$ (see Fig. 4.16) recognizes by separate computations the strings $L\,(0_E) = (a\,a)^+$ of even length and those of odd length $a \cdot L\,(1_S)$. After processing string $a\,a\,a$, the stack configuration of the *PDA* is as below:

	top
	$\{\ 0_S\ 0_E\ \}\ a\ \{\ 1_S\ 1_E\ 0_E\ \}\ a\ \{\ 2_E\ 1_E\ \}\ a\ \{\ 1_E\ 2_E\ \}$

An essential ability of *ELR* (1) parsers is to carry on multiple tentative analyzes at once. After shifting a, the presence of three states in m-state $\{\ 1_S,\ 1_E,\ 0_E\ \}$ says that the *PDA* has found three different computations for analyzing the first token a:

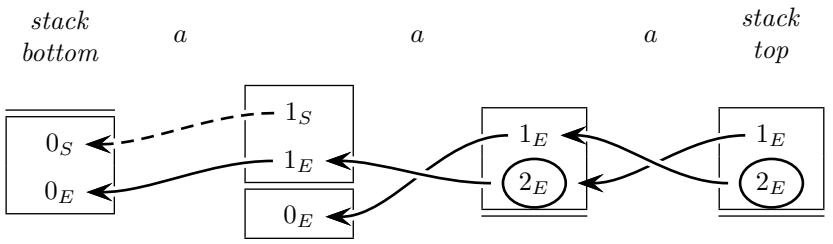
$$0_S \xrightarrow{a} 1_S \qquad 0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \qquad 0_S \xrightarrow{a} 1_S \xrightarrow{\varepsilon} 0_E$$

The *PDA* will carry on the multiple candidates until one, and only one, of them reaches a final state that commands a reduction; at that moment all remaining attempts are abandoned.

Observe again the stack: since the m-states are mathematical sets, the two m-states $\{\ 2_E,\ 1_E\ \}$ and $\{\ 1_E,\ 2_E\ \}$ are identical. Let now the terminator $\dashv$ be the next token; since state 2_E is final, a reduction $\ldots \rightsquigarrow E$ is commanded, which raises the problem of locating the handle in the stack. Should the reduction start from slot $\{\ 0_S,\ 0_E\ \}$ or slot $\{\ 1_S,\ 1_E,\ 0_E\ \}$? Clearly there is not enough information in the stack to make a decision, which ultimately depends on the parity class—odd versus even—of the string hitherto analyzed.

Different ways out of this uncertainty exist, and here we show how to use a linked list to trace simultaneous computation threads. In the stack, a pointer chain (solid arrows) that originates from each non-initial state of machine M_E reaches

the appropriate initial state of the same machine. Similarly, a pointer chain (dashed arrows) is created for the transitions of machine M_S. For evidence, the final states (here only 2_E) are encircled and a division separates the initial and non-initial states (if any):



With such structure, the *PDA*, starting from the top final state 2_E, locates the handle $a\ a \rightsquigarrow E$ by following the chain $2_E \rightarrow 1_E \rightarrow 0_E$. □

At first glance, the parser enriched with pointers may appear to trespass the power of an abstract pushdown automaton, but a closer consideration shows the contrary. Each pointer that originates in the stack position i points to an item present in position $i - 1$. The number of such items has a bound that does not depend on the input string, but only on the grammar: an m-state can contain at most all the network states (here we disregard the presence of look-ahead information, which is also finite). To select one out of the states present in an m-state, we need a finite number of pointer values. Therefore a stack element is a collection of pairs (state, pointer) taken out of a finite number of possible values. The collections can be viewed as the symbols of the stack alphabet of a *PDA*.

After this informal illustration of several features of deterministic parsers, in the next sections we formalize their construction.

4.6.2 Construction of *ELR* (1) Parsers

Given an *EBNF* grammar represented by a machine net, we show how to construct an *ELR* (1) parser if certain conditions are met. The method operates in three phases:

1. From the net we construct a *DFA*, to be called a *pilot* automaton.[13] A pilot state, named *macro-state* (m-state), includes a non-empty set of candidates, i.e., of pairs of states and terminal tokens (look-ahead set).

[13] For the readers acquainted with the classical theory of *LR* (1) parsers, we give the terminological correspondences. The pilot is named "recognizer of viable *LR* (1) prefixes" and the candidates are "*LR* (1) items". The macro-state-transition function we denote by theta (to avoid confusion with the state-transition function of the network machine denoted delta) is traditionally named "go to" function.

2. The pilot is examined to check the conditions for deterministic parsing, by inspecting the components of each m-state and the arcs that go out from it. Three types of failure may occur: a *shift-reduce* conflict signifies that both a shift and a reduction are possible in a parser configuration; a *reduce-reduce* conflict signifies that two or more reductions are similarly possible; and a *convergence* conflict occurs when two different parser computations that share a look-ahead token, lead to the same machine state.
3. If the previous test is passed, then we construct the deterministic *PDA*, i.e., the parser. The pilot *DFA* is the finite-state controller of *PDA*, to which it is necessary to add certain features for pointer management, which are needed to cope with the reductions having handles of unbounded length.

At last, it would be a simple exercise to encode the *PDA* in a programming language of some kind.

Now, we prepare to formally present the procedure (Algorithm 4.35) for constructing the pilot automaton. Here are a few preliminaries and definitions. The pilot is a *DFA*, named $\mathcal{P}$, defined by the following entities:

- the set R of m-states
- the *pilot alphabet* is the union $\Sigma \cup V$ of the terminal and nonterminal alphabets, to be also named the *grammar symbols*
- the initial m-state, I_0, is the set: $I_0 = closure\ (\langle 0_S, \dashv \rangle)$, where the closure function was defined on p. 243
- the m-state set $R = \{ I_0, I_1, \dots \}$ and the state-transition function $\vartheta\colon R \times (\Sigma \cup V) \to R$ are computed starting from I_0 as next specified

Let $\langle p_A, \rho \rangle$, with $p_A \in Q$ and $\rho \subseteq \Sigma \cup \{ \dashv \}$, be a candidate and X be a grammar symbol. The *shift* under X, qualified as *terminal/nonterminal* depending on the nature of X, is as follows:

$$\begin{cases} \vartheta\ (\langle p_A, \rho \rangle, X) = \langle q_A, \rho \rangle & \text{if the arc } p_A \xrightarrow{X} q_A \text{ exists} \\ \text{the empty set} & \text{otherwise} \end{cases}$$

For a set C of candidates, the shift under a symbol X is the union of the shifts of the candidates in C:

$$\vartheta\ (C, X) = \bigcup_{\forall \text{ candidate } \gamma \in C} \vartheta\ (\gamma, X)$$

Algorithm 4.35 (*construction of the ELR* (1) *pilot graph*) Computation of the m-state set $R = \{ I_0, I_1, \ldots \}$ and of the state-transition function ϑ (the algorithm uses auxiliary variables R' and I'):

$R' := \{ I_0 \}$ - - prepare the initial m-state I_0
- - loop that updates the m-state set and the state-transition function
do
  $R := R'$ - - update the m-state set R
  - - loop that computes possibly new m-states and arcs
  for $\big($each m-state $I \in R$ and symbol $X \in \Sigma \cup V\big)$ **do**
    - - compute the base of an m-state and its closure
    $I' := closure\big(\vartheta\ (I,\ X)\big)$
    - - check if the m-state is not empty and add the arc
    if $\big(I' \neq \emptyset\big)$ **then**
      add arc $I \xrightarrow{X} I'$ to the graph of ϑ
      - - check if the m-state is a new one and add it
      if $\big(I' \notin R\big)$ **then**
        add m-state I' to the set R'
      end if
    end if
  end for
while $\big(R \neq R'\big)$ - - repeat if the graph has grown

□

It helps to introduce a classification for the m-states and the pilot moves.

4.6.2.1 Base, Closure and Kernel of m-State

For an m-state I the set of candidates is assigned to two classes, named base and closure. The *base* includes the non-initial candidates:

$$I_{|\,base} = \{\ \langle q,\ \pi \rangle \in I \mid \ \ q \text{ is not an initial state } \}$$

Clearly for the m-state I' computed in the algorithm, the base $I'_{|\,base}$ coincides with the pairs computed by $\vartheta\ (I,\ X)$.

The *closure* contains the remaining candidates of m-state I:

$$I_{|\,closure} = \{\ \langle q,\ \pi \rangle \in I \mid \ \ q \text{ is an initial state } \}$$

Notice that by this definition the base of the initial m-state I_0 is empty and all other m-states have a non-empty base, while their closure may be empty.

Sometimes we do not need the look-ahead sets; the *kernel* of an m-state is the projection on the first component of every candidate:

$$I_{|\,kernel} = \{\, q \in Q \mid \quad \langle\, q,\ \pi \,\rangle \in I \,\}$$

Two m-states that have the same kernel, i.e., differing just for some look-ahead sets, are called *kernel-equivalent*. Some simplified parser constructions to be later introduced rely on kernel equivalence to reduce the number of m-states. We observe that for any two kernel-equivalent m-states I and I', and for any grammar symbol X, the next m-states $\vartheta\,(I,\ X)$ and $\vartheta\,(I',\ X)$ are either both defined or neither one, and are kernel-equivalent.

The next condition that may affect determinism occurs when two states within an m-state have outgoing arcs labeled with the same grammar symbol.

Definition 4.36 (*multiple-transition property and convergence*) A pilot m-state I has the *multiple-transition property* (*MTP*) if it includes two candidates $\langle\, q,\ \pi \,\rangle$ and $\langle\, r,\ \rho \,\rangle$ with $q \neq r$, such that for some grammar symbol X both transitions $\delta\,(q,\ X)$ and $\delta\,(r,\ X)$ are defined.

Then the m-state I and the transition (i.e., pilot arc) $\vartheta\,(I,\ X)$ are called *convergent* if it holds $\delta\,(q,\ X) = \delta\,(r,\ X)$. A convergent transition has a *convergence conflict* if $\pi \cap \rho \neq \emptyset$, meaning that the look-ahead sets of the two candidates overlap. □

It is time to illustrate the previous construction and definitions by means of the running example.

Example 4.37 (*pilot graph of the running example*) The pilot graph $\mathcal{P}$ of the machine net of Example 4.25 (see Fig. 4.12 on p. 237) is shown in Fig. 4.17. Each m-state is split by a double line into the base (top) and the closure (bottom). Either part can be missing: I_0 has no base and I_2 has no closure. The look-ahead tokens are grouped by state and the final states are encircled. Macro-states I_1 and I_4 are kernel-equivalent, and we observe that any two equally labeled arcs leaving I_1 and I_4 reach two kernel-equivalent m-states. None of the m-states has the *MTP*, therefore no arc of the pilot graph is convergent. □

In the next section determinism is examined in depth and a determinism condition is formulated. A few examples illustrate the cases when the condition is satisfied or violated, in the possible ways.

4.6.3 *ELR* (1) Condition

The presence of a final candidate f_A in m-state I instructs the parser to examine a possible reduction. The look-ahead set specifies which tokens should occur next, to confirm the decision to reduce. Clearly to avoid a conflict, any such token should not label an arc going out from I. In general two or more paths can lead from the

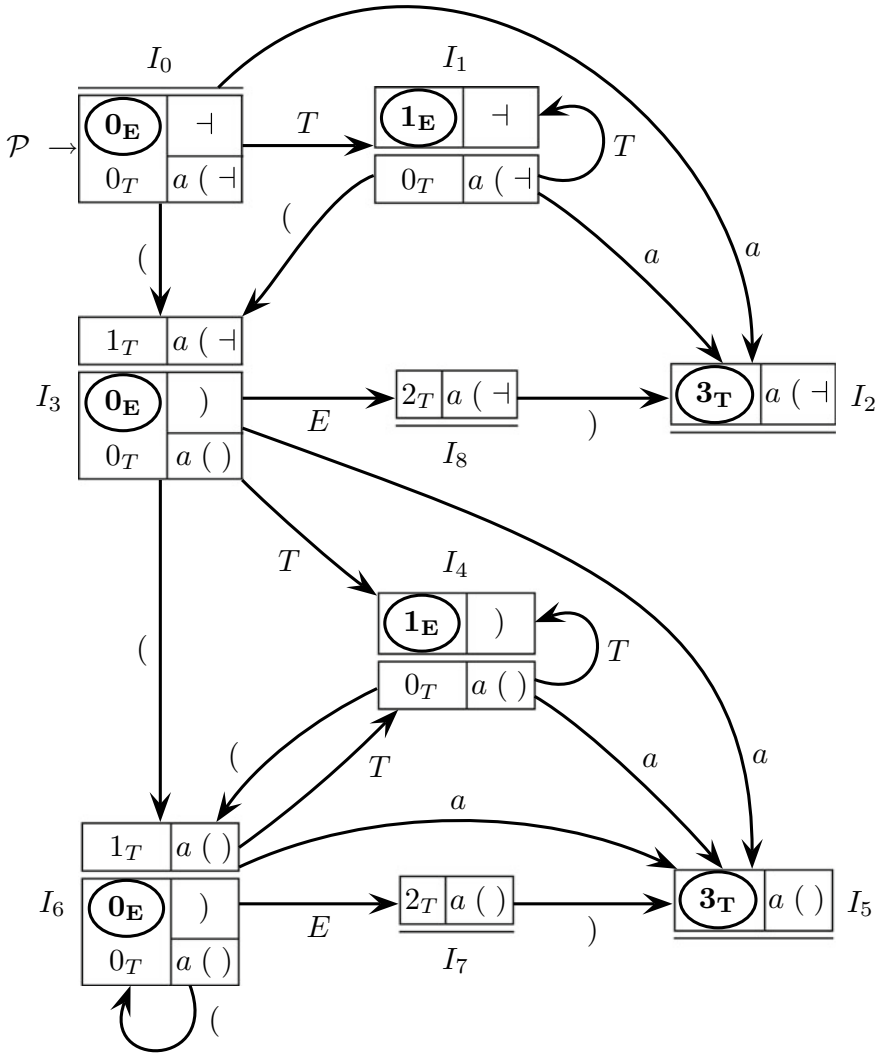


Fig. 4.17 *ELR* (1) pilot graph $\mathcal{P}$ of the machine net in Fig. 4.12

initial state of a machine to the same final state, therefore two or more reductions may be applied when the parser enters m-state I. To choose the correct reduction, the parser has to store additional information in the stack, as later explained; here too potential conflicts must be excluded. The next conditions ensure that all steps are deterministic.

Definition 4.38 (*ELR* (1) *condition*) An *EBNF* grammar or its machine net meets the condition *ELR* (1) if the corresponding pilot satisfies the two following requirements:

1. Every m-state I satisfies the next two clauses:
   no *shift-reduce conflict*:

$$\begin{gathered}\text{for all the candidates } \langle\, q,\ \pi \,\rangle \in I \text{ s.t. state } q \text{ is final} \\ \text{and for all the arcs } I \xrightarrow{a} I' \text{ that go out from } I \text{ with terminal label } a \\ \text{it must hold } a \notin \pi\end{gathered} \tag{4.9}$$

   no *reduce-reduce conflict*:

$$\begin{gathered}\text{for all the candidates } \langle\, q,\ \pi \,\rangle, \langle\, r,\ \rho \,\rangle \in I \text{ s.t. states } q, r \text{ are final} \\ \text{it must hold } \pi \ \cap\ \rho = \emptyset\end{gathered} \tag{4.10}$$

2. No transition of the pilot graph has a *convergence conflict*. □

If the grammar is purely *BNF*, then the previous condition is referred to as *LR* (1)[14] instead of *ELR* (1). We anticipate that convergence conflicts never occur in the *BNF* grammars.

The running example (Fig. 4.17) does not have either shift-reduce or reduce-reduce conflicts. First, consider requirement (1). Clause (4.9) is met in every m-state that contains a final state; for instance in I_6, the look-ahead set associated with reduction is disjoint from the labels of the outgoing arcs. Clause (4.10) is trivially met as there is not any m-state that contains two or more final states. Second, notice that no pilot arc is convergent, as already observed, so also requirement (2) is satisfied.

To clarify the different causes of conflict, we examine a few examples. To start, the contrived Example 4.39 illustrates shift-reduce conflicts.

Example 4.39 (*shift-reduce conflicts*) The grammar of Fig. 4.18 has a shift-reduce conflict in m-state I_1, where character b occurs in the look-ahead set of final state 1_A and also as the label of the arc going out to m-state 2_S. □

Next in Example 4.40, we consider a grammar with a pilot that has m-states with two reductions, some of which give rise to a reduce-reduce conflict.

Example 4.40 (*reduce-reduce conflicts in a nondeterministic language*) Consider the nondeterministic language:

$$L = \left\{\, a^n\, b^n \mid \ n \geq 1 \,\right\} \cup \left\{\, a^{2n}\, b^n \mid \ n \geq 1 \,\right\}$$

generated by the grammar below:

$$S \to A \mid B \qquad A \to a\, A\, b \mid a\, b \qquad B \to a\, a\, B\, b \mid a\, a\, b$$

Since language L is nondeterministic, every grammar that generates it necessarily has conflicts. Figure 4.19 shows the pilot graph of the grammar. The pilot graph has

[14] The original acronym comes for "Left-to-right Rightmost" [16].

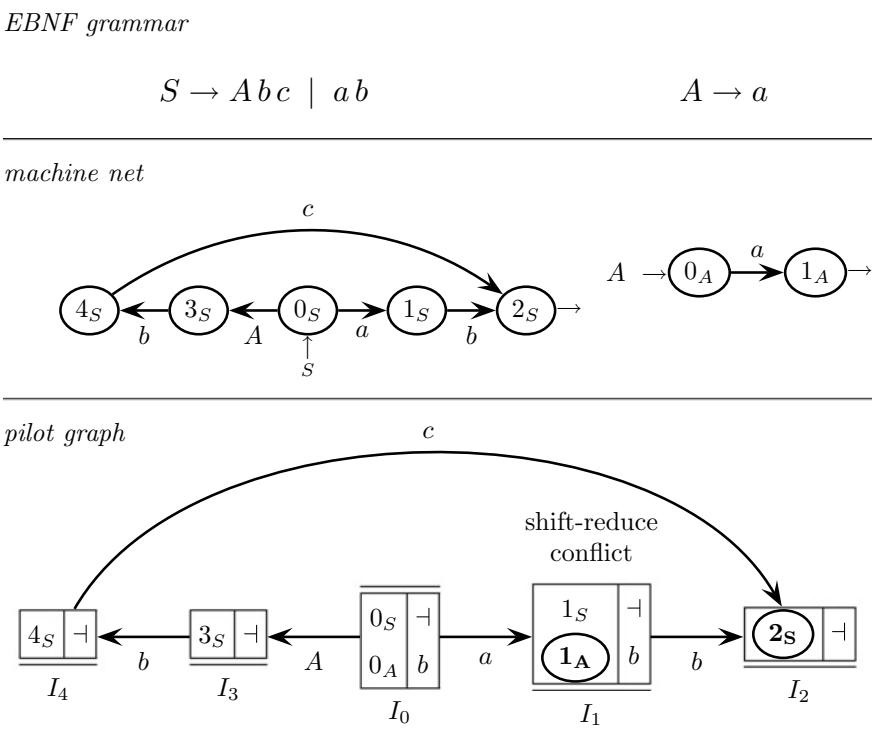


Fig. 4.18 Grammar with net and pilot with shift-reduce conflict (Example 4.39)

a reduce-reduce conflict in the m-state I_{12} since the final states 3_A and 4_B have identical look-ahead sets $\{\ b\ \}$. To understand how this affects parsing, consider the analysis of string $a\,a\,a\,a\,b\,b$: after reading its prefix $a\,a\,a\,a$, the next token b is compatible with both reductions $a\,b \rightsquigarrow A$ and $a\,a\,b \rightsquigarrow B$, which makes the analysis nondeterministic. □

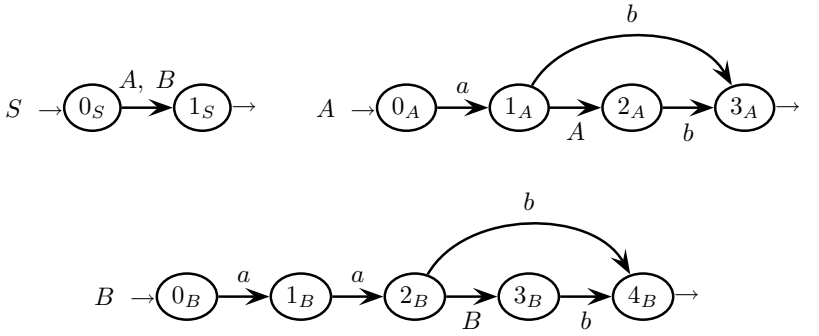
Finally in Example 4.41 we show the case of convergent transitions and convergence conflicts, which are a characteristic of *EBNF* grammars not possessed by purely *BNF* grammars.

Example 4.41 (*convergence conflicts*) Consider the *EBNF* grammar and the corresponding machines in Fig. 4.20. To illustrate the notion of convergent transition with and without conflict, we refer to the pilot in Fig. 4.20. Macro-state $I_5 = \{\ \langle\, 2_S,\ \dashv \,\rangle\ ,\ \langle\, 4_S,\ e\,\rangle\ \}$ satisfies the multiple-transition property, since the two arcs $\delta\,(2_S,\ c)$ and $\delta\,(4_S,\ c)$ are present in the net of Fig. 4.20. In the pilot graph, the two arcs are fused into the arc $\vartheta\,(I_5,\ c)$, which is therefore convergent. Anyway there is not any convergence conflict as the two look-ahead sets $\{\ \dashv\ \}$ and $\{\ e\ \}$ are disjoint. Similarly m-state I_8 has the *MTP*, but this time there is a convergence conflict, because the look-ahead set is the same, i.e., $\{\ e\ \}$. The pilot in Fig. 4.20 does not

EBNF grammar

$$S \to A \mid B \qquad A \to a\ (b \mid A\,b) \qquad B \to a\,a\ (b \mid B\,b)$$

machine net



pilot graph

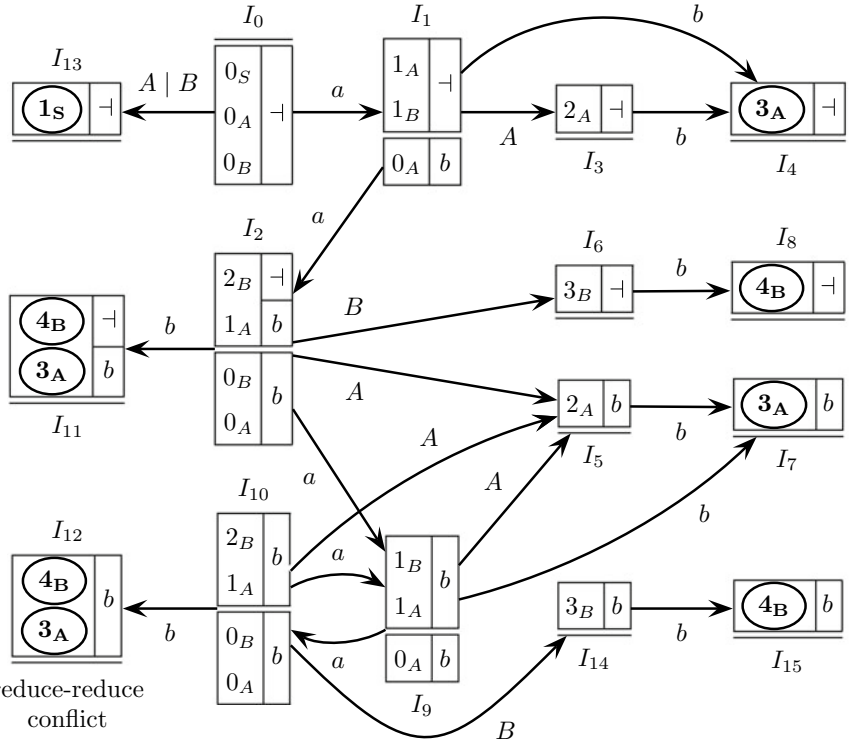
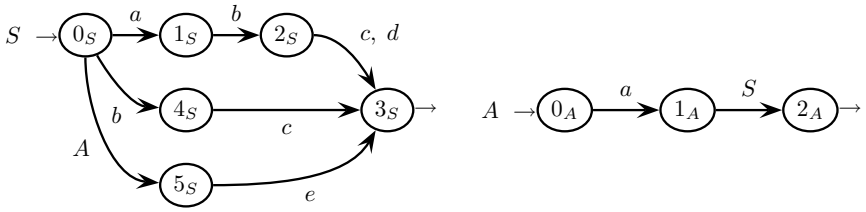


Fig. 4.19 Grammar with net and pilot with reduce-reduce conflict (Example 4.40)

EBNF grammar

$$S \to a\,b\,(\,c \mid d\,) \mid b\,c \mid A\,e \qquad\qquad A \to a\,S$$

machine net



pilot graph

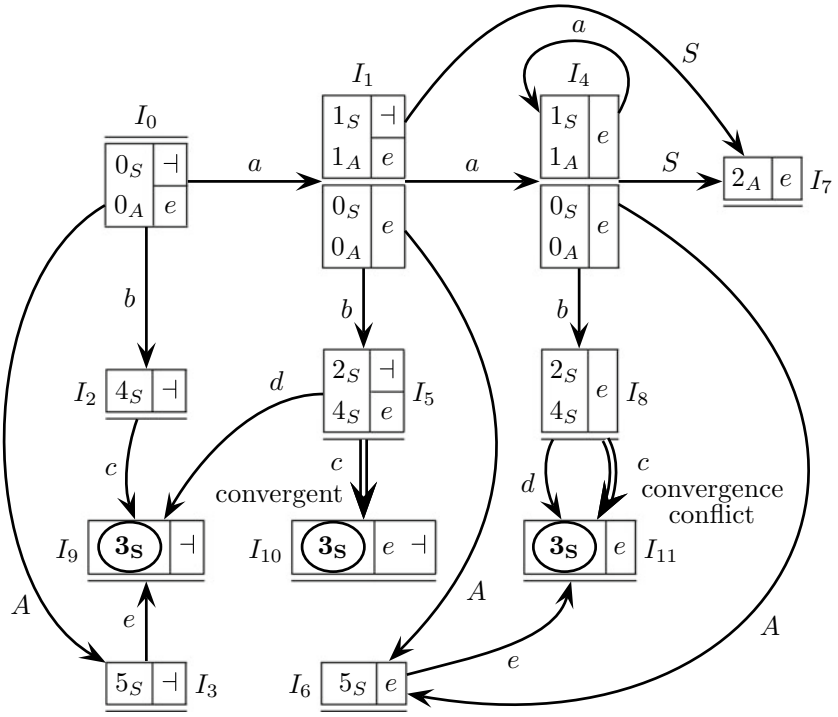


Fig. 4.20 Grammar with net and pilot (Example 4.41); the double-line arcs in the pilot are convergent; the pilot has a convergence conflict

have any shift-reduce conflicts and is trivially free from reduce-reduce conflicts, but having a convergence conflict, it violates the *ELR* (1) condition. □

Similarly Fig. 4.23 shows a grammar, its machine net representation and the corresponding pilot graph. The pilot has a convergent transition that is not conflicting, as the look-ahead sets are disjoint.

4.6.3.1 Parser Algorithm

Given the pilot *DFA* of an *ELR* (1) grammar or machine net, we explain how to obtain a deterministic pushdown automaton *DPDA* that recognizes and parses the sentences.

At the cost of some repetition, we recall the three sorts of abstract machines involved: the net $\mathcal{M}$ of *DFA* M_S, M_A, ..., with state set $Q = \{ q, \ldots \}$ (states are drawn as circular or oval nodes); the pilot *DFA* $\mathcal{P}$ with m-state set $R = \{ I_0, I_1, \ldots \}$ (drawn as rectangular nodes); and the *DPDA* to be next defined. As said, the *DPDA* stores in the stack the series of m-states entered during the computation, enriched with additional information used in the parsing steps. Moreover the m-states are interleaved with grammar symbols. The *current m-state*, i.e., the one on top of stack, determines the next move: either a shift that scans the next token, or a reduction of a topmost stack segment (*handle*) to one of the final states (i.e., a nonterminal) included in the current m-state. The absence of shift-reduce conflicts makes the choice between shift and reduction deterministic. Similarly the absence of reduce-reduce conflicts allows the parser to uniquely identify the final state of a machine. However this leaves open the problem to find the stack portions to be used as handle. For that two designs will be presented: the first uses a finite pushdown alphabet; the second uses unbounded integer pointers and, strictly speaking, no longer qualifies as a pushdown automaton.

First we specify the *pushdown stack alphabet*. Since for a given net $\mathcal{M}$ there are finitely many different candidates, the number of m-states is bounded; the number of candidates in any m-state is bounded by $| Q |$, by assuming that all the candidates with the same state are coalesced. Moreover we assume that the candidates in an m-state are (arbitrarily) ordered, so that each one occurs at a *position* (or offset), which will be referred to by other m-states using a pointer named *candidate identifier* (*cid*).

The *DPDA* stack elements are of two types: grammar symbols, i.e., elements of $\Sigma \cup V$, and *stack m-states* (*sms*). An *sms*, which is denoted by J, contains an ordered set of triples, named *stack candidates*, of the form:

$$\langle q_A, \pi, cid \rangle \quad \text{where} \begin{cases} q_A \in Q & \text{(state)} \\ \pi \subseteq \Sigma \cup \{ \dashv \} & \text{(look-ahead set)} \\ 1 \leq cid \leq | Q | \text{ or } cid = \perp & \text{(candidate identifier)} \end{cases}$$

where $\perp$ represents the nil value for the *cid* pointer. For readability, a *cid* value will be prefixed by a ♯ marker, e.g., ♯1, ♯2, etc.

We introduce a natural surjective mapping, to be denoted by μ, from the set of *sms* to the set of m-states defined by: $\mu(J) = I$ if, and only if, by deleting the third field (*cid*) from the stack candidates of J, we obtain exactly the candidates of I. As said, in the stack the *sms* are interleaved with grammar symbols.

Algorithm 4.42 (*ELR* (1) *parser* $\mathcal{A}$ *as a DPDA*) Let the current stack be as follows:

$$J[0]\ a_1\ J[1]\ a_2\ \ldots\ a_k\ \overset{top}{J[k]}$$

where a_i is a grammar symbol.

initialization

The initial stack just contains the *sms* below:

$$J_0 = \{\, s \mid \ \ s = \langle\, q,\ \pi,\ \perp \,\rangle \text{ for every candidate } \langle\, q,\ \pi \,\rangle \in I_0 \,\}$$

(thus $\mu(J_0) = I_0$, the initial m-state of the pilot).

shift move

Let the top *sms* be J and $I = \mu(J)$; and let the current token be $a \in \Sigma$. Assume that m-state I has transition $\vartheta\,(I,\ a) = I'$. The shift move does as follows:

1. push token a on stack and get the next token
2. push on stack the *sms* J' computed as follows:

$$J' = \left\{ \left\langle q_A{}',\ \rho,\ \sharp i \right\rangle \,\middle|\, \begin{array}{l} \langle\, q_A,\ \rho,\ \sharp j \,\rangle \text{ is at position } i \text{ in } J \\ \text{and } q_A \xrightarrow{a} q_A{}' \in \delta \end{array} \right\} \tag{4.11}$$

$$\cup \left\{ \langle\, 0_B,\ \sigma,\ \perp \,\rangle \,\middle|\, \langle\, 0_B,\ \sigma \,\rangle \in I'_{|\,closure} \right\} \tag{4.12}$$

(thus $\mu(J') = I'$)

(note: by the last condition in (4.11), state $q_A{}'$ is of the base of I')

reduction move (non-initial state)

The current stack is as follows:

$$J[0]\ a_1\ J[1]\ a_2\ \ldots\ a_k\ J[k]$$

and let the corresponding m-states be $I[i] = \mu\,(\,J[i]\,)$ with $0 \le i \le k$; also let the current token be a.

Assume that by inspecting $I[k]$, the pilot chooses the reduction candidate $c = \langle\, q_A,\ \pi \,\rangle \in I[k]$, where q_A is a final but non-initial state. Let $t_k = \langle\, q_A,\ \rho,\ \sharp i_k \,\rangle \in J[k]$ be the (only) stack candidate such that the current token a satisfies $a \in \rho$.

From i_k a *cid* chain starts, which links stack candidate t_k to a stack candidate $t_{k-1} = \langle\, p_A,\ \rho,\ \sharp i_{k-1} \,\rangle \in J[k-1]$, and so on until a stack candidate $t_h \in J[h]$ is reached that has a nil *cid* (therefore its state is initial):

$$t_h = \langle\, 0_A,\ \rho,\ \perp \,\rangle$$

The reduction move does as follows:

1. grow the syntax forest by applying the following reduction:

$$a_{h+1}\ a_{h+2}\ \ldots\ a_k \rightsquigarrow A$$

2. pop the following stack symbols:

$$J[k]\ a_k\ J[k-1]\ a_{k-1}\ \ldots\ J[h+1]\ a_{h+1}$$

3. execute the nonterminal shift move $\vartheta\,(\,J[h],\ A\,)$ (see below)

reduction move (initial state)
It differs from the preceding case in the state of the chosen candidate $c = \langle\, 0_A,\ \pi,\ \perp \,\rangle$, which is final and initial. The parser move grows the syntax forest by the reduction $\varepsilon \rightsquigarrow A$ and performs the nonterminal shift move corresponding to $\vartheta\,(\,I[k],\ A\,)$.

nonterminal shift move
It is the same as a shift move, except that the shifted symbol A is a nonterminal. The only difference is that the parser does not read the next input token at line (1) of *Shift move*.

acceptance
The parser accepts and halts when the stack is J_0, the move is the nonterminal shift defined by $\vartheta\,(\,I_0,\ S\,)$ and the current token is $\dashv$. □

For shift moves, we note that the set computed at line (4.11) contains multiple candidates that have the same state whenever arc $\vartheta\,(\,I,\ a\,)$ is convergent. It may help to look at the situation of a shift move (Eqs. (4.11) and (4.12)) schematized in Fig. 4.21.

Example 4.43 (*parsing trace*) The step-by-step execution of the parser on input string $(\,(\,)\,a\,)$ produces the trace shown in Fig. 4.22. For clarity there are two parallel tracks: the input string, progressively replaced by the nonterminal symbols shifted on the stack; and the stack, containing stack m-states. The stack has one more entry than the scanned prefix; the suffix yet to be scanned is to the right of the stack. Inside a stack element $J[k]$, each 3-tuple is identified by its ordinal position, starting from 1 for the first 3-tuple; a value $\sharp i$ in a *cid* field of element $J[k+1]$ encodes a pointer to the i-th 3-tuple of $J[k]$.

For readability, the m-state number appears framed in each stack element, e.g., I_0 is denoted $\boxed{0}$; the final candidates are encircled, e.g., $\textcircled{1_E}$. To avoid clogging, the

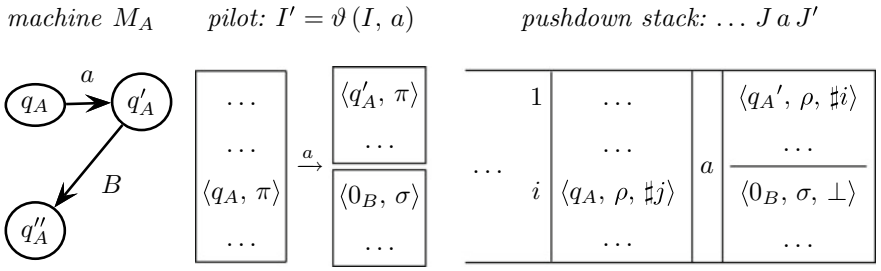


Fig. 4.21 From left to right: machine transition $q_A \xrightarrow{a} q'_A$, shift move $I \xrightarrow{a} I'$ and stack configuration. The *cid* value $\sharp i$ in the topmost stack m-state points to the candidate at offset i in the underlying stack m-state

look-ahead sets are not shown but can be found in the pilot graph on p. 254 (they are only needed for convergent transitions, which do not occur here).

Figure 4.22 highlights the shift moves as dashed forward arrows that link two topmost stack candidates. For instance, the first (from the figure top) terminal shift, on '(', is $\langle 0_T, \perp \rangle \xrightarrow{(} \langle 1_T, \sharp 2 \rangle$. The first nonterminal shift is the third one, on E, i.e., $\langle 1_T, \sharp 3 \rangle \xrightarrow{E} \langle 2_T, \sharp 1 \rangle$, soon after the null reduction $\varepsilon \rightsquigarrow E$.

In the same figure, the reduction handles are evidenced by means of solid backward arrows that link the candidates involved. For instance, for reduction $(\,E\,) \rightsquigarrow T$ the stack configuration shows the chain of three pointers $\sharp 1$, $\sharp 1$ and $\sharp 3$—these three stack elements form the handle and are popped—and finally the nil pointer $\perp$—this stack element is the reduction origin and is not popped. The nil pointer marks the initial candidate $\langle 0_T, \perp \rangle$. In the same stack element, a dotted arrow from candidate $\langle 0_T, \perp \rangle$ to candidate $\langle 0_E, \perp \rangle$ points to the candidate from which the subsequent shift on T starts: see the dashed arrow in the stack configuration below. The solid self-loop on candidate $\langle 0_E, \perp \rangle$ (at the 2^{nd} shift move on "("—see the effect on the line below) highlights the null reduction $\varepsilon \rightsquigarrow E$, which does not pop anything from the stack, and immediately triggers a shift on E.

We observe that the parser must store on stack the scanned grammar symbols, because in a reduction move, at step 2 of the algorithm, they may be necessary for selecting the correct reduction.

Returning to the example, the reductions are executed in the order:

$$\varepsilon \rightsquigarrow E \qquad (\,E\,) \rightsquigarrow T \qquad a \rightsquigarrow T \qquad T\,T \rightsquigarrow E \qquad (\,E\,) \rightsquigarrow T \qquad T \rightsquigarrow E$$

which matches a rightmost derivation but in reversed order. □

We examine more closely the case of convergent conflict-free arcs. Going back to the scheme in Fig. 4.21 (p. 262), notice that the stack candidates linked by a *cid* chain are mapped onto m-state candidates that have the same machine state (here the stack candidate $\langle q'_A, \rho, \sharp i \rangle$ is linked via $\sharp i$ to $\langle q_A, \rho, \sharp j \rangle$): the look-ahead set π of such m-state candidates is in general a superset of the set ρ included in the stack candidate, due to the possible presence of convergent transitions; the two sets ρ and

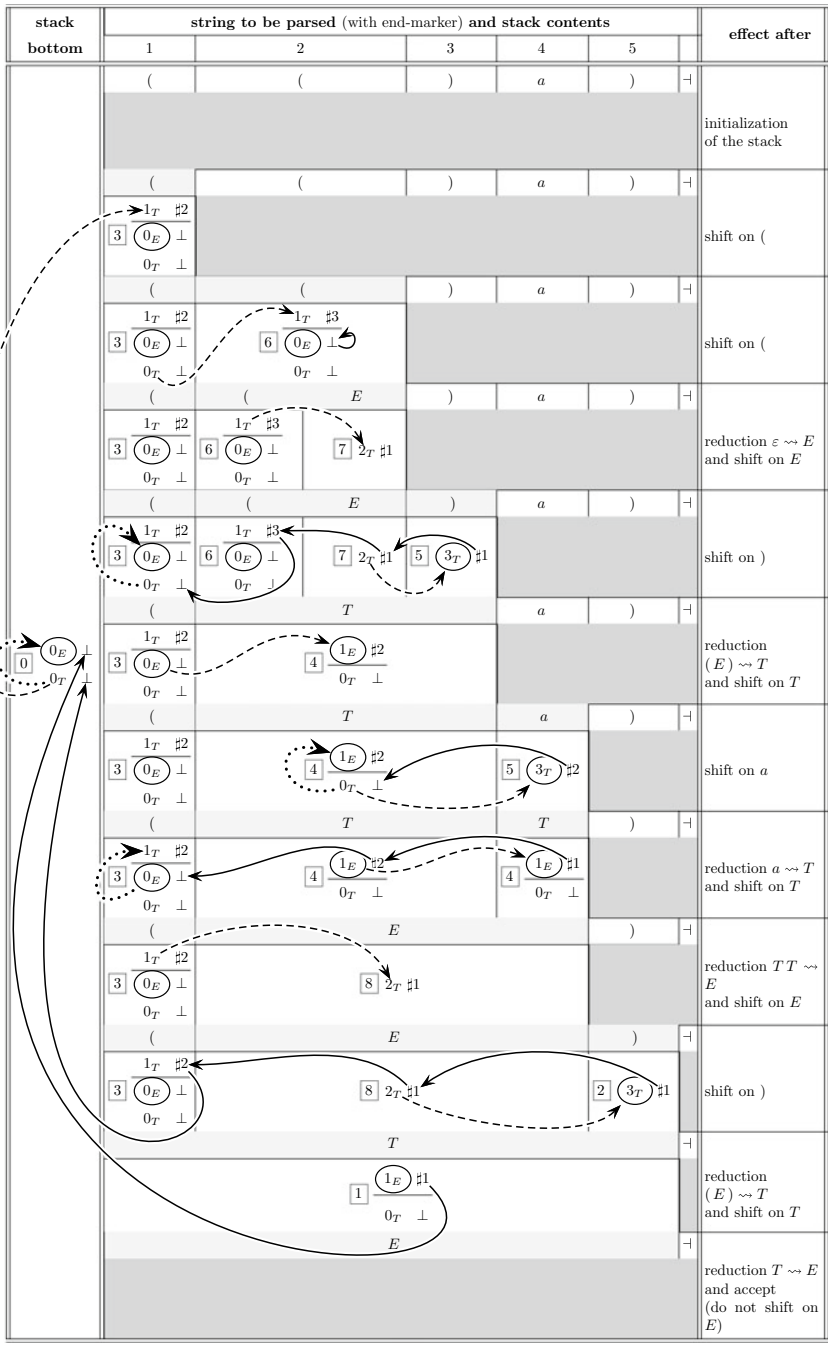


Fig. 4.22 Parsing steps for string (( ) *a*); the grammar and the *ELR* (1) pilot are in Figs. 4.12 and 4.17 on p. 254; the name J_h of a *sms* maps onto the corresponding m-state $\mu(J_h) = I_h$

π coincide when no convergent transition is taken on the pilot automaton at parsing time.

An *ELR* (1) grammar with convergent arcs is studied next.

Example 4.44 (*convergent arcs in the pilot*) The net, the pilot graph, and the trace of a parse are shown in Fig. 4.23, where for readability the *cid* values in the stack candidates are visualized as backward pointing arrows. Stack m-state J_3 contains two candidates, which just differ in their look-ahead sets. In the corresponding pilot m-state I_3, the two candidates are the targets of a convergent non-conflicting m-state transition (highlighted with a double line in the pilot graph). □

Quite frequently the base of some m-state includes more than one candidate. Since this feature distinguishes the bottom-up parsers from the top-down ones to be later studied, we present an example (Example 4.45).

Example 4.45 (*several candidates in the base of a pilot m-state*) We refer to Fig. 4.38 on p. 289. The grammar below:

$$S \to a^* N \qquad N \to a\, N\, b \mid \varepsilon$$

generates the language $\{ a^n b^m \mid \ n \geq m \geq 0 \}$. The base of m-states I_1, I_2, I_4 and I_5 includes two candidates. The pilot satisfies the *ELR* (1) condition for the following reasons:

- At most one reduction candidate is present in any m-state, therefore reduce-reduce conflicts are ruled out.
- No shift-reduce conflict is present, because for every m-state that includes a reduction candidate with look-ahead set ρ, no outgoing arc is labeled with a token that is in ρ. We check this: for m-states I_0, I_1 and I_2, the label a of the outgoing arcs is not included in the look-ahead sets for those m-states; for m-states I_4 and I_5, token b labeling the outgoing arcs is not in the look-ahead of the final candidates.

A step-by-step analysis of string $a\, a\, a\, b$ with the pilot in Fig. 4.38 would show that several parsing attempts proceed simultaneously, which correspond to paths in the machines M_S and M_N of the net. □

Next, we show the case of non-conflicting reductions in the same m-state.

Example 4.46 (*multiple reductions in an ELR* (1) *m-state*) Figure 4.24 shows the grammar of a finite language, along with its machine network representation and the corresponding pilot automaton. In the pilot of Fig. 4.24, the m-states I_3 and I_6 include two final candidates, but no reduce-reduce conflict is present as the look-ahead sets of any two final candidates that are in the same m-state are disjoint. Referring to m-state I_3, if symbol $\dashv$ is scanned after reading a letter a, then the reduction $a \rightsquigarrow A$ must be executed, corresponding to derivation $S \Rightarrow A \Rightarrow a$; instead, if a second a is

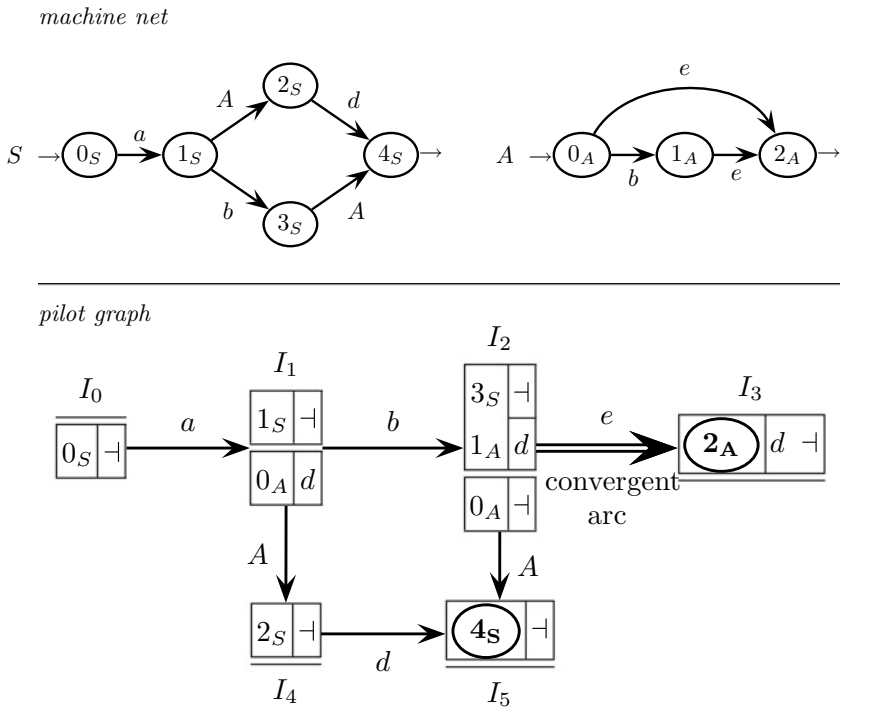
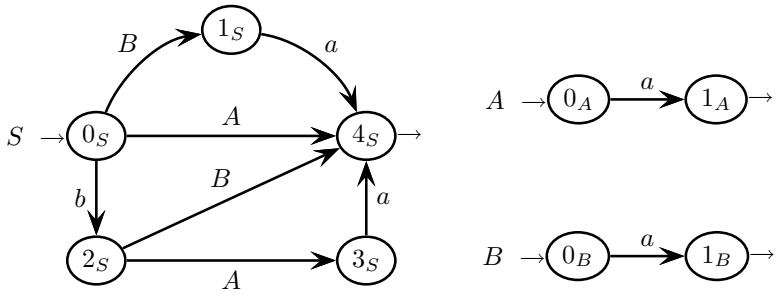


Fig. 4.23 *ELR* (1) net and pilot with convergent arcs (double line), and parsing trace of string $a\,b\,e\,d\ \dashv$, for Example 4.44

EBNF grammar

$$S \to B\,a \mid A \mid b\,(B \mid A\,a) \qquad A \to a \qquad B \to a$$

machine net



pilot graph

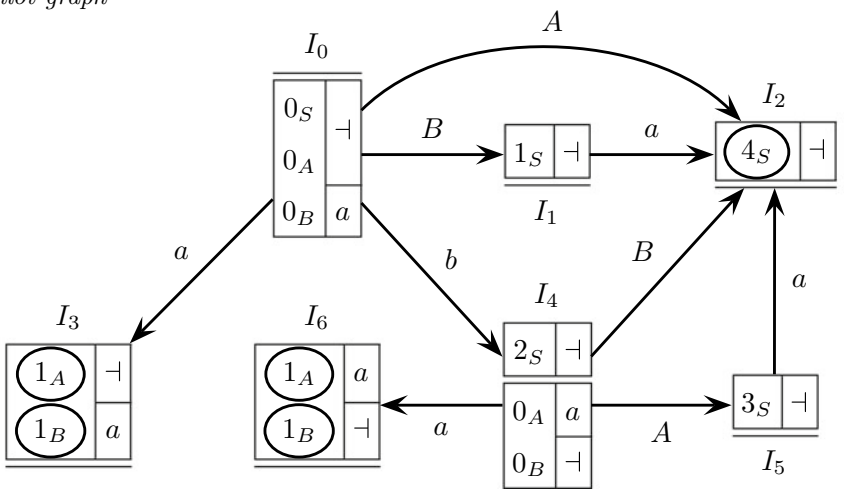


Fig. 4.24 Grammar with net and pilot with non-conflicting reductions for Example 4.46

scanned after the first one, then the correct reduction is $a \rightsquigarrow B$ because the derivation is $S \Rightarrow B\,a \Rightarrow a\,a$.

Similarly but with reference to m-state I_6, if the input symbol is $\dashv$ after reading string $b\,a$, then a reduction $a \rightsquigarrow B$ must be performed and the derivation is $S \Rightarrow b\,B \Rightarrow b\,a$; otherwise, if after prefix $b\,a$ the input is a, then the correct reduction is $a \rightsquigarrow A$ and the derivation is $S \Rightarrow b\,A\,a \Rightarrow b\,a\,a$. □

4.6.4 Simplifications for *BNF* Grammars

Historically, the original theory of shift-reduce parsers applied only to pure *BNF* grammars and, although attempts were made to extend it to *EBNF* grammars, to our knowledge the present one is the first systematic presentation of *ELR* (1) parsers, based on the machine net representation of *EBNF* grammars. Since the widespread parser generation tools (such as Bison) as well as the best known textbooks still use non-extended grammars, we briefly comment on how our approach relates to the classical theory.[15] The reader more interested to practical parser design should jump to p. 270, where we indicate the useful simplifications made possible by pure *BNF* grammars.

4.6.4.1 *ELR* (1) Versus Classical *LR* (1) Definitions

In a *BNF* grammar each nonterminal A has finitely many alternative rules $A \to \alpha \mid \beta \mid \ldots$. Since an alternative does not contain iterative, i.e., star, or union operations, it is straightforward to represent nonterminal A as a nondeterministic finite machine (*NFA*) N_A that has a very simple acyclic graph: from the initial state 0_A as many legs originate as there are alternative rules for A. Each leg exactly reproduces an alternative and ends in a different final state. Therefore the graph has the form of a tree, such that the only one branching node is the initial state. Such a machine is nondeterministic if two alternatives start with the same grammar symbol. We compare such an *NFA* with the *DFA* machine M_A we would build for the same set of alternative rules. First we explain why the *LR* (1) pilot constructed with the traditional method is substantially equivalent to our pilot from the point of view of its use in a parser. Then we show that no m-state in the traditional *LR* (1) pilot can exhibit the multiple-transition property (*MTP*). Therefore the only pertinent conditions for parser determinism are the absence of shift-reduce and reduce-reduce conflicts; see clauses (4.9) and (4.10) of Definition 4.38 on p. 254.

Our machine M_A may differ from N_A in two ways: determinism and minimality. First, machine M_A is assumed to be deterministic, but this assumption is made out of convenience, not of necessity. For instance, consider a nonterminal C with two alternatives: $C \to \mathit{if}\ E\ \mathit{then}\ I \mid \mathit{if}\ E\ \mathit{then}\ I\ \mathit{else}\ I$. Determinization has the effect of normalizing the alternatives by left factoring the longest common prefix; this yields the equivalent *EBNF* grammar $C \to \mathit{if}\ E\ \mathit{then}\ I\ (\varepsilon \mid \mathit{else}\ I)$. The pilots constructed for the original and for the extended grammar are equivalent in the sense that either both ones have a conflict or neither one has it.

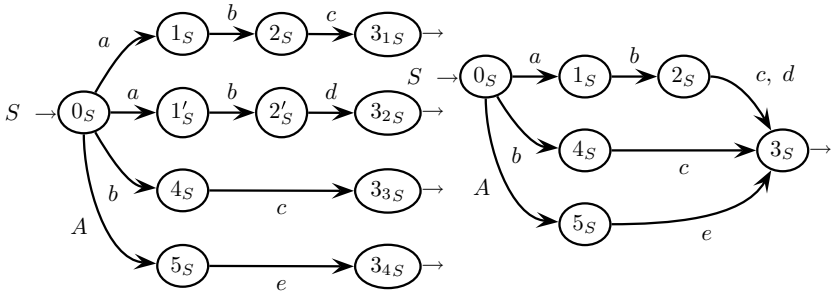
Second, we allow and actually recommend that the graph of machine M_A be minimal with respect to the number of states, provided that the initial state is not reentered. Clearly in the graph of machine N_A, no arc may reenter the initial state,

[15] We mainly refer to the original paper by Knuth [16] and to its dissemination in many classical textbooks such as [17,18]. There have been numerous attempts to extend the theory to *EBNF* grammars; we refer to the method of Borsotti et al. [19] which includes further references.

grammars

BNF	*EBNF*
$S \to a\,b\,c \mid a\,b\,d \mid b\,c \mid A\,e$	$S \to a\,b\,(c \mid d) \mid b\,c \mid A\,e$
$A \to a\,S$	$A \to a\,S$

machine nets



net common part

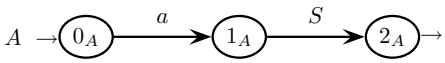


Fig. 4.25 *BNF* and *EBNF* grammars and their machine nets $\{ M_S,\ M_A \}$

exactly as in machine M_A; but the final states of the grammar alternatives are all distinct in N_A, while in M_A they are merged together by determinization and state minimization. Moreover, also several non-final states of N_A are possibly coalesced if they are *undistinguishable* for M_A.

The effect of such a state reduction on the pilot is that some pilot arcs may become convergent. Therefore in addition to checking conditions (4.9) and (4.10), Definition 4.38 imposes that any convergent arc be free from conflicts. Since this point is quite subtle, we illustrate it by the next example.

Example 4.47 (*state-reduction and convergent transitions*) Consider the equivalent *BNF* and *EBNF* grammars and the corresponding machines in Fig. 4.25. After determinizing, the states $3_{1S}, 3_{2S}, 3_{3S}, 3_{4S}$ of machine N_S are equivalent and are merged into the state 3_S of machine M_S. Turning our attention to the *LR* and *ELR* conditions, we find that the *BNF* grammar has a reduce-reduce conflict caused by the derivations below:

$$S \Rightarrow A\,e \Rightarrow a\,S\,e \Rightarrow a\,b\,c\,e \qquad \text{and} \qquad S \Rightarrow A\,e \Rightarrow a\,S\,e \Rightarrow a\,a\,b\,c\,e$$

On the other hand, for the *EBNF* grammar the pilot $\mathcal{P}_{\{M_S,\,M_A\}}$, already shown in Fig. 4.20 on p. 258, has two convergent arcs (double line), one with a conflict and the

other without. Arc $I_8 \xrightarrow{c} I_{11}$ violates the *ELR* (1) condition because the look-ahead sets of 2_S and 4_S in I_8 are not disjoint. □

Thus, the state minimization of the net machines may cause the *ELR* pilot to have a convergence conflict whereas the *LR* pilot has a reduce-reduce conflict. On the other hand, shift-reduce conflicts equally affect the *LR* and *ELR* pilots.

From these findings, it is tempting to assert that, given a *BNF* grammar and an equivalent net, either both ones are conflict-free or both ones have conflicts (although not always of the same type). But the assertion needs to be made more precise, since starting from an *ELR* (1) net, several equivalent *BNF* grammars can be obtained by removing the regular expression operations in different ways. Such grammars may or may not be *LR* (1), but at least one of them is *LR* (1): the right-linearized grammar (p. 239).[16]

To illustrate the discussion, it helps us consider a simple example where the machine net is *ELR* (1), yet another equivalent grammar obtained by a fairly natural transformation, has conflicts.

Example 4.48 (*grammar transformations may not preserve ELR* (1)) A phrase S has the structure $E\ (\ s\ E\ )^*$, where a construct E has either the form $b^+\ b^n\ e^n$ or the form $b^n\ e^n\ e$, with $n \geq 0$. The language is defined by the *EBNF* grammar and net in Fig. 4.26, which are *ELR* (1). On the contrary, there is a conflict in the equivalent *BNF* grammar below:

$$S \to E\ s\ S \mid E \qquad E \to B\ F \mid F\ e \qquad F \to b\ F\ e \mid \varepsilon \qquad B \to b\ B \mid b$$

In fact, consider the corresponding net (only the part that has changed is redrawn) and its partial pilot graph (the whole pilot has many more m-states and arcs which here do not matter) shown in Fig. 4.27. The m-state I_1 has a shift-reduce conflict due to the indecision whether in analyzing a legal string of the language such as $b\,b\,e$, the parser, after shifting the first character b, should perform a reduction $b \rightsquigarrow B$ or shift the second character b.

On the other hand, the right-linearized grammar below (derived from the original *ELR* (1) net), which is *BNF* by construction:

$$\begin{array}{lll}
0_S \to 0_E\ 1_S & 0_E \to b\ 1_E \mid 0_F\ 3_E & 0_F \to \varepsilon \mid b\ 1_F \\
1_S \to \varepsilon \mid s\ 2_S & 1_E \to b\ 1_E \mid 0_F\ 2_E & 1_F \to 0_F\ 2_F \\
2_S \to 0_E\ 1_S & 2_E \to \varepsilon & 2_F \to e\ 3_F \\
 & 3_E \to e\ 4_E & 3_F \to \varepsilon \\
 & 4_E \to \varepsilon &
\end{array}$$

[16] A rigorous presentation of these and other theoretical properties is in [20].

EBNF grammar

$$S \to E\ (\,s\,E\,)^* \qquad E \to b^+\,F \ \mid\ F\,e \qquad F \to b\,F\,e \ \mid\ \varepsilon$$

machine net

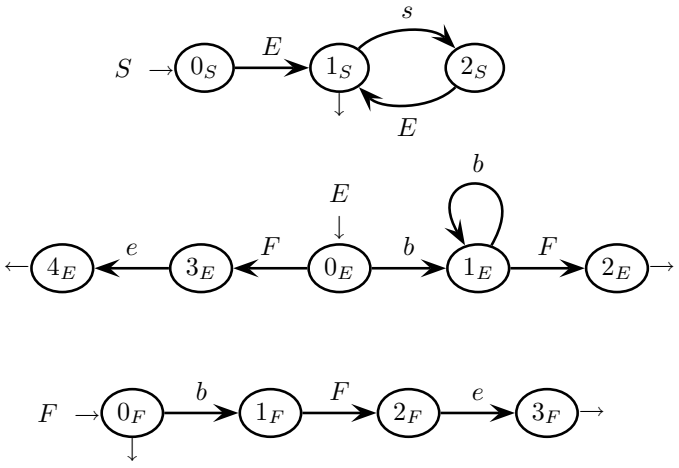


Fig. 4.26 An *EBNF* grammar that is *ELR* (1) (Example 4.48)

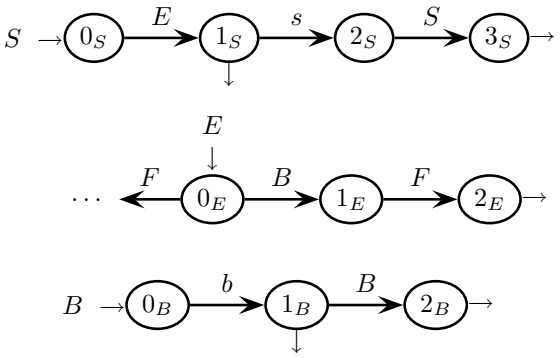
is surely of type *LR* (1). Notice in fact that it postpones any reduction decision as long as possible, and thus avoids any conflicts. □

4.6.4.2 Superfluous Features

For *LR* (1) grammars that do not use regular expressions, some features of the *ELR* (1) parsing algorithm (p. 260) become superfluous. We briefly discuss them to highlight the differences between extended and basic shift-reduce parsers. Since now the graph of every machine is a tree, two arcs originating from distinct states may not enter the same state, therefore in the pilot the convergent arcs never occur. Moreover, if the alternatives $A \to \alpha \mid \beta \mid \ldots$, are recognized in distinct final states $q_{\alpha,A}$, $q_{\beta,A}$, ... of machine M_A, the bookkeeping information needed for reductions becomes useless: if the candidate chosen by the parser corresponds to state $q_{\alpha,A}$, the reduction move simply pops $2 \times |\alpha|$ stack elements and performs the reduction $\alpha \rightsquigarrow A$. Since the pointers to a preceding stack candidate are no longer needed, the stack m-states coincide with the pilot ones.

Second, for a related reason the parser can do without the interleaved grammar symbols on the stack, because the pilot of an *LR* (1) grammar has the well-known property that all the arcs entering an m-state carry the same terminal/nonterminal label. Therefore, when in the current m-state the parser decides to reduce by using a

machine net (partial)



pilot graph (partial)

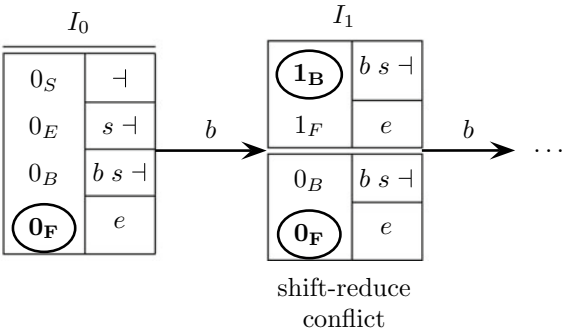


Fig. 4.27 An equivalent network that has conflicts (Example 4.48)

certain final candidate, the reduction handle is uniquely determined by the final state contained in the candidate.

The above simplifications have another consequence: for *BNF* grammars the use of machine nets and their states becomes subjectively less attractive than the classical notation based on marked grammar rules (p. 236).

4.6.5 Parser Implementation Using a Vector Stack

Before finishing with bottom-up parsing, we present a parser implementation alternative to Algorithm 4.42 on p. 260, where the memory of the analyzer is a *vector stack*, i.e., an array of elements such that any element can be directly accessed by means of an integer index. There are two reasons for presenting the new implementation: this technique anticipates the implementation of nondeterministic tabular parsers (Sect. 4.11); and it is potentially faster. On the other hand, a vector stack as data type is

more general than a pushdown stack, therefore this parser cannot be viewed as a pure *DPDA*.

As before, the elements in the vector stack are of two alternating types: *vector-stack m-states*, *vsms* and grammar symbols. A *vsms*, denoted by J, is a set of pairs, named *vector-stack candidates*, the form $\langle\, q_A,\ elemid \,\rangle$ of which simply differs from the earlier stack candidates because the second component is a positive integer named *element identifier* (*elemid*) instead of the candidate identifier (*cid*). Notice also that now the set is not ordered. The surjective mapping from vector-stack m-states to pilot m-states is denoted μ, as before. Each *elemid* points back to the vector-stack element containing the initial state of the current machine, so that when a reduction move is performed, the length of the string to be reduced—and the reduction handle—can be obtained directly without inspecting the stack elements below the top one. Clearly the value of *elemid* ranges from 1 to the maximum vector-stack height, which linearly depends on the input length.

Algorithm 4.49 (*ELR* (1) *parser that uses a vector stack*) The current (vector-)stack is denoted by:

$$J[0]\ a_1\ J[1]\ a_2\ \ldots\ a_k\ J[k]$$

with $k \geq 0$, where the top element is $J[k]$.

initialization

The analysis starts by pushing on the stack the following element:

$$J_0 = \{\ s \mid\ \ s = \langle\, q,\ \pi,\ 0 \,\rangle \text{ for every candidate } \langle\, q,\ \pi \,\rangle \in I_0\ \}$$

shift move

Assume that the top element is J with $I = \mu(J)$, the variable k stores the index of the top stack element, and the current input character is a. Assume that by inspecting I, the shift $\vartheta\,(\,I,\ a\,) = I'$ has been chosen. The parser move does the following:

1. push token a on stack and get next token
2. push on stack the *vsms* J' (more precisely do $k++$ and $J[k] := J'$) that contains the following candidates:

if cand. $c = \langle\, q_A,\ \rho,\ i \,\rangle \in J$ and $q_A \xrightarrow{a} q'_A \in \delta$ **then** J' contains cand. $\langle\, q_A{}',\ \rho,\ i \,\rangle$ **end if**	**if** cand. $c = \langle\, 0_A,\ \pi \,\rangle \in I'_{\lvert closure}$ **then** J' contains cand. $\langle\, 0_A,\ \pi,\ k \,\rangle$ **end if**

(thus $\mu(J') = I'$)

nonterminal shift move
It applies after a reduction move that started in a final state of machine M_B. It is the same as a shift move, except that the shifted symbol B is a nonterminal, i.e., $\vartheta(I, B) = I'$. The difference with respect to a shift move is that the parser does not read the next token.

reduction move (non-initial state)
The stack is as follows:

$$J[0]\, a_1\, J[1]\, a_2\, \ldots\, a_k\, J[k]$$

and let the corresponding m-states be $I[0]\, I[1]\, \ldots\, I[k]$. Assume that the pilot chooses the reduction candidate $c = \langle q_A, \pi \rangle \in I[k]$, where q_A is a final but non-initial state of machine M_A; let $\langle q_A, \rho, h \rangle$ be the (only) candidate in $J[k]$ corresponding to c. The reduction move does:

1. pop the following vector-stack elements:

$$J[k]\, a_k\, J[k-1]\, a_{k-1}\, \ldots\, J[h+1]$$

2. build a portion of the syntax forest by applying the following reduction:

$$a_{h+1}\, \ldots\, a_k \rightsquigarrow A$$

3. execute the nonterminal shift move $\vartheta(J[h], A)$

reduction move (initial state)
It differs from the preceding case in that, for the chosen reduction candidate $c = \langle 0_A, \pi, i \rangle \in J[k]$, state 0_A is initial and final; then necessarily the *elemid* field i is equal to k, the current top stack index. Therefore reduction $\varepsilon \rightsquigarrow A$ has to be applied. The parser move does the following:

1. build a portion of the syntax forest corresponding to this reduction
2. go to the nonterminal shift move $\vartheta(I, A)$

acceptance
The parser accepts and halts when the stack is J_0, the move is the nonterminal shift defined by $\vartheta(I_0, S)$ and the current token is $\dashv$. □

Although the algorithm uses a stack, it cannot be viewed as a *PDA* because the stack alphabet is unbounded since a *vsms* contains integer values.

Example 4.50 (*parsing trace*) Figure 4.28, to be compared with Fig. 4.22 on p. 263, shows the same step-by-step execution of the parser as in Example 4.43, on input string (( ) *a*); the graphical conventions are the same.

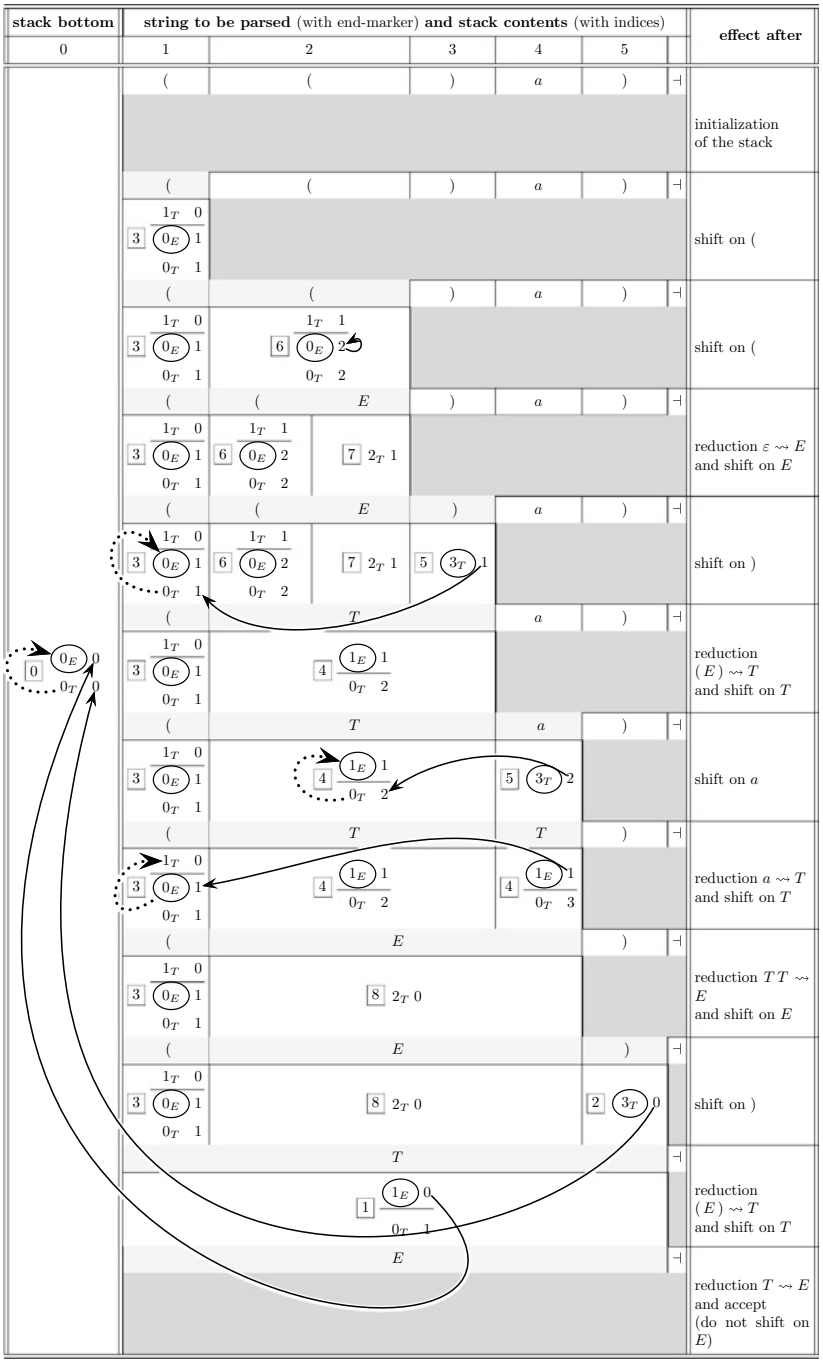


Fig. 4.28 Tabulation of parsing steps for string $(\,(\,)\,a\,)$; the parser uses a vector stack and is generated by the grammar in Fig. 4.12 with the *ELR* (1) pilot in Fig. 4.17

In every *vsms* on the stack, the second field of each candidate is the *elemid* index, which points back to some inner position of the stack. Index *elemid* is equal to the current stack position for all the candidates in the closure part of a stack element, and it points to some previous position for the candidates of the base. The reduction handles are highlighted by means of solid backward pointers, and by a dotted arrow to locate the candidate to be shifted soon after reducing. Notice that now the arrows span a longer distance than in Fig. 4.22, as index *elemid* goes directly to the origin of the reduction handle. The forward shift arrows are the same as those in Fig. 4.22 and are not shown.

The results of the analysis, i.e., the execution order of the reductions and the obtained syntax tree, are identical to those of Example 4.43.

To complete the exemplification, we apply the vector-stack parser to a previous net featuring a convergent arc (Fig. 4.23 on p. 265): the parsing traces are shown in Fig. 4.29, where the integer pointers are represented by backward pointing solid arrows. □

4.6.6 Lengthening the Look-Ahead

More often than not, technical grammars meet the *ELR* (1) or *LR* (1) condition, but one may find a grammar that needs some adjustment. Here we consider certain grammar transformations for *BNF* grammars and for simplicity we omit the *EBNF* case that, to our knowledge, has been less investigated. However, remember that any *EBNF* grammar can always be transformed into an equivalent *BNF* one: for instance into the right-linearized form; see Sect. 4.5.2.1 on p. 239.

In Sect. 4.6 on p. 244, we have intuitively hinted to the meaning of a look-ahead of length two (or more): two (or more) consecutive characters are inspected to let the parser move deterministically. For instance, suppose that a grammar has an alternative for the axiom: $S \to A\,c\,d \mid B\,c\,e$; and that both nonterminals A and B are nullable, i.e., $A \to \varepsilon$ and $B \to \varepsilon$. Then in the initial m-state I_0 of the pilot, there obviously is a conflict between the reductions of A and B, since both of them have look-ahead c if we look at the first following character, that is if we set $k = 1$. But, if we look at the next two consecutive characters that may follow nonterminals A and B, i.e., if we set $k = 2$, then we, respectively, see strings $c\,d$ and $c\,e$, which are different, and the conflict vanishes. The situation would be more involved if d and e were nonterminals, say D and E, since then we should inspect their initial characters; but the next examples will help to understand. To sum up, a *BNF* grammar is of type *LR* (k), for $k \geq 1$, if it has a deterministic bottom-up parser that uses a look-ahead of length k.

An interesting case to discuss is when a grammar is of type *LR*, though with a look-ahead of two or more, i.e., *LR* (k) with $k > 1$. Then, we may wish to transform the grammar into an equivalent one with lower look-ahead length, possibly down to $k = 1$. In principle, this target can always be achieved; see Sect. 4.9 on p. 346. Here

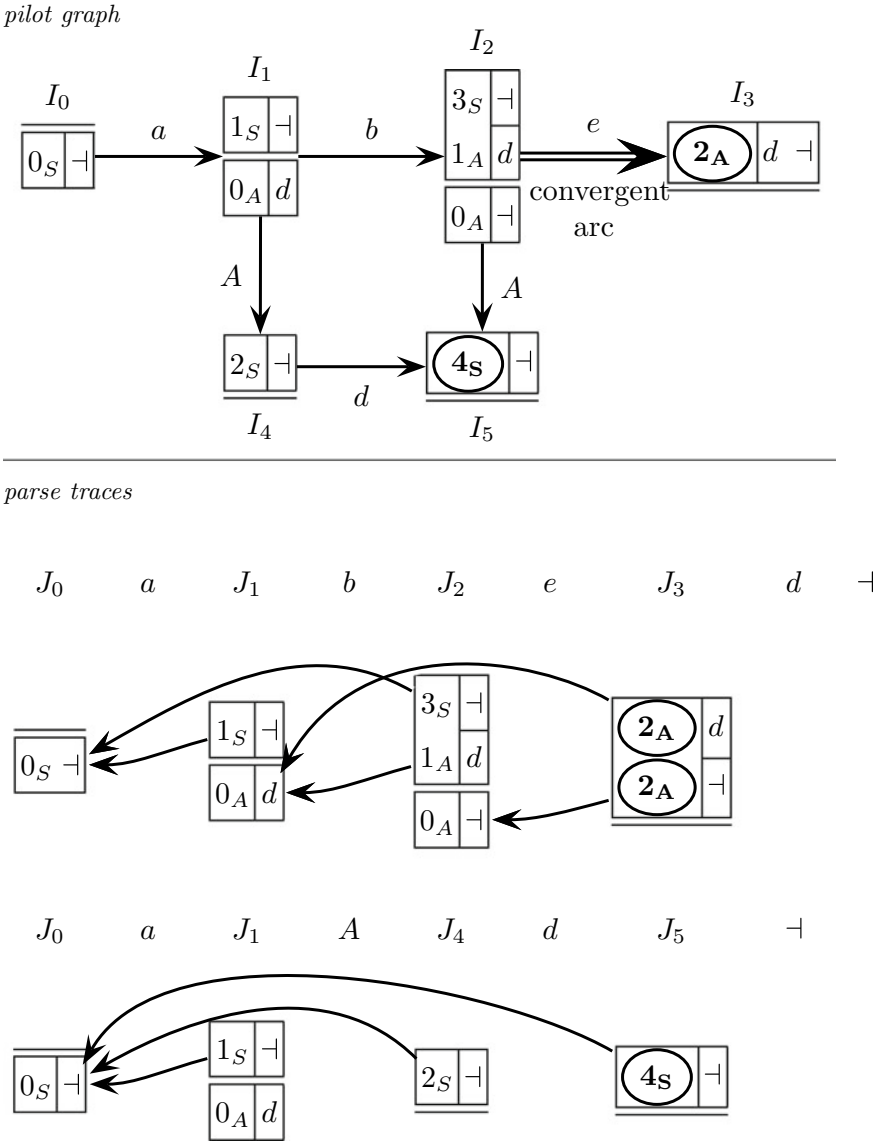


Fig. 4.29 Parsing steps of the parser using a vector stack for string $a\,b\,e\,d\ \dashv$ recognized by the net in Fig. 4.23, with the pilot here reproduced for convenience

we show a few useful *BNF* grammar transformations from *LR* (2) down to *LR* (1), which do not pretend to be exhaustive.

From Sect. 4.6.4.1 we know that the net machines of *BNF* grammars do not contain any circuits and that their state-transition graphs are trees, where the root-to-leaf paths of a machine M_A are in one-to-one correspondence with the alternative rules of nonterminal A; see for instance the machines in Fig. 4.31. First we introduce two transformations useful for lowering the value of the look-ahead length parameter k, then we hint a possible transformation for non-*LR* (k) grammars.[17]

Grammar with Reduce-Reduce Conflict

Suppose a *BNF* grammar is of type *LR* (2), but not *LR* (1). The situation is as in Fig. 4.30, where we see two rules $A \to \alpha$ and $B \to \beta$, their machines and an m-state of the pilot. We consider a reduce-reduce conflict, which is apparent as the m-state contains two reduction candidates, with final states f_A and f_B, that have the same look-ahead, say character a. Since the grammar is *BNF*, it is convenient to represent machine states with marked rules as mentioned in Sect. 4.5 on p. 237, enriched with the look-ahead. In this case we have:

reduction	*reduction*	*representation*
$A \to \alpha \bullet,\ \{\ a\ \}$	$B \to \beta \bullet,\ \{\ a\ \}$	*marked rule*
$\langle\ f_A,\ a\ \rangle$	$\langle\ f_B,\ a\ \rangle$	*state candidate*

Both marked rules denote reductions as the bullet $\bullet$ is at their end and have the same look-ahead set $\{\ a\ \}$. However, since the grammar is assumed to be of type *LR* (2), the candidates are surely discriminated by the characters that follow the look-ahead character a.

To obtain an equivalent *LR* (1) grammar, we apply a transformation called *early scanning*. The idea is to lengthen the right part of certain rules by appending to them the character a causing the conflict. In this way, the new rules will have in their look-ahead sets the characters that follow symbol a, which from the *LR* (2) hypothesis do not overlap.

More precisely, the transformation introduces two new nonterminals $\langle\ A\,a\ \rangle$, $\langle\ B\,a\ \rangle$, and their rules, as below:

$$\langle\ A\,a\ \rangle \to \alpha\,a \qquad\qquad \langle\ B\,a\ \rangle \to \beta\,a$$

For preserving grammar equivalence, we then have to adjust the rules containing nonterminals A or B in their right parts. The transformation must ensure that the derivation below:

$$\langle\ A\,a\ \rangle \stackrel{+}{\Rightarrow} \gamma\,a$$

[17] For a broader discussion of all such grammar transformations, we refer to [21].

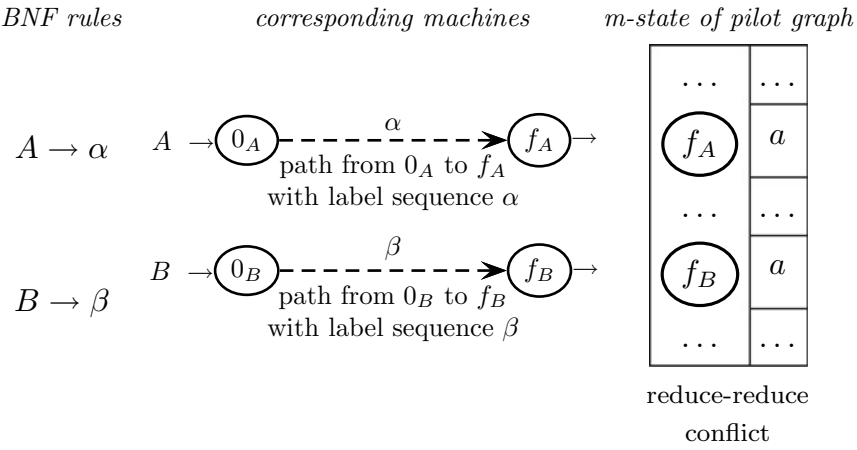


Fig. 4.30 *BNF* grammar not of type *LR* (1) with a reduce-reduce conflict

exists if, and only if, the original grammar has the following derivation:

$$A \stackrel{+}{\Rightarrow} \gamma$$

and character a can follow nonterminal A. The next Example 4.51, with both marked rules and pilot graphs, instances an early scanning case.

Example 4.51 (*early scanning*) Consider the *BNF* grammar G_1 below:

$$\begin{cases} S \to A\,b\,b \\ S \to B\,b\,c \end{cases} \qquad \begin{cases} A \to a\,A \\ B \to a\,B \end{cases} \qquad \begin{cases} A \to a \\ B \to a \end{cases}$$

Grammar G_1 is of type *LR* (2), but not *LR* (1) because it clearly has a reduce-reduce conflict between marked rules $A \to a\,\bullet,\ \{\ b\ \}$ and $B \to a\,\bullet,\ \{\ b\ \}$: the look-ahead is b for both ones. The situation is as in Fig. 4.31, where the conflict appears in the pilot m-state I_1. By increasing the length parameter to $k = 2$, we see that such reductions, respectively, have look-ahead $\{\ b\,b\ \}$ and $\{\ b\,c\ \}$, which are disjoint strings. Thus, by applying early scanning, we obtain the *LR* (1) grammar below:

$$\begin{array}{lll} S \to \langle\, A\,b\,\rangle\ b & \langle\, A\,b\,\rangle \to a\ \langle\, A\,b\,\rangle & \langle\, A\,b\,\rangle \to a\,b \\ S \to \langle\, B\,b\,\rangle\ c & \langle\, B\,b\,\rangle \to a\ \langle\, B\,b\,\rangle & \langle\, B\,b\,\rangle \to a\,b \end{array}$$

For the two reductions, the marked rules are $\langle\, A\,b\,\rangle \to a\,b\,\bullet,\ \{\ b\ \}$ and $\langle\, B\,b\,\rangle \to a\,b\,\bullet,\ \{\ c\ \}$, which do not conflict because their look-ahead sets are disjoint. Now the situation becomes as in Fig. 4.32, where no conflict occurs in the m-states I_1 or I_2, which replace the conflicting one in Fig. 4.31. Therefore, after the early scanning

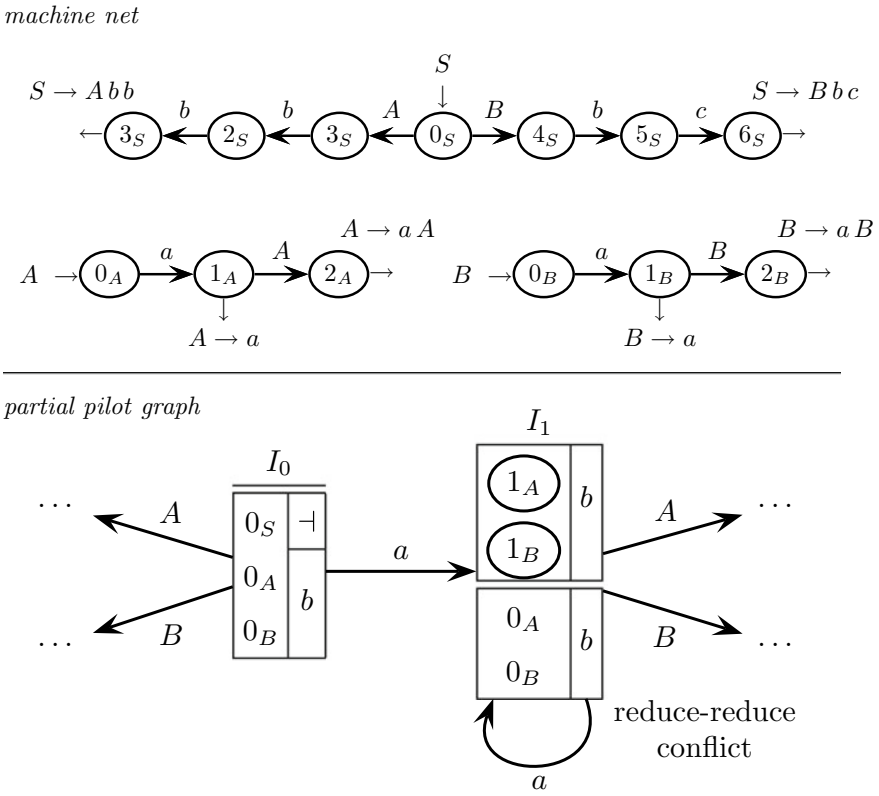


Fig. 4.31 *BNF* grammar G_1 of Example 4.51 before early scanning

transformation, the reduce-reduce conflict has vanished and the grammar has become of type *LR* (1). □

Grammar with a Shift-Reduce Conflict

Another common source of conflict occurs when an m-state of the pilot contains two marked rules, which are a shift and a reduction candidate, as below:

$$A \to \alpha \bullet a\ \beta,\ \pi \qquad\qquad B \to \gamma \bullet,\ \{\ a\ \}$$

These candidates cause a shift-reduce conflict, due to character a: a shift on character a is enabled by candidate $A \to \alpha \bullet a\ \beta$; and character a belongs to the look-ahead set $\{\ a\ \}$ of the reduction candidate $B \to \gamma \bullet$ (the look-ahead set π of the shift candidate is not relevant for the conflict).

How to fix the conflict: create a new nonterminal $\langle\ B\ a\ \rangle$ to do the same service as nonterminal B followed by terminal a; and replace rule $B \to \gamma$ with rule $\langle\ B\ a\ \rangle \to \gamma\ a$. For consistency, adjust all the rules having nonterminal B in their right parts so

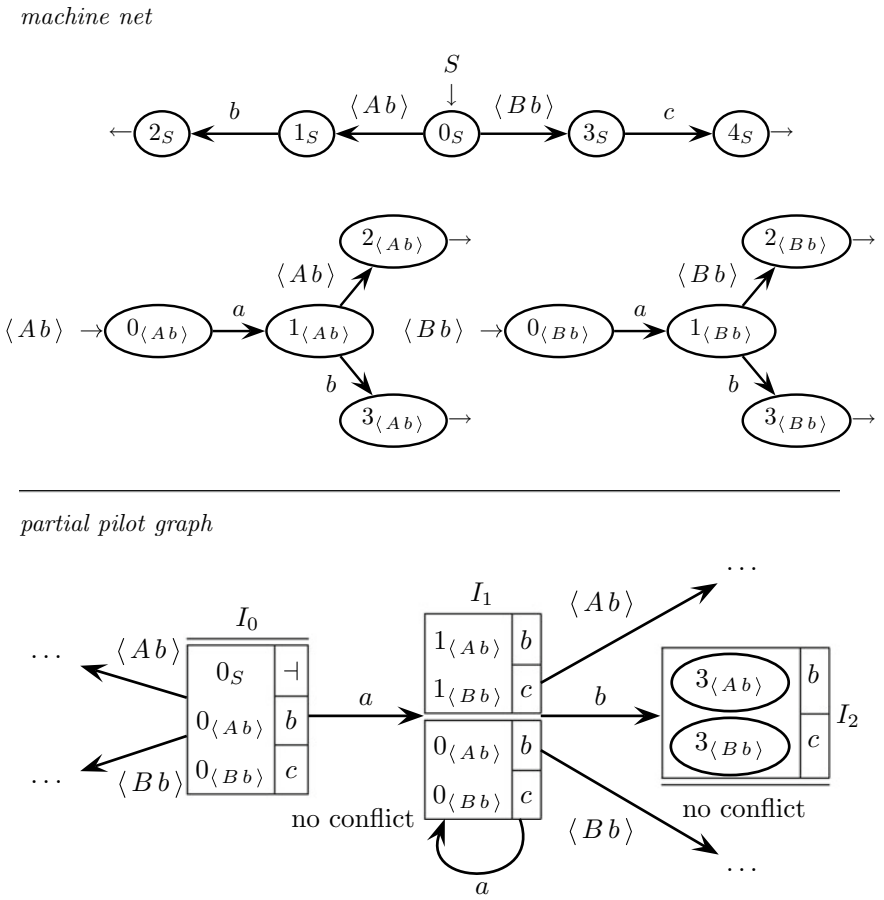


Fig. 4.32 *BNF* grammar G_1 of Example 4.51 after early scanning

as to preserve grammar equivalence. For instance, if there is a rule like $X \to B\, a\, c$, replace it with rule $X \to \langle\, B\, a\,\rangle\; c$. Since we assume the grammar is of type *LR* (2), the look-ahead set of rule $\langle\, B\, a\,\rangle \to \gamma\, a$ may not include character a,[18] and so the conflict disappears.

The transformation is more involved if the character a, which causes the conflict between candidates $A \to \alpha \bullet a\, \beta,\ \pi$ and $B \to \gamma \bullet,\ \{\, a\,\}$, is produced as initial character by a derivation from a nonterminal C, which immediately follows nonterminal B in a sentential form, like:

$$S \overset{+}{\Rightarrow} \ldots\, B\, C \ldots \overset{+}{\Rightarrow} \ldots\, B\, a\, \ldots\ldots$$

[18] More generally, in this case the *LR* (2) assumption implies that the look-ahead set (with $k = 1$) of rule $\langle\, B\, a\,\rangle \to \gamma\, a$ and the set $Ini\left(L\,(\beta) \cdot \pi\right)$ may not overlap.

Transformation: create a new nonterminal, named $\langle\, a/_L C \,\rangle$, to generate the strings obtained from those derived from C by cutting prefix a. Formally:

$$\langle\, a/_L C \,\rangle \stackrel{+}{\Rightarrow} \gamma \qquad \text{if and only if} \qquad C \stackrel{+}{\Rightarrow} a\,\gamma \quad \text{in the original grammar}$$

Then, the affected grammar rules are adjusted so as to preserve equivalence. Now, the grammar obtained is amenable to early scanning, which will remove the conflict. This transformation is called *left quotient*, since it is based on the operation $/_L$ of the same name, defined on p. 17.

The next Example 4.52 shows the transformation. For brevity, we represent it only with marked rules.

Example 4.52 (*left quotient preparation for early scanning*) Consider the *BNF* grammar G_2 below:

$$\begin{array}{llll} S \to A\,d & A \to a\,b & & \\ S \to B\,C & B \to a & C \to b\,C & C \to c \end{array}$$

Grammar G_2 is of type $LR\,(2)$, but not $LR\,(1)$ as evidenced by the following shift-reduce conflict, due to character b:

$$A \to a \;\bullet\; b,\ \{\, d \,\} \qquad\qquad B \to a\,\bullet,\ \{\, b,\ c \,\}$$

After the left quotient transformation applied to nonterminal C, we obtain the grammar below (nonterminal C is left-quotiented by b and c):

$$\begin{array}{llll} S \to A\,d & A \to a\,b & & \\ S \to B\,b\,\langle\, b/_L C \,\rangle & B \to a & \langle\, b/_L C \,\rangle \to b\,\langle\, b/_L C \,\rangle & \langle\, b/_L C \,\rangle \to c \\ S \to B\,c\,\langle\, c/_L C \,\rangle & & \langle\, c/_L C \,\rangle \to \varepsilon & \end{array}$$

Now, the shift-reduce conflict can be removed by applying the early scanning transformation, as done in Example 4.51, by means of two more nonterminals $\langle\, B\,b \,\rangle$ and $\langle\, B\,c \,\rangle$, as follows:

$$\begin{array}{llll} S \to A\,d & A \to a\,b & & \\ S \to \langle\, B\,b \,\rangle\,\langle\, b/_L C \,\rangle & \langle\, B\,b \,\rangle \to a\,b & \langle\, b/_L C \,\rangle \to b\,\langle\, b/_L C \,\rangle & \langle\, b/_L C \,\rangle \to c \\ S \to \langle\, B\,c \,\rangle\,\langle\, c/_L C \,\rangle & \langle\, B\,c \,\rangle \to a\,c & \langle\, c/_L C \,\rangle \to \varepsilon & \end{array}$$

The shift-reduce conflict has vanished, and there are not reduce-reduce conflicts. In fact, the only rules that reduce together are $A \to a\,b\,\bullet,\ \{\, d \,\}$ and $\langle\, B\,b \,\rangle \to a\,b\,\bullet,\ \{\, b,\ c \,\}$, but their look-ahead sets (with $k = 1$) are disjoint. Therefore, the grammar has the $LR\,(1)$ property. □

To sum up, grammar transformations based on left quotient and early scanning can help to lower the value of k, for a given grammar meeting the *LR* condition with parameter k larger than one. Actually, such transformations are special instances of an important theoretical result: for any $LR(k)$ grammar with $k > 1$, there is an equivalent $LR(1)$ grammar (see below Property 4.94 on p. 349). Notice, however, that the $LR(1)$ grammar may be much larger than the given $LR(k)$ grammar.

Transformation of Non-*LR* (k) Grammars

We have seen some transformations that allow to reduce the look-ahead length k of a given $LR(k)$ grammar. Unfortunately, if the language is deterministic but the given grammar is not of type $LR(k)$ for some $k \geq 1$, as for instance if it is ambiguous, then we do not have any systematic way to turn the grammar into an $LR(k)$ one for some $k \geq 1$. It remains the possibility to study the languages generated by the nonterminals involved in parsing conflicts, to identify conflict causes and finally to adjust such subgrammars for the $LR(1)$ property.

Quite often, an effective grammar transformation is to turn left-recursive rules or derivations into right-recursive ones; see Sect. 2.5.13.8 on p. 76. The reason stems from the fact that a *LR* parser carries on multiple computations until one of them reaches a reduction, which is deterministically decided and applied. A right-recursive rule (more generally a derivation) has the effect of delaying the decision moment, whereas a left-recursive rule does the opposite. In other words, with right-recursive rules the automaton accumulates more information on stack before reducing. Truly, the stack grows larger with right-recursive rules, but such an increase in memory occupation is negligible for modern computers.

4.7 Deterministic Top-Down Parsing

A simpler and very flexible top-down parsing method, traditionally called $ELL(1)$,[19] applies if an $ELR(1)$ grammar satisfies further conditions. Although less general than the $ELR(1)$, this method has several assets, primarily the ability to anticipate parsing decisions thus offering a better support for syntax-directed translation, and to be implemented by a neat modular structure made of recursive procedures that mirror the graphs of network machines. We informally introduce the idea by an example.

[19] The acronym means Extended, Left to right, Leftmost, with length of look-ahead equal to one. Deterministic top-down parsers were among the first to be constructed by compilation pioneers, and their theory for *BNF* grammars was shortly after developed by [22,23]. A sound method to extend such parsers to *EBNF* grammars was popularized by [24] recursive descent compiler, systematized in the book [25], and included in widely known compiler textbooks, e.g., [17], where top-down deterministic parsing is presented independently of shift-reduce methods. Taking a novel approach from [26], this section shows that top-down parsing for *EBNF* grammars is a corollary of the $ELR(1)$ parser construction that we have presented above. For pure *BNF* grammars, the relationship between $LR(k)$ and $LL(k)$ grammar and language families has been carefully investigated in the past, in particular by [27].

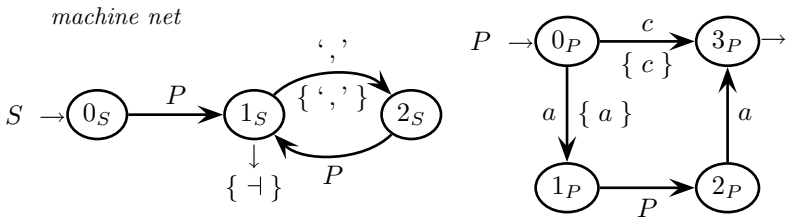


Fig. 4.33 Machine net for lists of balanced strings (Example 4.53). On the arcs and final darts exiting furcating nodes, the tokens within braces (so-called *guide sets*, see p. 299 or p. 307) tell the parser which branch to follow

Example 4.53 (*list of balanced strings*) The machine net in Fig. 4.33, equivalent to the *EBNF* grammar:

$$S \to P \left(\text{‘,’} \; P \right)^* \qquad\qquad P \to a \; P \; a \;\mid\; c$$

generates the language $\{ a^n \, c \, a^n \mid \; n \geq 0 \} \left(\text{‘,’} \; \{ a^n \, c \, a^n \mid \; n \geq 0 \} \right)^*$. Intuitively, a deterministic top-down parser is a *goal-oriented* or *predictive* algorithm. Given a source string x, the initial goal is to find a leftmost derivation from axiom S to x, which entails to choose one of the arcs leaving 0_S. Clearly, the choice must agree with the current input token: thus, for string $x = a \, c \, a \;\dashv$, current parser state 0_S and current token a, the only possibility is arc $0_S \xrightarrow{P} 1_S$, which *invokes* machine M_P, i.e., the activation of machine M_S is suspended in the state 0_S, to be named the *return state*, and state 0_P becomes active.

Two moves of M_P are open, but only arc $0_P \xrightarrow{a} 1_P$ agrees with the input token, which is then consumed. In the state 1_P with token c, machine M_P is invoked a second time (notice that the preceding activation of M_P has not been completed), setting state 1_P as return state; the parser enters state 0_P, this time choosing move $0_P \xrightarrow{c} 3_P$. The steps made so far spell the derivation $S \Rightarrow P \; \ldots \Rightarrow a \; P \; \ldots \Rightarrow a \, c \; \ldots$. Since the current state 3_P is final, the last activation of M_P terminates, and the parser returns to the (most recent) return state, 1_P, reactivating machine M_P, and moves over arc $1_P \xrightarrow{P} 2_P$ to 2_P. The next move consumes token a (the last), and enters state 3_P, final. The only pending return state 0_S is resumed, moving over arc $0_S \xrightarrow{P} 1_S$ to 1_S, final. The string is accepted as legal since it has been entirely scanned (equivalently the current token is $\dashv$), the current state is final for the axiom machine, and there are no pending return states. The complete leftmost derivation is $S \Rightarrow P \Rightarrow a \; P \; a \Rightarrow a \, c \, a$. On the other hand, if the input were $a \, c \, a \; , \; c \; \dashv$, after processing $a \, c \, a$ the parser is in the state 1_S, the next token is a comma, and arc $1_S \xrightarrow{\cdot} 2_S$ is taken. □

Syntactic Procedures

Our talking of machines invoked, suspended and returned to, suggests that a machine is similar to a subroutine, and hints to an implementation of the top-down parser by means of a set of so-called syntactic procedures, one per nonterminal symbol. The

```
    procedure S
LOOP: call P
1:      if cc = '⊣' then
            accept
            stop
2:      else if cc = ',' then
            cc := next
            goto LOOP
3:      else
            error
        end if
    end procedure
```

```
    procedure P
1:      if cc = 'a' then
            cc := next
            call P
1a:         if cc = 'a' then
                cc := next
1b:         else
                error
            end if
2:      else if cc = 'c' then
            cc := next
3:      else
            error
        end if
    end procedure
```

Fig. 4.34 Syntactic procedures of a recursive descent parser (Example 4.75 below)

syntactic procedures for the net in Fig. 4.33 are listed in Fig. 4.34. Parsing starts in the axiomatic procedure *S* and scans the first character of the string. Function *next* invokes the scanner or lexical analyzer, and returns the next input token. Comparing the procedure and the corresponding machine, we see that the procedure control-flow graph reproduces (details apart) the machine state-transition graph. We make the correspondence explicit. A machine state is the same as a program point inside the procedure code, a nonterminal label on an arc is the same as a procedure call, and the return state corresponds to the program point following the call; furcation states correspond to conditional statements. The analogy goes further: an invoked procedure, like a machine, can call another procedure, and such nested calls can reach unbounded depth if the grammar has recursive rules. Accordingly, we say that the parsers operate by *recursive descent*.

Deterministic Choice at Furcation Points

We have skipped over a critical point: if the current state bifurcates on two arcs such that at least one label is nonterminal, how do we decide which branch to take? Similarly, if the state is final and an arc exits from it, how do we decide whether to terminate or to take the arc? More generally, how can we make sure that the goal-directed parser is deterministic in every furcating state? Deferring to later sections

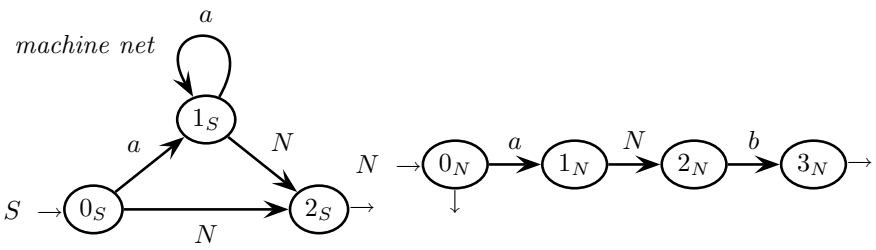


Fig. 4.35 Net of Example 4.54. The bifurcation in the node 0_S causes a nondeterministic choice (see also the *ELR* (1) pilot graph on p. 289)

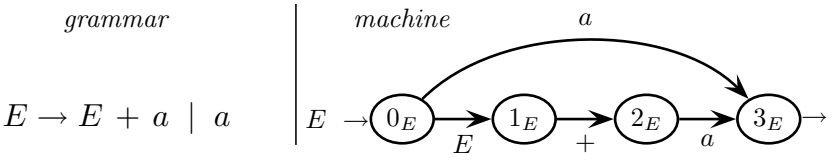


Fig. 4.36 A left-recursive grammar and machine that makes top-down deterministic choices impossible. The case of left-recursive grammars is discussed at length on p. 286 and p. 309

the full answer, we anticipate some simple examples that hint to the necessity of restrictive conditions to be imposed on the net, if we want the parser to be deterministic.

Example 4.54 (*choices at furcating states*) The net in Fig. 4.35 defines language $\{ a^* a^n b^n \mid \ n \geq 0 \}$. In the state 0_S with an a as current token, the choice of either arc leaving the state is nondeterministic, since the arc invoking N transfers control to state 0_N, from where the a-labeled arc to 1_N originates. Differently stated, the syntactic procedure S is unable to decide whether to call procedure N or to move over arc $0_S \xrightarrow{a} 1_S$, since both choices are compatible with an input string such as $a\,a\,a\,b\,\ldots$. To make the right decision, one has to look a long distance ahead of the current token, to see if the string is, say, $a\,a\,a\,b\ \dashv$ or $a\,a\,a\,b\,b\,b\ \dashv$. We shall see that this confusing situation is easily diagnosed by examining the macro-states of the *ELR* (1) pilot. □

The next example of nondeterministic bifurcation point is caused by a left-recursive rule in the grammar depicted in Fig. 4.36. In the state 0_E, both arcs agree with the current token a, and the parser cannot decide whether to call machine M_E or to move to state 3_E by a terminal shift.

Both examples exhibit non-ambiguous grammars that are *ELR* (1), and thus witness that the deterministic top-down method is less general than the bottom-up one. We are going to state further conditions (named *ELL* (1)) on the *ELR* (1) pilot graph, that ensure deterministic top-down parsing.

4.7.1 *ELL* (1) Condition

Before we characterize top-down deterministically parsable grammars or networks, we recall the multiple-transition and convergence property. Given an *ELR* (1) net $\mathcal{M}$ and its *ELR* (1) pilot $\mathcal{P}$, a macro-state has the multiple-transition property (p. 253) *MTP* if it contains two candidates such that from their states two identically labeled arcs originate. Here we are more interested in the opposite case and we say that an m-state has the *single-transition property* (*STP*) if the *MTP* does not hold for the m-state.

Definition 4.55 (*ELL* (1) *condition*) A machine net $\mathcal{M}$ meets the *ELL* (1) condition if the following three clauses are[20] satisfied:

1. there are no left-recursive derivations
2. the net meets the *ELR* (1) condition, i.e., it does not have either shift-reduce, or reduce-reduce, or convergence conflicts (p. 253)
3. the net has the single-transition property (*STP*) □

Example 4.56 (*clauses of the ELL* (1) *condition*) We check that the machine net that introduced top-down parsing on p. 283, reproduced in Fig. 4.37 for convenience, meets the three conditions. First, the grammar does not admit any left-recursive derivation, since neither nonterminal S nor nonterminal P derives a string that starts with S and P, respectively. Second, the pilot shown in Fig. 4.37 does not have conflicts, therefore it meets the *ELR* (1) condition. Third, every m-state of the pilot has the *STP*. Let us check it just on the arcs outgoing from m-state I_3: the arc to I_4 matches only one arc of the net, i.e., $1_P \xrightarrow{P} 2_P$; similarly, the arc to I_6 matches $0_P \xrightarrow{a} 1_P$, and the arc to I_8 matches $0_P \xrightarrow{c} 3_P$ only.

We have seen the recursive descent parser for this net in Fig. 4.34. □

Some previous examples that raised difficulties for deterministic decisions at furcation points are now revisited to illustrate violations of the *ELL* (1) condition.

Clause 1: the grammar in Fig. 4.36 on p. 285 has an immediate left-recursive derivation on nonterminal E; but also non-immediate left-recursive derivations are excluded by clause 1. To check if a grammar or net has a left-recursive derivations, we can use the simple method described in Chap. 2 on p. 41. We mention that certain types of left-recursive derivations necessarily violate also clause 3 or clause 2; in other words, the three clauses overlap to some extent.

Moreover, if a grammar violates clause 1, it is always possible to change it in such a way that left-recursive derivations are turned into right-recursive ones, by applying

[20] The historical acronym "*ELL* (1)" has been reused over and over in the past by several authors with slight different meanings. We hope that reusing again the acronym will not be considered an abuse.

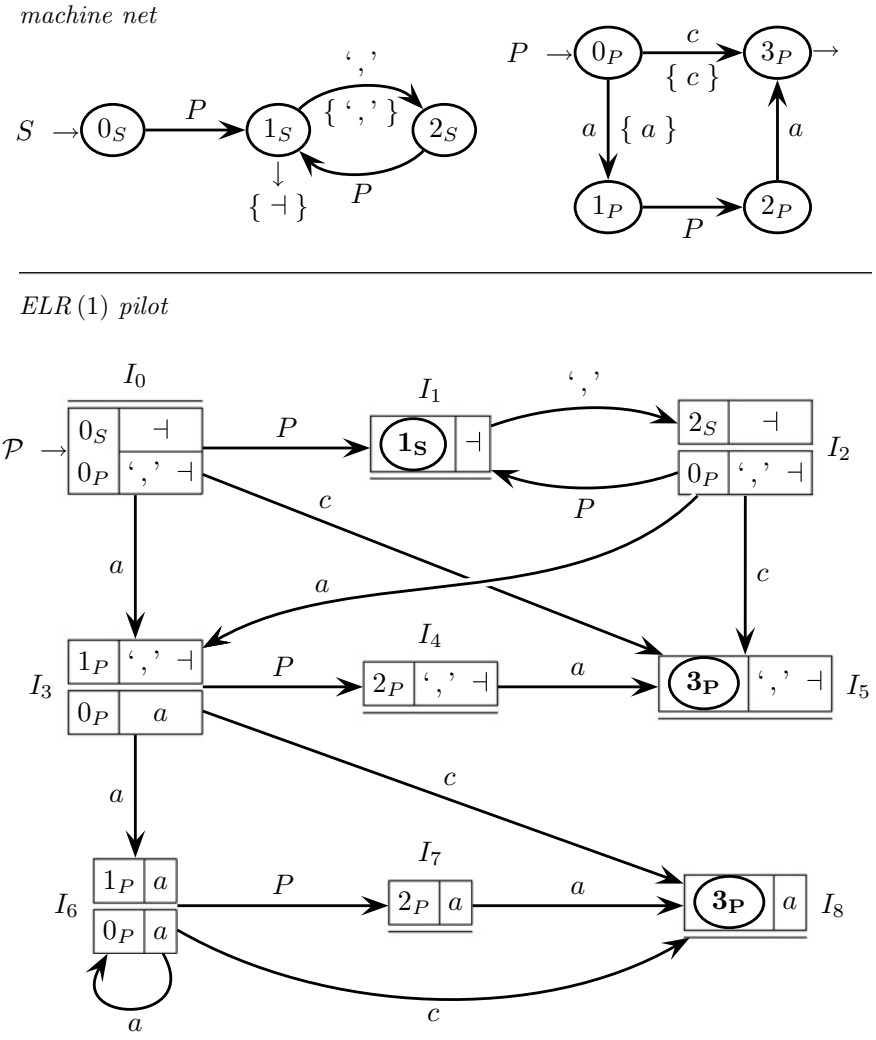


Fig. 4.37 Machine net (reproduced from Fig. 4.33 on p. 283) and its *ELR* (1) pilot. The *ELL* (1) condition is satisfied

the conversion method presented on p. 76. Therefore, in practice the condition that excludes left-recursive derivations is never too compelling for the compiler designer.

Discussion of the Single-Transition Property

The last clause of Definition 4.55 states that the net must satisfy *STP*; we discuss some cases of violation.

Example 4.57 (*violations of STP in ELR* (1) *nets*) Three cases are examined, none of them left-recursive. First, the net in Fig. 4.38 has two candidates in the base of m-state I_1 (and also in other m-states). This says that the bottom-up parser has to carry on two simultaneous attempts at parsing until a reduction takes place, which by the *ELR* (1) property is unique, since the pilot does not have either shift-reduce, or reduce-reduce, or convergence conflicts. The failure of the *STP* clause is consistent with the fact that a top-down parser cannot carry on more than one parsing attempt at a time, since just one syntactic procedure is active at any time, and its run-time configuration consists of only one machine state.

Second, an older example, the net in Fig. 4.23 on p. 265 illustrates the case of a candidate in the base of an m-state, I_3, that is entered by a convergent arc. Third, we observe the net of grammar $S \to b\ (a \mid S\,c) \mid a$ in Fig. 4.39. The pilot has two convergent transitions, therefore it does not have the *STP*, yet it contains only one candidate in each m-state base. □

For a given grammar, the *STP* condition has two important consequences: first, the range of parsing choices at any time is restricted to one; second, the presence of convergent arcs in the pilot is automatically excluded. Therefore, some technical complications needed in bottom-up parsing can be dispensed with, as we shall see.

Now, the book offers two alternative reading paths:

1. Path one goes through Sect. 4.7.2 and continues the bottom-up shift-reduce theory of previous sections; it introduces, step by step, the parser simplifications that are made possible by *ELL* (1) conditions.
2. On the other hand, path two bypasses the step-by-step development and jumps to the construction of top-down deterministic parsers directly from the grammar or machine net.

Readers aiming for immediate practical indications for building such parsers should skip Sect. 4.7.2 and jump ahead to Sect. 4.7.3 on p. 305. Conversely, readers interested in understanding the relationship between bottom-up and top-down parsers should continue reading the next section.

4.7.2 Step-by-Step Derivation of *ELL* (1) Parsers

Starting from the familiar bottom-up parser, we derive step by step the structural simplifications that become possible, as we add one by one the *STP* and the no left recursion conditions to the *ELR* (1) condition.[21]

To start, we focus on two important effects. First, condition *STP* permits to greatly reduce the number of m-states, down to the number of net states. Second, convergent

[21] The reader is referred to [26] for a more formal presentation of the various steps, including the proofs of their correctness.

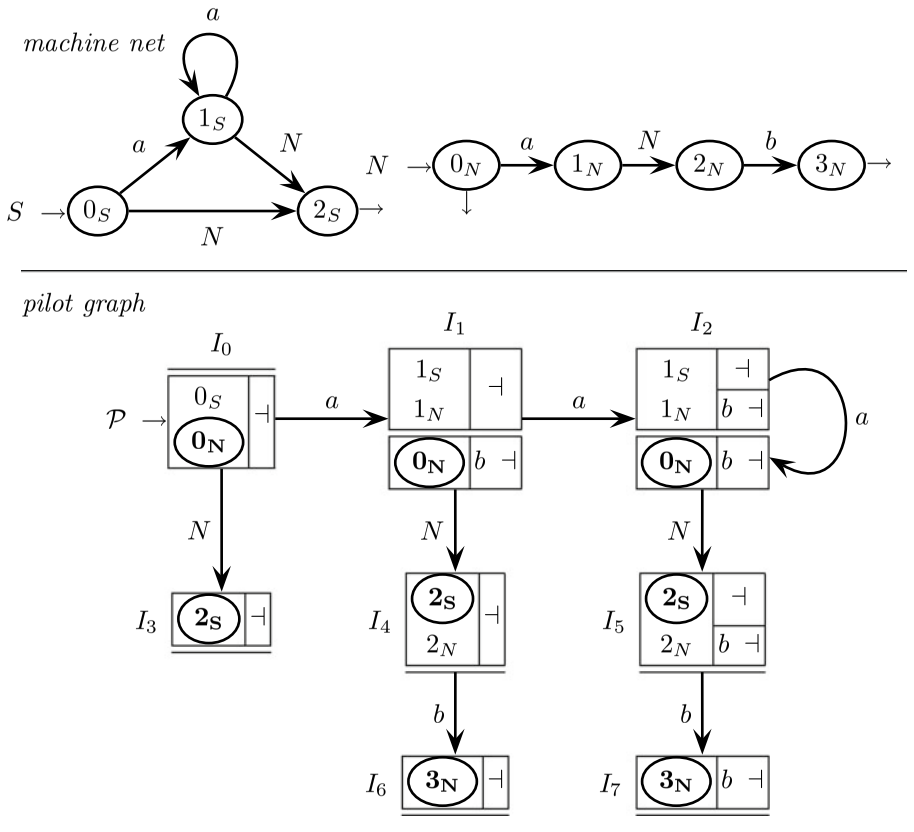


Fig. 4.38 *ELR* (1) net of Example 4.57 with multiple candidates in the m-state base of I_1, I_2, I_4 and I_5

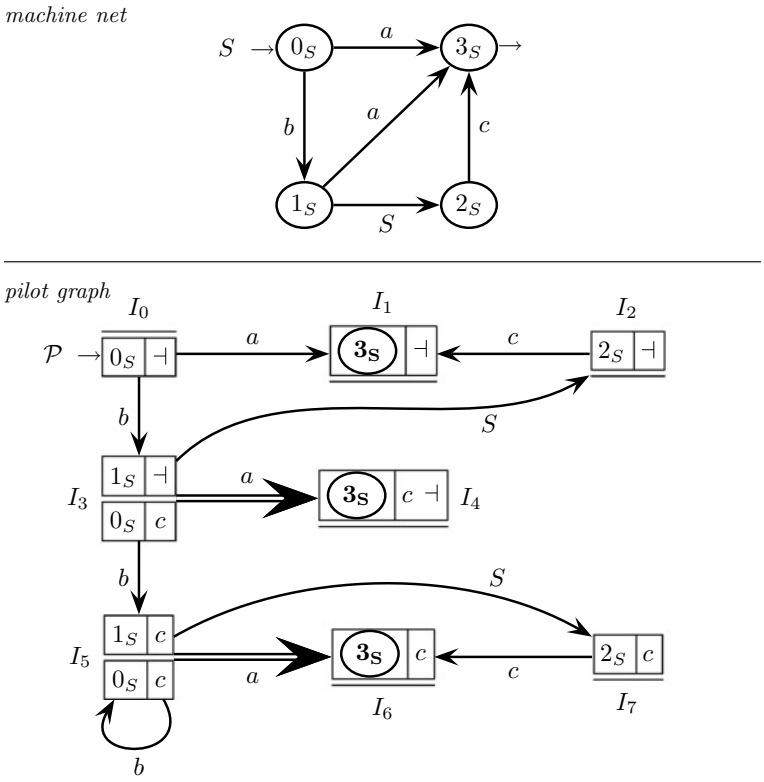


Fig. 4.39 *ELR* (1) net of Example 4.57 with just one candidate in each m-state base, yet with convergent arcs (evidenced by double arrows)

arcs disappear and consequently the chain of stack pointers can be eliminated from the parser.

Then, we add the requirement that the grammar is not left-recursive, and we obtain a parser specified by a parser control-flow graph or *PCFG*, essentially isomorphic to the machine net graphs. The result is a top-down predictive parser of the type already considered, which is able to construct the syntax tree in pre-order as it proceeds.

4.7.2.1 Single Transition Property and Pilot Compaction

Recall that the kernel of an m-state (defined on p. 253) is the projection on the first component of every candidate, i.e., the result of deleting the look-ahead sets. The relation between kernel-identical m-states is an equivalence and the set of all kernel-equivalent states forms an equivalence class.

We show that kernel-identical m-states can be safely coalesced into an m-state and that the resulting smaller pilot behaves as the original one, when used to control

a parser. We hasten to say that in general such a transformation does not work for an *ELR* (1) pilot, but it turns out to be correct under the *STP* hypothesis.

The next algorithm defines the *merging operation*, which coalesces two kernel-identical m-states I_1 and I_2, suitably adjusts the pilot graph and then possibly merges more kernel-identical m-states.

Algorithm 4.58 (*operation Merge* (I_1, I_2))

1. replace m-states I_1 and I_2 by a new kernel-identical m-state, denoted by $I_{1,2}$, where for each candidate the look-ahead set is the union of the corresponding ones in the merged m-states:

$$\langle\, p,\, \pi \,\rangle \in I_{1,2} \iff \langle\, p,\, \pi_1 \,\rangle \in I_1 \text{ and } \langle\, p,\, \pi_2 \,\rangle \in I_2 \text{ and } \pi = \pi_1 \cup \pi_2$$

2. m-state $I_{1,2}$ becomes the target for all the labeled arcs that entered I_1 or I_2:

$$I \xrightarrow{X} I_{1,2} \iff I \xrightarrow{X} I_1 \text{ or } I \xrightarrow{X} I_2$$

3. for each pair of arcs leaving I_1 and I_2, labeled X, the target m-states (which necessarily are kernel-identical) are merged:

$$\text{if } \vartheta(I_1,\, X) \neq \vartheta(I_2,\, X) \text{ then call } \mathit{Merge}\big(\vartheta(I_1,\, X),\, \vartheta(I_2,\, X)\big)$$

□

Clearly the merge operation terminates and produces a graph with fewer nodes. We observe that the above procedure resembles the one for minimizing deterministic finite automata.

By applying the *Merge* algorithm (Algorithm 4.58) to the members of every equivalence class, we construct a new graph called the *compact pilot*,[22] denoted by $\mathcal{C}$.

Example 4.59 (*compact pilot*) We reproduce (from the running example on p. 237 and 254) in Fig. 4.40 the machine net, the original *ELR* (1) pilot and, in the bottom part, the compact pilot, where the m-states have been renumbered to give evidence to the correspondence with the net states. More precisely, the mapping from the m-state bases (which by the *STP* hypothesis are singletons) to the machine states that are not initial is one to one: for instance m-state K_{1_T} is identified by state 1_T. On the other hand, the initial states 0_E and 0_T repeatedly occur in the closure part of several m-states. To obtain a complete one-to-one mapping including also the initial states,

[22] For the reader acquainted with the classical theory: *LR* (0) and *LALR* (1) pilots (which are historical simpler variants of *LR* (1) parsers not considered here, see for instance [17]) have in common with compact pilots the property that kernel-identical m-states cannot exist. However, neither *LR* (0) nor *LALR* (1) pilots have to comply with the *STP* condition.

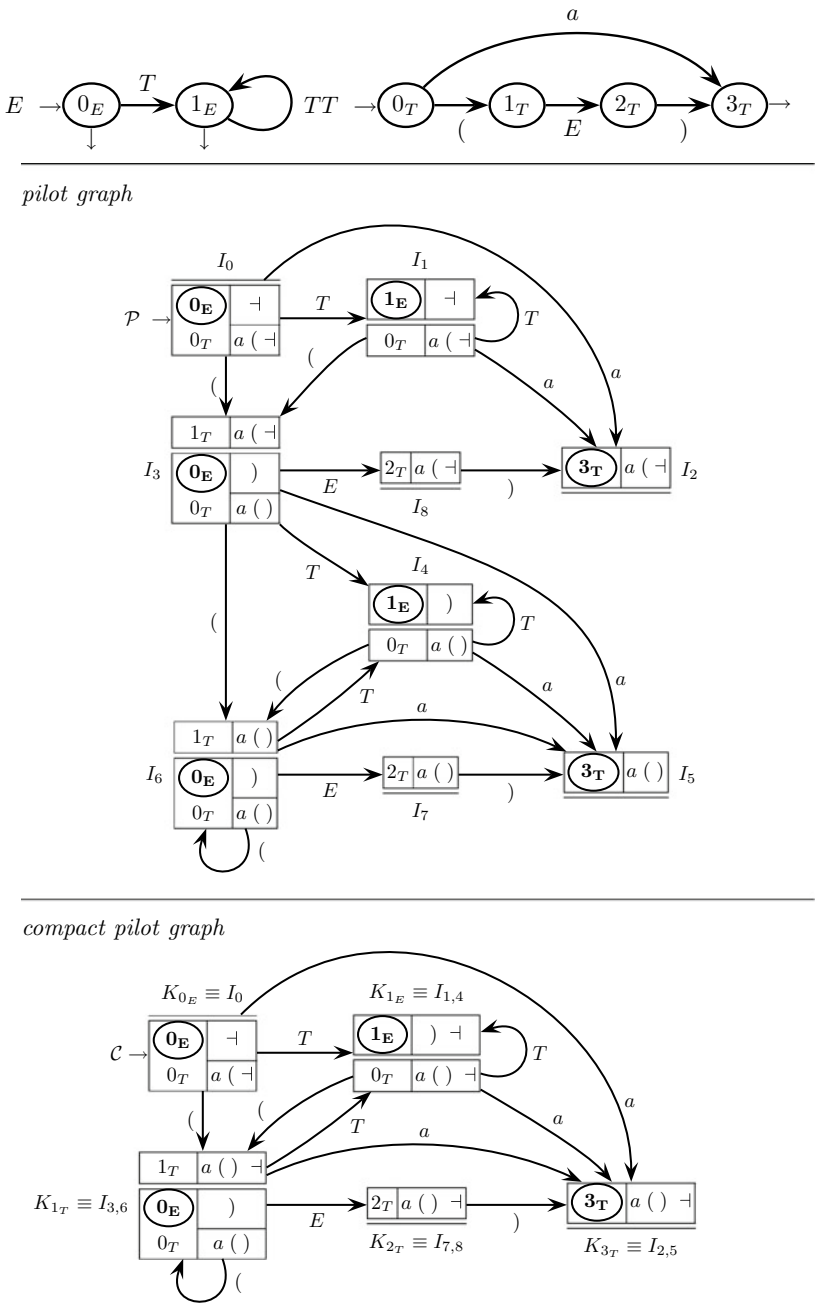


Fig. 4.40 From top to bottom: machine net $\mathcal{M}$, *ELR*(1) pilot graph $\mathcal{P}$ and compact pilot $\mathcal{C}$; the equivalence classes of m-states are: $\{ I_0 \}, \{ I_1, I_4 \}, \{ I_2, I_5 \}, \{ I_3, I_6 \}$ and $\{ I_7, I_8 \}$; the m-states of $\mathcal{C}$ are named $K_{0_E}, \ldots, K_{3_T}$ to evidence their correspondence with the states of net $\mathcal{M}$

we will in a moment extract the initial states from the m-states, to obtain the already mentioned parser control-flow graph.

The look-ahead sets in the compact pilot $\mathcal{C}$ are larger than in $\mathcal{P}$: e.g., in K_{1_T} the set in the first row is the union of the corresponding look-ahead sets in the merged m-states I_3 and I_6. This loss of precision is not harmful for parser determinism, thanks to the stronger constraints imposed by *STP*. □

The following statement says that the compact pilot can be safely used as a parser controller.

Property 4.60 (compact pilot) *Let pilots $\mathcal{P}$ and $\mathcal{C}$ be, respectively, the* ELR (1) *pilot and the compact pilot of a net $\mathcal{M}$ satisfying* STP. *The* ELR (1) *parsers controlled by $\mathcal{P}$ and by $\mathcal{C}$ are equivalent, i.e., they recognize language $L(\mathcal{M})$ and construct the same syntax tree for every string $x \in L(\mathcal{M})$.* □

We omit a complete proof[23] and we just offer some justification. First, it is possible to prove that if pilot $\mathcal{C}$ has an *ELR* (1) conflict, then also pilot $\mathcal{P}$ has, which is excluded by hypothesis. In particular, it is easy to see that an m-state $I_{1,2} = Merge(I_1, I_2)$ cannot contain a reduce-reduce conflict between two candidates $\langle p, \pi \rangle$ and $\langle r, \rho \rangle$, with $\pi \cap \rho \neq \emptyset$, such that states p and r are final and non-initial, because the *STP* rules out the presence of two candidates in the base of I_1, and $I_{1,2}$ has the same kernel.

Having excluded the presence of conflicts in pilot $\mathcal{C}$, we argue that the parsers controlled by $\mathcal{C}$ and $\mathcal{P}$ recognize the same language. Consider a string accepted by the latter parser. Since any m-state created by *Merge* encodes exactly the same cases for reduction and for shift as the merged m-states of $\mathcal{P}$, the two parsers will perform exactly the same moves. Moreover, the stack m-states and their stack candidates (p. 259) of the two parsers can only differ in the look-ahead sets, which are here irrelevant. In particular, the chains of candidate identifiers *cid* created by the parsers are identical, since the offset of a candidate $\langle q, \{ \ldots \} \rangle$ inside an m-state remains the same after merging. Therefore, at any time the two parser stacks store the same elements, and the compact parser recognizes the same strings and constructs the same tree.

To finish, it is impossible that an illegal string is accepted by the $\mathcal{C}$-based parser and rejected by the original parser, because, the first time that the $\mathcal{P}$-based parser stops by error, say, for an impossible terminal shift, also the $\mathcal{C}$ parser will stop with the same error condition.

At the risk of repetition, we stress that Property 4.60 does not hold in general for an *ELR* (1) but not *STP*-compliant pilot, because the graph obtained by merging kernel-equivalent m-states is not guaranteed to be free from conflicts.

[23] It can be found in [26].

4.7.2.2 Candidate Identifiers or Pointers Unnecessary

Thanks to the *STP* property, Algorithm 4.42 (p. 260) will be further simplified to remove the need for *cid* (or stack pointers). We recall that a *cid* was needed to find the reach of a non-empty reduction move into the stack: elements were popped until the *cid* chain reached an initial state, the end-of-list sentinel. Under *STP* hypothesis, that test is now replaced by a simpler device, to be later incorporated in the final *ELL* (1) parser (Algorithm 4.73 on p. 311). With reference to the "old" Algorithm 4.42, only shift and reduction moves are modified.

First, let us focus on the situation when the old parser, with element J on top of stack and $J_{|\,base} = \langle\, q_A,\ \ldots \rangle$, performs the shift of X (terminal or non-) from state q_A, which is necessarily non-initial; the shift would require to compute and record a non-null *cid* into the *sms* J' to be pushed on stack. In the same situation, the "new" parser behaves differently: it cancels from the top-of-stack element J all candidates other than $\langle\, q_A,\ \ldots \rangle$, since they correspond to discarded parsing alternatives. Notice that the *STP* implies that the canceled candidates are necessarily in the closure of J, hence they contain only initial states. After eliminating from the *sms* the irrelevant candidates, the new parser can identify the reduction to be made when a final state of machine M_A is entered, by using a simple rule: keep popping the stack until the first occurrence of initial state 0_A is found.

Second, consider a shift of X (terminal or non-) from an initial state $\langle\, 0_A,\ \pi \,\rangle$, which is necessarily in the closure of element J. Notice that in this case the new parser leaves element J unchanged and pushes the element $\vartheta\,(J,\ X)$ on the stack. The other candidates present in J are not canceled because they may be the origin of future nonterminal shifts.

Since *cid* are not used by the new parser, a stack element is identical to an m-state (of the compact pilot). Thus, a shift move first trims the top-of-stack element by discarding some candidates, then it pushes the input token and the next m-state. The next algorithm lists only the moves that differ from Algorithm 4.42.

Algorithm 4.61 (*pointerless parser* $\mathcal{A}_{PL}$) Let the pilot be compacted; m-states are denoted K_i and stack symbols H_i; the set of candidates of H_i is weakly included in K_i.

shift move

Let the current character be a, H be the top-of-stack element, containing candidate $\langle\, q_A,\ \pi \,\rangle$. Let $q_A \xrightarrow{a} q'_A$ and $\vartheta\,(K,\ a) = K'$ be, respectively, the state transition and m-state transition, to be applied. The shift move does:

1. if $q_A \in K_{|\,base}$ (i.e., state q_A is not initial), eliminate all other candidates from H, i.e., set H equal to $K_{|\,base}$
2. push token a on stack and get next token
3. push $H' = K'$ on stack

reduction move (non-initial state)
Let the stack be as follows:

$$H[0]\ a_1\ H[1]\ a_2\ \ldots\ a_k\ H[k]$$

Assume that the pilot chooses the reduction candidate $\langle\, q_A,\ \pi\,\rangle \in K[k]$, where state q_A is final but non-initial. Let $H[h]$ be the topmost stack element such that $0_A \in K[h]_{|\,kernel}$. The move does:

1. grow the syntax forest by applying the reduction:

$$a_{h+1}\, a_{h+2}\ \ldots\ a_k \rightsquigarrow A$$

and pop the stack symbols:

$$H[k]\ a_k\ H[k-1]\ a_{k-1}\ \ldots\ H[h+1]\ a_{h+1}$$

2. execute the nonterminal shift move $\vartheta\,(K[h],\ A)$

reduction move (initial state)
It differs from the preceding case in that, for the chosen reduction candidate $\langle\, 0_A,\ \pi\,\rangle$, the state is initial and final. Reduction $\varepsilon \rightsquigarrow A$ is applied to grow the syntax forest. Then the parser performs the nonterminal shift move $\vartheta\,(K[k],\ A)$.
nonterminal shift move
It is the same as a shift move, except that the shifted symbol is a nonterminal. The only difference is that the parser does not read the next input token at line 2 of *shift move*. □

Clearly this reorganization removes the need of the *cid* (or of the vector stack) used in previous algorithms.

Property 4.62 (pointerless pilot) *If the* ELR (1) *pilot (compact or not) of an* EBNF *grammar or machine net satisfies the* STP *condition, the pointerless parser* $\mathcal{A}_{PL}$ *of Algorithm 4.61 is equivalent to the* ELR (1) *parser* $\mathcal{A}$ *of Algorithm 4.42.* □

It is useful to support this statement by some arguments to better understand how the two algorithms are related.

First, after parsing the same string, the stacks of parsers $\mathcal{A}$ and $\mathcal{A}_{PL}$ contain the same number k of stack elements, respectively, $J[0]\ \ldots\ J[k]$ and $K[0]\ \ldots\ K[k]$. For every pair of corresponding elements, the set of states included in $K[i]$ is a subset of the set of states included in $J[i]$ because Algorithm 4.61 may have discarded a few candidates.

Second, we examine the stack elements at the same position i and $i-1$. The following relations hold:

$$\begin{gathered}\text{in } J[i],\ \text{candidate } \langle\, q_A,\ \pi,\ \sharp j \,\rangle \text{ points to } \langle\, p_A,\ \pi,\ \sharp l \,\rangle \text{ in } J[i-1]_{|\,base} \\ \text{if and only if} \\ \text{candidate } \langle\, q_A,\ \pi \,\rangle \in K[i] \text{ and the only candidate in } K[i-1] \text{ is } \langle\, p_A,\ \pi \,\rangle\end{gathered}$$

$$\begin{gathered}\text{in } J[i],\ \text{candidate } \langle\, q_A,\ \pi,\ \sharp j \,\rangle \text{ points to } \langle\, 0_A,\ \pi,\ \perp \,\rangle \text{ in } J[i-1]_{|\,closure} \\ \text{if and only if} \\ \text{candidate } \langle\, q_A,\ \pi \,\rangle \in K[i] \\ \text{and elem. } K[i-1] \text{ equals the projection of } J[i-1] \text{ on } \langle\, \text{state},\ \text{look-ahead} \,\rangle\end{gathered}$$

Then, by the way the reduction moves operate, parser $\mathcal{A}$ performs reduction $a_{h+1}\, a_{h+2}\, \ldots\, a_k \rightsquigarrow A$ if and only if parser $\mathcal{A}_{PL}$ performs the same reduction.

To complete the presentation of the new parser, we list an execution trace.

Example 4.63 (*pointerless parser trace for compact pilot*) Consider Fig. 4.40 on p. 292. Given the input string (( ) *a*) ⊣, Fig. 4.41 shows the execution trace, which should be compared with the one of the *ELR* (1) parser in Fig. 4.22 on p. 263, where *cid* are used. The same graphical conventions are used: the m-state (of the compact pilot) in each cell is framed, e.g., K_{0_S} is denoted $\boxed{0_S}$, etc.; the final candidates are encircled, e.g., $\textcircled{3_T}$, etc.; and the look-ahead is omitted to avoid clogging. So a candidate appears as a pure machine state. Of course, here there are no pointers, and the initial candidates canceled by Algorithm 4.61 from an m-state are stricken out.

Notice that, if initial candidates had not been canceled, several instances of the initial state of the active machine would occur in the stack m-states to be popped when a reduction starts, thus causing a loss of determinism. For instance, the first (from the figure top) reduction (*E*) $\rightsquigarrow T$ of machine M_T pops the three stack elements K_{3_T}, K_{2_T}, K_{1_T}, and the last one would have contained the (now stricken out) candidate 0_T, which is initial for machine M_T, but is not the correct initial state for the reduction: the correct one, unstriked, is below in the stack, in m-state K_{1_T}, which is not popped and is the origin of the nonterminal shift on T following the reduction. We point out that Algorithm 4.61 avoids to cancel an initial candidate from a stack element, if a shift move has to be later executed starting from it: see the *Shift move* case in Algorithm 4.61. The motivation for not canceling is twofold. First, these candidates will not cause any premature stop of the series of pop moves in the reductions that may come later, or said differently, they will not break any reduction handle. For instance, the first (from the figure top) shift on ' (' of machine M_T keeps both initial candidates 0_E and 0_T in the stack m-state K_{1_T}, as the shift originates from the initial candidate 0_T. This candidate will instead be canceled (i.e., it will show striked) when a shift on E is executed (soon after reduction $T\ T \rightsquigarrow E$), as the shift originates from the non-initial candidate 1_T. Second, some of such initial candidates may be needed for a subsequent nonterminal shift move. □

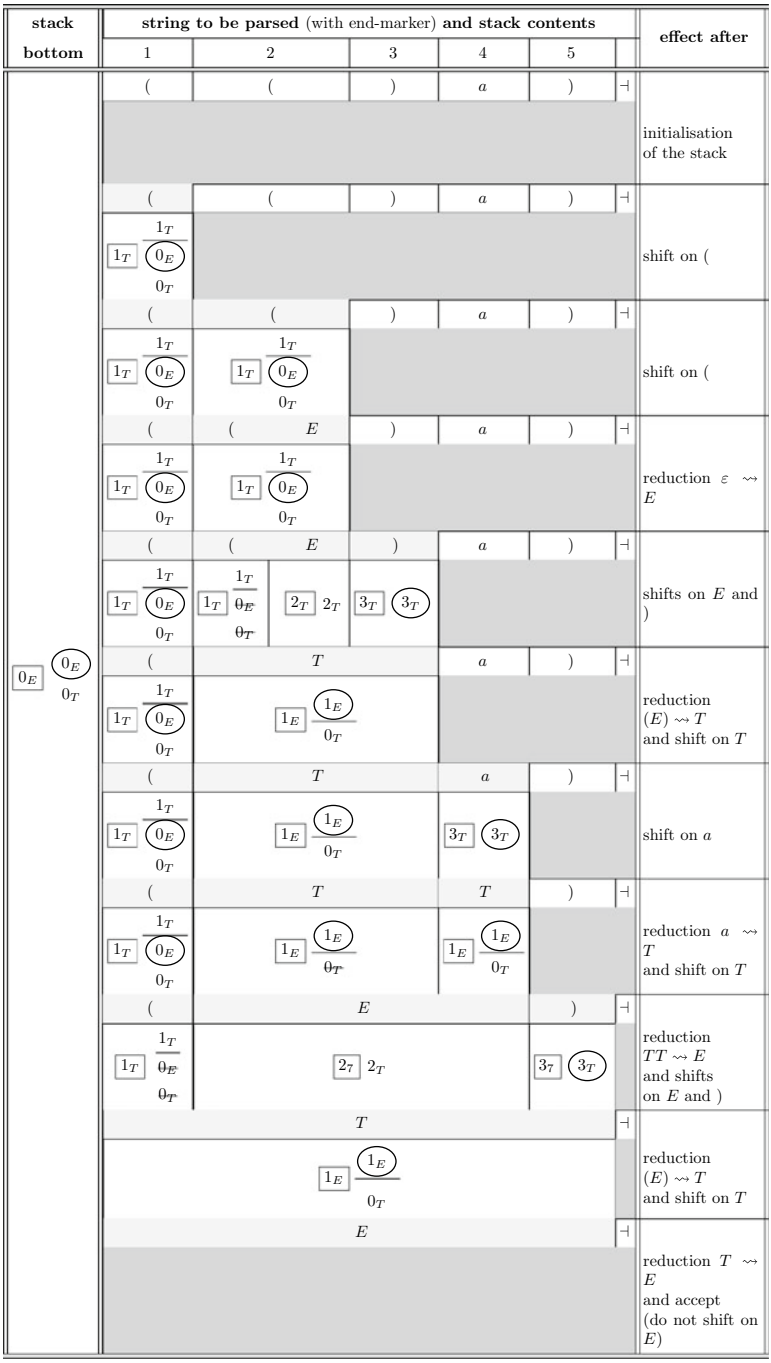


Fig. 4.41 Steps of the pointerless parsing Algorithm 4.61 $\mathcal{A}_{\mathcal{PL}}$; irrelevant and potentially confusing initial candidates are canceled by shift moves as explained in the algorithm

To sum up, we have shown that condition *STP* permits to construct a simpler shift-reduce parser, which has a smaller stack alphabet and does not need pointers to manage reductions. The same parser can be further simplified, if we make another hypothesis: that the grammar is not left-recursive.

4.7.2.3 Stack Contraction and Predictive Parser

The last development to be presented transforms the already compacted pilot graph into the control-flow graph of a *predictive* or goal-oriented parser. The way the parser uses the stack differs from the previous two models, the shift-reduce and the pointerless parsers. More precisely, now a terminal shift move, which always executes a push operation in the previous parsers, is sometimes implemented without a push and sometimes with multiple pushes. The former case happens when the shift remains inside the same machine: the predictive parser does not push an element upon performing a terminal shift, but it updates the top-of-stack element to record the new state of the active machine. Multiple pushes happen when the shift determines one or more transfers from the current machine to others: the predictive parser performs a push for each transfer.

Now, the essential information to be kept on stack is the sequence of machines that have been activated and have not reached a final state (where a reduction occurs). At each parsing time, the current or *active* machine is the one that is doing the analysis, and the *current* state is kept in the top-of-stack element. Previous non-terminated activations of the same or other machines are in a *suspended* state. For each suspended machine M_A, a stack entry is needed to store the state q_A, from where the machine will resume the computation when control is returned after performing the relevant reductions. A major advantage of predictive parsing is that the construction of the syntax tree can be anticipated: the parser can generate online the left derivation of the input.

Control-Flow Graph of Parser

Moving from the above considerations, we adjust the compact pilot graph $\mathcal{C}$ to make it isomorphic to the original machine net $\mathcal{M}$, and thus we obtain a graph named *parser control-flow graph* or *PCFG*, because it represents the blueprint of the parser code. First, every m-node of $\mathcal{C}$ that contains many candidates is split into as many nodes. Second, the kernel-equivalent nodes are coalesced and the original look-ahead sets are combined into one. The third step creates new arcs, named *call arcs*, which represent the transfers of control from a machine to another. At last, each call arc is labeled with a set of terminals, named *guide set*, which will determine the parser decision to transfer control to the called machine.

Definition 4.64 (*parser control-flow graph or PCFG*) Every *node* of the *PCFG*, denoted by $\mathcal{F}$, is identified and denoted (at no risk of confusion) by a state q of machine net $\mathcal{M}$. Moreover, every final node f_A additionally contains a set π of

terminals, named *prospect set*.[24] Such nodes are therefore associated with the pair $\langle f_A, \pi \rangle$. The prospect set is the union of the look-ahead sets π_i of every candidate $\langle f_A, \pi_i \rangle$ existing in the compact pilot graph $\mathcal{C}$, as follows:

$$\pi = \bigcup_{\forall \langle f_A, \pi_i \rangle \in \mathcal{C}} \pi_i \tag{4.13}$$

The *arcs* of pilot $\mathcal{F}$ are of two types, named *shift* and *call* (respectively depicted as solid and dashed arrows):

1. There exists in $\mathcal{F}$ a shift arc $q_A \xrightarrow{X} r_A$ with X terminal or non-, if the same arc is in the machine M_A.
2. There exists in $\mathcal{F}$ a call arc $q_A \dashrightarrow^{\gamma_1} 0_{A_1}$, where A_1 is a nonterminal possibly different from A, if arc $q_A \xrightarrow{A_1} r_A$ is in the machine M_A, hence necessarily some m-state K of pilot $\mathcal{C}$ contains candidates $\langle q_A, \pi \rangle$ and $\langle 0_{A_1}, \rho \rangle$; and the next m-state $\vartheta(K, A_1)$ contains candidate $\langle r_A, \pi_{r_A} \rangle$. □

The call arc label $\gamma_1 \subseteq \Sigma \cup \{ \dashv \}$, named *guide set*, is specified in the next definition, which also specifies the guide sets of terminal shift arcs and of the arrows (darts) that mark final states.

Definition 4.65 (*guide set*)

1. For every call arc $q_A \dashrightarrow^{\gamma_1} 0_{A_1}$ associated with a nonterminal shift arc $q_A \xrightarrow{A_1} r_A$, a terminal b is in the guide set γ_1, also written as $b \in Gui(q_A \dashrightarrow 0_{A_1})$, if, and only if, one of the following conditions holds:

$$b \in Ini\left(L\left(0_{A_1}\right)\right) \tag{4.14}$$

$$A_1 \text{ is nullable and } b \in Ini\left(L\left(r_A\right)\right) \tag{4.15}$$

$$A_1 \text{ and } L\left(r_A\right) \text{ are both nullable and } b \in \pi_{r_A} \tag{4.16}$$

$$\exists \text{ in } \mathcal{F} \text{ a call arc } 0_{A_1} \dashrightarrow^{\gamma_2} 0_{A_2} \text{ and } b \in \gamma_2 \tag{4.17}$$

2. For every terminal shift arc $p \xrightarrow{a} q$ (with $a \in \Sigma$), we set $Gui(p \xrightarrow{a} q) := \{ a \}$.
3. For every dart that tags a final node containing candidate $\langle f_A, \pi \rangle$ (with f_A final), we set $Gui(f_A \rightarrow) := \pi$. □

[24] Although traditionally the same word "look-ahead" has been used for both shift-reduce and top-down parsers, the sets differ and we prefer to differentiate their names.

Guide sets are usually represented as arc labels enclosed within braces.

Relations (4.14)–(4.16) are not recursive and respectively consider that terminal b is generated by M_{A_1} called by M_A; or by M_A but starting from state r_A; or that terminal b follows machine M_A. Equation (4.17) is recursive and traverses the net as far as the chain of call sites activated. We observe that Eq. (4.17) induces the set inclusion relation $\gamma_1 \supseteq \gamma_2$ between any two concatenated call arcs $q_A \xrightarrow{\gamma_1} 0_{A_1} \xrightarrow{\gamma_2} 0_{A_2}$.

Next, we move to a procedural interpretation of a *PCFG*: all the arcs (except nonterminal shifts) can be viewed as conditional instructions, enabled if the current character cc belongs to the associated guide set: thus a terminal shift arc labeled with $\{ a \}$ is enabled by predicate $cc \in \{ a \}$, simplified to $cc = a$; a call arc labeled with set γ represents the procedure invocation conditioned by predicate $cc \in \gamma$; a final node dart labeled with set π is interpreted as a conditional return-from-procedure instruction to be executed if $cc \in \pi$. The remaining *PCFG* arcs are nonterminal shifts, which are viewed as unconditional return-from-procedure instructions.

The essential properties of guide sets are next stated (see [26] for a proof).

Property 4.66 (disjointness of guide sets)

1. *For every node q of the* PCFG *of a grammar that satisfies the* ELL (1) *condition, the guide sets of any two arcs originating from q are disjoint.*
2. *If the guide sets of a* PCFG *are disjoint, then the machine net satisfies the* ELL (1) *condition of Definition 4.55.* □

The first statement says that, for the same state, membership in different guide sets is mutually exclusive. The second statement is the converse of the first: it makes the condition of having disjoint guide sets, a characteristic property of *ELL* (1) grammars (a marginal case contradicting statement 2 of Property 4.66 will be briefly discussed at the end of this section on p. 304). We also anticipate that statement 2 can be directly checked on the *PCFG*, thus saving the effort to build the *ELR* (1) pilot.

Example 4.67 (*PCFG of the running example*) The *PCFG* of the running example is represented in Fig. 4.42, with the same layout as the machine net for comparability. In the *PCFG* there are new nodes (here only node 0_T), which derive from the initial candidates (except those containing the axiom 0_S) extracted from the closure part of the m-states of the compact pilot $\mathcal{C}$. After creating such new nodes, the closure part of the nodes of $\mathcal{C}$ (except node I_0) becomes redundant and is eliminated from the contents of the *PCFG* nodes.

As said, the prospect sets are needed only in the final states and have the following properties (see also (4.13)):

- The prospect set of a final state that is not initial coincides with the look-ahead set of the corresponding node in the compact pilot $\mathcal{C}$. This is the case of nodes 1_E and 3_T.

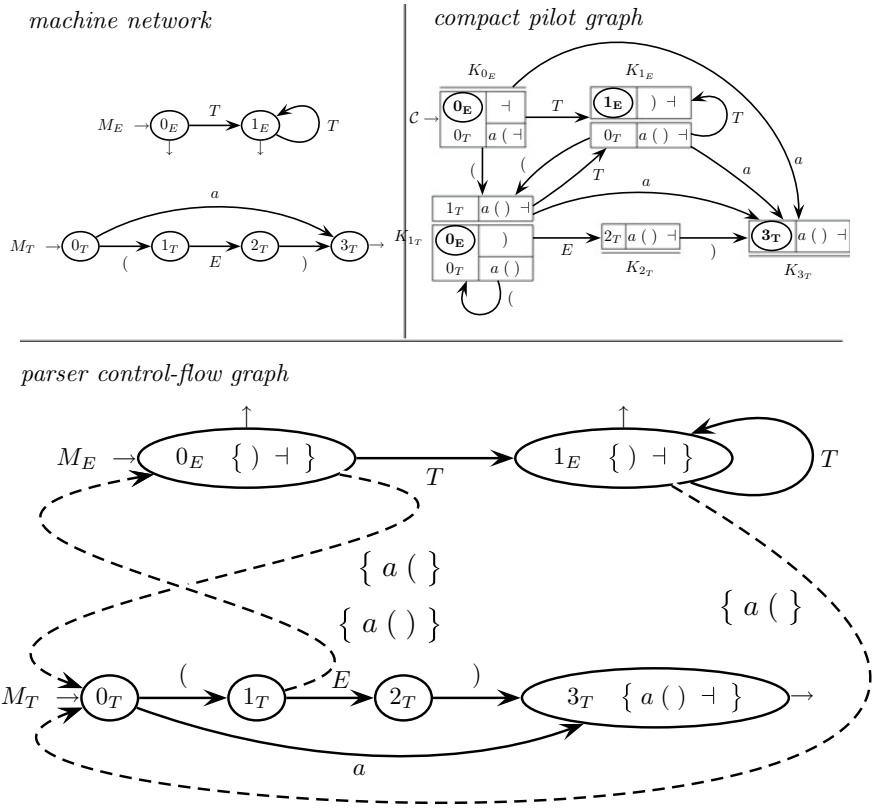


Fig. 4.42 The parser control-flow graph $\mathcal{F}$ (*PCFG*) of the running example, from the net and the compact pilot

- The prospect set of a final state that is also initial, e.g., 0_E, is the union of the look-ahead sets of every candidate $\langle\, 0_E,\, \pi \,\rangle$ occurring in $\mathcal{C}$. For instance, $\langle\, 0_E,\, \{\, \text{`)'},\, \dashv \,\}\,\rangle$ takes terminal `)' from m-state K_{1_T} and $\dashv$ from K_{0_E}.

Solid arcs represent the shifts, already present in the machine net and in the pilot graph. Dashed arcs represent the calls, labeled by guide sets, the computation of which follows:

- the guide set of call arc $0_E \dashrightarrow 0_T$ (and of $1_E \dashrightarrow 0_T$ as well) is $\{\, a,\, \text{`('} \,\}$, since both terminals $a$ and `(' can be shifted starting from state 0_T, which is the call arc destination (see (4.14))
- the guide set of call arc $1_T \dashrightarrow 0_E$ includes the following terminals:
  - a and `(', since from state $0_E$ (the call arc destination) one more call arc goes out to $0_T$ and has guide set $\{\, a,\, \text{`('} \,\}$ (see (4.17))

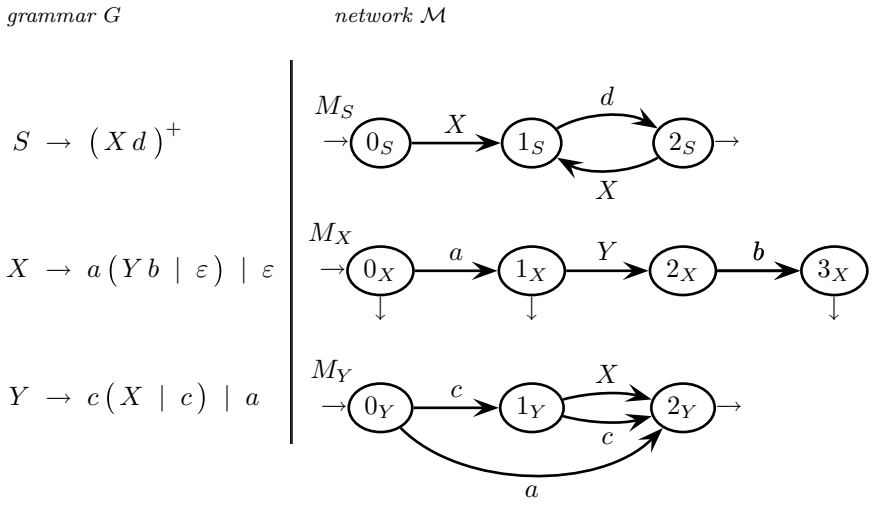


Fig. 4.43 Grammar and net of Example 4.68

– ‘) ’, since the shift arc $1_T \xrightarrow{E} 2_T$ (associated with the call arc) is in machine M_T, language $L\,(0_E)$ is nullable and it holds ‘) ’ $\in$ *Ini* (2_T) (see (4.16)).

Observe the chain of two call arcs, $1_T \dashrightarrow 0_E$ and $0_E \dashrightarrow 0_T$; relation (4.17) implies that the guide set of the former arc includes that of the latter.

In accordance with Property 4.66, the terminal labels of all the arcs (shift and call) that originate from the same node do not overlap. □

Since relation (4.15) has not been used, we illustrate it in the next example.

Example 4.68 (*from the pilot to the PCFG*) Consider the *EBNF* grammar G and net $\mathcal{M}$ in Fig. 4.43. The machine net $\mathcal{M}$ is *ELL* (1); its pilot $\mathcal{P}$ in Fig. 4.44 does not have any *ELR* (1) conflicts and also satisfies the *ELL* condition. The *PCFG* of net $\mathcal{M}$, derived from $\mathcal{P}$, is shown in Fig. 4.45. We check that the prospect sets in the *PCFG* states that result from merging the m-states of the pilot graph are the union of their look-ahead sets (see (4.13)):

- the prospect set of state 0_X is $\pi_{0_X} = \{\, b,\ d \,\}$, and is the union of the look-ahead set $\{\, b \,\}$ of candidate $\langle\, 0_X,\ \{\, b \,\}\,\rangle$ in m-state I_7, and of the look-ahead set $\{\, d \,\}$ of candidate $\langle\, 0_X,\ \{\, d \,\}\,\rangle$ in m-state I_0
- similarly, the prospect set $\pi_{1_X} = \{\, b,\ d \,\}$ is the union of the look-ahead sets of candidates $\langle\, 1_X,\ \{\, b \,\}\,\rangle$ in I_8 and $\langle 1_X,\ \{\, d \,\}\,\rangle$ in I_3
- the prospect set $\pi_{3_X} = \{\, b,\ d \,\}$ derives from candidates $\langle\, 3_X,\ \{\, b \,\}\,\rangle$ in I_{10} and $\langle\, 3_X,\ \{\, d \,\}\,\rangle$ in I_5

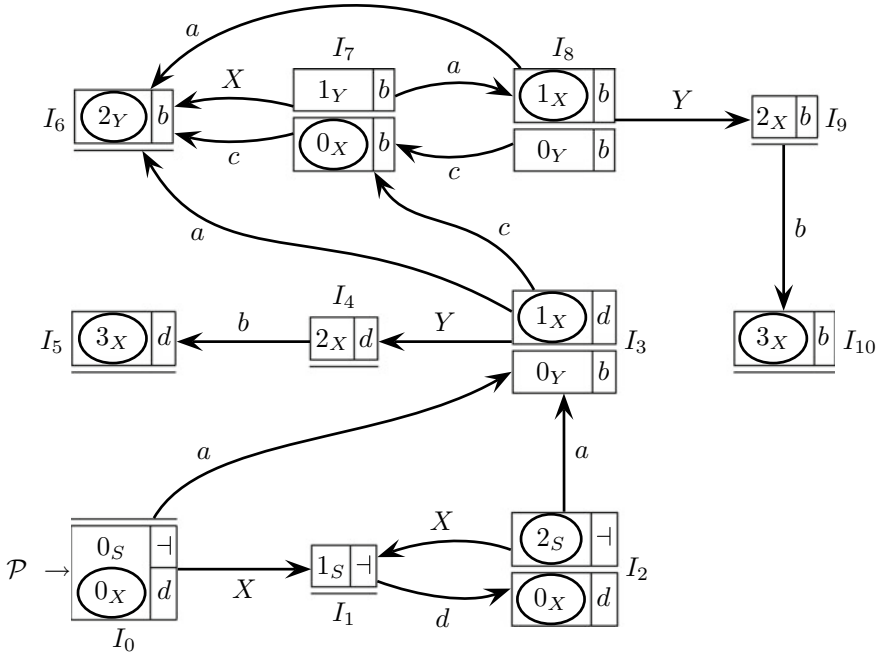


Fig. 4.44 *ELR* (1) pilot of the machine net in Fig. 4.43

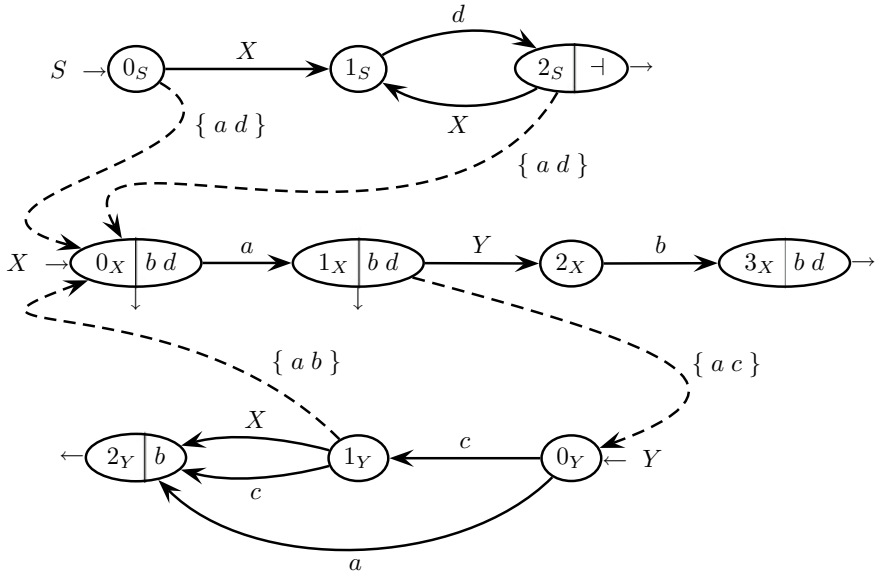


Fig. 4.45 Parser control-flow graph derived from the pilot graph in Fig. 4.44

Next, we explain how the guide sets on the call arcs are computed:

- $Gui\,(0_S \dashrightarrow 0_X)$ is the union of terminal a on the shift arc from 0_X (see (4.14)) and of terminal d on the shift arc from 1_S, which is a successor of the nonterminal shift arc $0_S \xrightarrow{X} 1_S$ associated with the call arc, because nonterminal X is nullable (see (4.15))
- $Gui\,(2_S \dashrightarrow 0_X)$ is the union of terminal a on the shift arc from 0_X (see (4.14)) and of terminal d on the shift arc from 1_S, which is a successor of the nonterminal shift arc $2_S \xrightarrow{X} 1_S$ associated with the call arc, because nonterminal X is nullable (see (4.15))
- $Gui\,(1_Y \dashrightarrow 0_X)$ is the union of terminal a on the shift arc from 0_X (see (4.14)) and of terminal b in the prospect set of 2_Y, which is the destination state of the nonterminal shift arc $1_Y \xrightarrow{X} 2_Y$ associated with the call arc, because nonterminal X is nullable and additionally state 2_Y is final, i.e., language $L\,(2_Y)$ is nullable (see (4.16))
- $Gui\,(1_X \dashrightarrow 0_Y)$ includes only the terminals a and c that label the shift arcs from state 0_Y (see (4.14)); it does not include terminal b since Y is not nullable.

Notice the two call arcs $0_S \dashrightarrow 0_X$ and $2_S \dashrightarrow 0_X$, and the call arc $1_Y \dashrightarrow 0_X$, although directed to the same machine M_X, have different guide sets, because they are associated with different call sites.

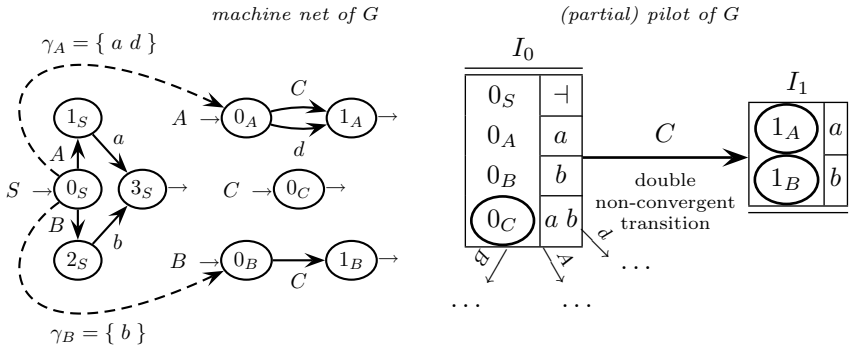
In this example we have used three relations (4.14)–(4.16) for computing the guide sets on call arcs. Relation (4.17) is not needed because in the *PCFG* no concatenated call arcs occur (that case has been already illustrated in the preceding example in Fig. 4.42). □

After thus obtaining the *PCFG* of a net, a short way remains to construct the deterministic predictive parser.

Observation on a Marginal Case

Going back to Property 4.66 on p. 300 (disjunction of guide sets), it must be said that statement 2 fails in some peculiar cases, which are however marginal in practical grammars. The contrived grammar below is an example of such exception. Notice that nonterminal C generates language $\{\,\varepsilon\,\}$. Despite the guide sets on the two call arcs in M_S (below, left) are disjoint, the grammar pilot violates the *STP*, and hence the condition *ELL* (1):

$$G \begin{cases} 1\colon\ S \to A\,a \mid B\,b \\ 2\colon\ A \to C \mid d \\ 3\colon\ B \to C \\ 4\colon\ C \to \varepsilon \end{cases}$$



This kind of exception can be removed by adding to statement 2 of Property 4.66 the hypothesis that the grammar does not have any nonterminal generating exclusively the empty string ε. Notice that such an hypothesis still allows to have nullable nonterminals, provided that each such nonterminal generates also non-empty strings.

In fact, grammar G has an equivalent form with disjoint guide sets: $S \to A\,a \mid b$ and $A \to \varepsilon \mid d$. It is structurally similar to G, but now neither one of nonterminals S and A generates exclusively string ε, thus the additional hypothesis is satisfied. Therefore, the pilot has the *STP* and the new grammar satisfies condition *ELL* (1) (p. 286). We finally observe that nonterminals generating only ε obviously play an irrelevant role in a grammar, and that they can always be eliminated as shown above; therefore such marginal cases have no practical impact.

4.7.3 Direct Construction of Top-Down Predictive Parsers

Section 4.7.2 has rigorously derived step by step the parser control-flow graph *PCFG* starting from the *ELR* (1) pilot graph of the given machine net, by assuming it complies with the *ELL* (1) condition on p. 286. As promised, we indicate a way to check the *ELL* (1) condition and then to derive the *PCFG* directly from the net, bypassing the calculation of the pilot.

From the Net to the Parser Control-Flow Graph

For the reader who has not gone through the formal steps of Sect. 4.7.2, we briefly introduce the notion of parser control-flow graph. The *PCFG*, denoted by $\mathcal{F}$, of a machine net $\mathcal{M}$, is a graph that has the same nodes and includes the same arcs as the original graph of the net; thus each node corresponds to a machine state. In $\mathcal{F}$ the following elements are also present, which do not occur in the net $\mathcal{M}$:

call arcs

An arc (depicted as dashed) links every node q that is the origin of a nonterminal shift $q \xrightarrow{B} r$, to the initial state of machine M_B. The label of a call arc is a set of terminals, named a *guide set*. Intuitively, a call arc represents a (possibly recursive) conditional invocation of the parsing procedure associated with machine M_B,

subjected to the condition that the current input token is in the guide set. Node r is named the *return state* of the invocation.

prospect sets
In $\mathcal{F}$ every final node q_A carries a prospect set as additional information. The set contains every token that can immediately follow the recognition of a string that derives from nonterminal A; its precise definition follows in Eq. (4.18), (4.19) on p. 306.

guide sets
Let us name *arrow* a terminal shift arc, a call arc, and also a dangling arrow (named *dart* or *exit arrow*) that exits a final state; a guide set is a set of terminals, to be associated with every arrow in $\mathcal{F}$:

- for a shift of terminal b, the guide set coincides with $\{ b \}$
- for a call arc $q_A \dashrightarrow 0_B$, the guide set indicates the tokens that are consistent with the invocation of machine M_B; the guide set γ of call arc $q_A \dashrightarrow 0_B$ is indicated by writing $q_A \overset{\gamma}{\dashrightarrow} 0_B$ or, equivalently, $\gamma = Gui(q_A \dashrightarrow 0_B)$; its exact computation follows in Eq. (4.20) on p. 307
- for the dart of a final state, the guide set equals the prospect set

Notice that nonterminal shift arcs have no guide set.
After the *PCFG* has been constructed, the *ELL* (1) condition can be checked using the following test (which rephrases Property 4.66 on p. 300).

Definition 4.69 (*direct ELL* (1) *test*) A machine network satisfies the *direct ELL* (1) *condition* if in the corresponding parser control-flow graph, for every pair of arrows originating from the same state q, the associated guide sets are disjoint. □

We list the rules for computing the prospect and guide sets.

Equations for Prospect Sets

We use a set of recursive equations to compute the prospect and guide sets for all the states and arcs of the *PCFG*; the equations are interpreted as instructions to iteratively compute the sets. To compute the prospect sets of the final states, we need to compute also the prospect sets of the other states, which are eventually discarded.

1. If the graph includes any nonterminal shift arc $q_i \xrightarrow{A} r_i$ (therefore also the call arc $q_i \dashrightarrow 0_A$), then the prospect set π_{0_A} for the initial state 0_A of machine M_A is computed as:

$$\pi_{0_A} := \pi_{0_A} \cup \bigcup_{q_i \xrightarrow{A} r_i} \Big(Ini\big(L(r_i)\big) \cup \textbf{if } Nullable\big(L(r_i)\big) \textbf{ then } \pi_{q_i} \textbf{ else } \emptyset \Big) \tag{4.18}$$

2. If the graph includes any terminal or nonterminal shift arc $p_i \xrightarrow{X_i} q$, then the prospect set π_q of state q is computed as:

$$\pi_q := \bigcup_{p_i \xrightarrow{X_i} q} \pi_{p_i} \tag{4.19}$$

The two sets of rules apply in an exclusive way to disjoints sets of nodes, because in a normalized machine no arc enters the initial state. To initialize the computation, we assign the end-marker to the prospect set of 0_S: $\pi_{0_S} := \{ \dashv \}$. All other sets are initialized to empty.

Equations for Guide Sets

The equations make use of prospect sets.

1. For each call arc $q_A \dashrightarrow 0_{A_1}$ associated with a nonterminal shift arc $q_A \xrightarrow{A_1} r_A$, such that possibly other call arcs $0_{A_1} \dashrightarrow 0_{B_i}$ depart from state 0_{A_1}, the guide set $Gui\,(q_A \dashrightarrow 0_{A_1})$ of the call arc is defined as follows, see also rules (4.14)–(4.17) on p. 299:

$$Gui\,(q_A \dashrightarrow 0_{A_1}) := \bigcup \begin{cases} Ini\,\big(L\,(A_1)\big) \\ \textbf{if } Nullable\,(A_1) \textbf{ then } Ini\,\big(L\,(r_A)\big) \\ \textbf{else } \emptyset \textbf{ endif} \\ \textbf{if } Nullable\,(A_1) \wedge Nullable\,\big(L\,(r_A)\big) \textbf{ then } \pi_{r_A} \\ \textbf{else } \emptyset \textbf{ endif} \\ \bigcup_{0_{A_1} \dashrightarrow 0_{B_i}} Gui\,(0_{A_1} \dashrightarrow 0_{B_i}) \end{cases} \tag{4.20}$$

2. For a final state $f_A \in F_A$, the guide set of the tagging dart equals the prospect set:

$$Gui\,(f_A \rightarrow) := \pi_{f_A} \tag{4.21}$$

3. For a terminal shift arc $q_A \xrightarrow{a} r_A$ with $a \in \Sigma$, the guide set is simply the shifted terminal:

$$Gui\,\left(q_A \xrightarrow{a} r_A\right) := \{ a \} \tag{4.22}$$

Initially all the guide sets are empty.

To compute the solution for prospect and guide sets, the above rules are repeatedly applied until none of the sets has changed in the last iteration; at that moment the set values give the solution.

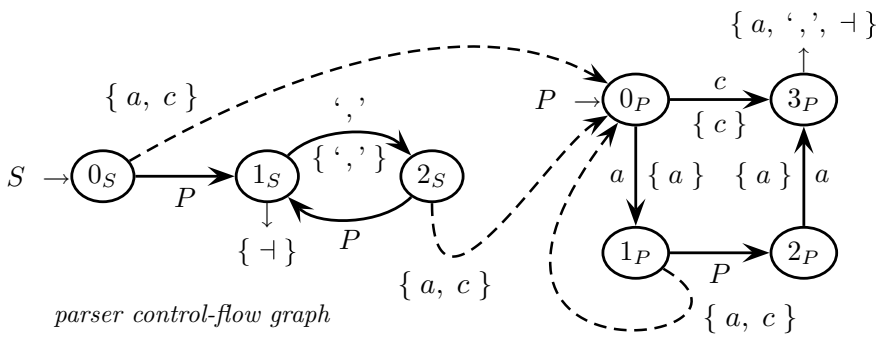


Fig. 4.46 Parser control-flow graph (*PCFG*) for Example 4.70 (formerly Example 4.53 on p. 283); the call arcs are dashed; the guide sets are in braces

Notice that the rules for computing prospect sets are consistent with the definition of look-ahead set given in Sect. 4.5.3; furthermore, the rules for computing the guide sets are consistent with the definition of parser control-flow graph listed in Definition 4.64 on p. 298.

Example 4.70 (*PCFG for list of balanced strings*) The net in Fig. 4.33 on p. 283 is now represented by the *PCFG* in Fig. 4.46. The following table shows the computation of the prospect sets for the *PCFG* in Fig. 4.46.

Node	*Init.*	*Equation*	*Arc instance*	*Prospect set*
0_S	$\{ \dashv \}$			$\{ \dashv \}$
1_S	$\emptyset$	(4.19)	$0_S \xrightarrow{P} 1_S$	$\{ \dashv \}$
2_S	$\emptyset$	(4.19)	$1_S \xrightarrow{‘,’} 2_S$	$\{ \dashv \}$
0_P	$\emptyset$	(4.18)	$0_S \xrightarrow{P} 1_S$	$\{ ‘, ’ \} \cup \{ \dashv \} \cup$
			$2_S \xrightarrow{P} 1_S$	$\{ ‘, ’ \} \cup \{ \dashv \} \cup$
			$1_P \xrightarrow{P} 2_P$	$\{ a \} \cup \emptyset = \{ a, ‘, ’, \dashv \}$
1_P	$\emptyset$	(4.19)	$0_P \xrightarrow{a} 1_P$	$\{ a, ‘, ’, \dashv \}$
2_P	$\emptyset$	(4.19)	$1_P \xrightarrow{P} 2_P$	$\{ a, ‘, ’, \dashv \}$
3_P	$\emptyset$	(4.19)	$2_P \xrightarrow{a} 3_P$	$\{ a, ‘, ’, \dashv \} \cup$
			$0_P \xrightarrow{c} 3_P$	$\{ a, ‘, ’, \dashv \} = \{ a, ‘, ’, \dashv \}$

The guide sets are computed by three equations, but the last two are trivial and we comment only Eq. (4.20) that applies to the three call arcs of the graph. The guide set of the call arc $0_S \dashrightarrow 0_P$ is simply $Ini\ ((0_P)) = \{ a, c \}$ since nonterminal P is not nullable and the call arc is not concatenated with other call arcs. Similarly, the guide sets of $2_S \dashrightarrow 0_P$ and $1_P \dashrightarrow 0_P$ have the same value.

Finally, it is immediate to check that the guide sets of bifurcating arrows are mutually disjoint, therefore the net is *ELL* (1). This confirms the earlier empirical analysis of the same net. □

The running example that follows illustrates the cases not covered and permits to compare the alternative approaches, based on the pilot graph or on the guide sets, for testing the *ELL* (1) condition.

Example 4.71 (*running example—computing the prospect and guide sets*) The following two tables show the computation of most prospect and guide sets for the *PCFG* of Fig. 4.42 (p. 301) here reproduced in Fig. 4.47. For both tables, the computation is completed at the third step.

prospect sets of states

0_E	1_E	0_T	1_T	2_T	3_T
$\dashv$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$)\ \dashv$	$)\ \dashv$	$a\ (\ )\ \dashv$	$a\ (\ )\ \dashv$	$a\ (\ )\ \dashv$	$a\ (\ )\ \dashv$
$)\ \dashv$	$)\ \dashv$	$a\ (\ )\ \dashv$	$a\ (\ )\ \dashv$	$a\ (\ )\ \dashv$	$a\ (\ )\ \dashv$

guide sets of call arcs

$0_E \dashrightarrow 0_T$	$1_E \dashrightarrow 0_T$	$1_T \dashrightarrow 0_E$
$\emptyset$	$\emptyset$	$\emptyset$
$a\ ($	$a\ ($	$a\ (\ )$
$a\ ($	$a\ ($	$a\ (\ )$

The disjointness test on guide sets confirms that the net is top-down deterministic. □

We examine the guide sets of the cascaded call arcs $1_T \overset{\gamma_1}{\dashrightarrow} 0_E \overset{\gamma_2}{\dashrightarrow} 0_T$, and we confirm the general property that the guide set upstream (γ_1) includes the one downstream (γ_2).

Violations of *ELL* (1) Condition

We know that the failure of the *ELL* (1) condition for a net that is *ELR* (1) can be due to two causes: the presence of left-recursive derivations, and the violation of the single-transition property in the pilot graph. First, we argue that left recursion implies non-disjointness of guide sets. Consider in the *PCFG* the bifurcation shown below:

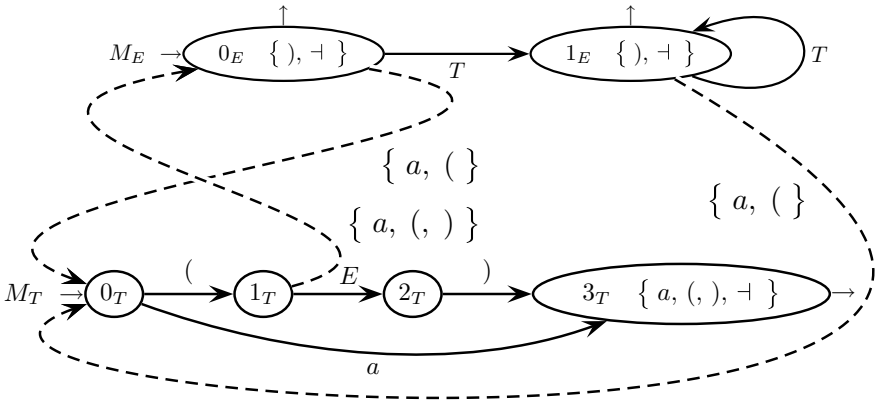
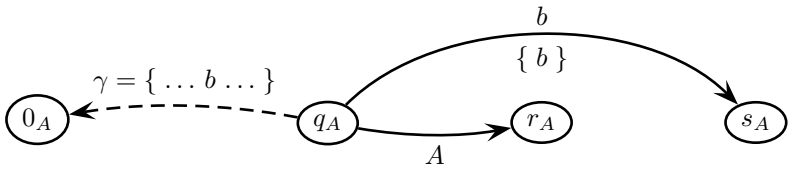


Fig. 4.47 The parser control-flow graph (*PCFG*) of the running example



By Eq. (4.20) the guide set γ of the call arc $q_A \dashrightarrow 0_A$ includes set $Ini\big(L(A)\big)$, which includes terminal b, hence it overlaps with the guide set of the terminal shift under b.

Second, the next example shows the consequence of *STP* violations.

Example 4.72 (*guide sets in a non-ELL* (1) *grammar*) The grammar with rules $S \to a^* N$ and $N \to a\, N\, b \mid \varepsilon$ of Example 4.57 (p. 288) violates the singleton base property as shown in Fig. 4.38, hence it is not *ELL* (1). This can be verified also by computing the guide sets on the *PCFG*. We have: $Gui\,(0_S \dashrightarrow 0_N) \cap Gui\,(0_S \xrightarrow{a} 1_S) = \{\, a \,\} \neq \emptyset$ and $Gui\,(1_S \dashrightarrow 0_N) \cap Gui\,(1_S \xrightarrow{a} 1_S) = \{\, a \,\} \neq \emptyset$: the guide sets on the arcs departing from states 0_S and 1_S are not disjoint. □

4.7.3.1 Predictive Parser

Predictive parsers come in two styles, as deterministic pushdown automata (*DPDA*), and as recursive syntactic procedures. Both are straightforward to obtain from the *PCFG*.

For the automaton, the pushdown stack elements are the nodes of the *PCFG*: the top-of-stack element identifies the state of the active machine, while inner stack elements refer to return states of suspended machines, in the correct order of suspension. There are four sorts of moves. A *scan* move, associated with a terminal shift arc, reads the current token, cc, as the corresponding machine would do. A *call* move, associated with a call arc, checks the enabling predicate, saves on stack the return state and switches to the invoked machine without consuming token cc. A *return* move is triggered when the active machine enters a final state, the prospect set of which includes token cc: the active state is set to the return state of the most recently suspended machine. A *recognizing* move terminates parsing.

Algorithm 4.73 (*predictive recognizer as DPDA*$\mathcal{A}$)

- The stack elements are the states of *PCFG* $\mathcal{F}$. The stack is initialized with element $\langle\, 0_S \,\rangle$.
- Let $\langle\, q_A \,\rangle$ be the top-of-stack element, meaning that the active machine M_A is in state q_A. Different move types are next specified:

scan move
: if the shift arc $q_A \xrightarrow{cc} r_A$ exists, then scan the next token and replace the stack top by $\langle\, r_A \,\rangle$ (the active machine does not change)

call move
: if there exists a call arc $q_A \xrightarrow{\gamma} 0_B$ such that $cc \in \gamma$, let $q_A \xrightarrow{B} r_A$ be the corresponding nonterminal shift arc; then pop, push element $\langle\, r_A \,\rangle$ and push element $\langle\, 0_B \,\rangle$

return move
: if q_A is a final state and token cc is in the prospect set associated with q_A, then pop

recognition move
: if M_A is the axiom machine, q_A is a final state and $cc = \dashv$, then accept and halt

– in any other case, reject the string and halt □

It should be clear that, since the guide sets at bifurcating arrows are disjoint (Property 4.66 or Definition 4.69), in every parsing configuration at most one move is possible, i.e., the algorithm is deterministic.

Computing Leftmost Derivations

To construct the syntax tree, the recognizer is extended with an output function, thus turning the *DPDA* into a so-called pushdown *transducer* (a model studied in Chap. 5). The algorithm produces the sequence of grammar rules of the leftmost derivation of the input string, but since the grammar G is in *EBNF* form, we have

Table 4.7 Derivation steps computed by a predictive parser; the current token is $cc = b$

Parser move	RL rule used
Scan move for transition $q_A \xrightarrow{b} r_A$	$q_A \underset{\hat{G}}{\Longrightarrow} b\, r_A$
Call move for call arc $q_A \xrightarrow{\gamma} 0_B$ and transition $q_A \xrightarrow{0_B} r_A$	$q_A \underset{\hat{G}}{\Longrightarrow} 0_B\, r_A$
Return move from state $q_A \in F_A$	$q_A \underset{\hat{G}}{\Longrightarrow} \varepsilon$

to use the rules of the equivalent *right-linearized* (*RL*) *grammar* $\hat{G}$, as anticipated in Sect. 4.5.2 on p. 239. The reason why we do not use *EBNF* derivations is that a derivation step may consist of an unbounded number of moves of the parser. Using the *RL* grammar, such a large derivation step is expanded into a series of derivation steps that match the parser moves. We recall that the syntax tree for grammar $\hat{G}$ is essentially an encoding of the syntax tree for the *EBNF* grammar G, such that each node has at most two children.

For each type of parser move, Table 4.7 reports the output, i.e., the rule of $\hat{G}$, which is printed. The correspondence is so direct that no justification is needed and an example suffices.

Example 4.74 (*running example—trace of predictive parser DPDA*) The *RL* grammar (from Example 4.27 on p. 240) is the following:

$$
\begin{array}{ll}
0_E \to 0_T\ 1_E \ \mid\ \varepsilon & 1_E \to 0_T\ 1_E \ \mid\ \varepsilon \\
0_T \to a\ 3_T \ \mid\ \text{`(`}\ 1_T & 1_T \to 0_E\ 2_T \\
2_T \to \text{`)`}\ 3_T & 3_T \to \varepsilon
\end{array}
$$

For the input string $x = (\,a\,)$, we simulate the computation including (from left to right) the stack content and the remaining input string, the test (predicate) on the guide set, and the rule used in the left derivation. Guide and prospect sets are those of the *PCFG* of Fig. 4.47 on p. 310.
For the original *EBNF* grammar, the corresponding derivation is:

$$E \Rightarrow T \Rightarrow (\,E\,) \Rightarrow (\,T\,) \Rightarrow (\,a\,) \qquad \square$$

stack	*x*	*predicate*	*left derivation*
$\langle 0_E \rangle$	$(\ a\ )\ \dashv$	$(\in \gamma = \{\ a\ (\ \}$	$0_E \Rightarrow 0_T\ 1_E$
$\langle 1_E \rangle\ \langle 0_T \rangle$	$(\ a\ )\ \dashv$	scan	$0_E \stackrel{+}{\Rightarrow} (\ 1_T\ 1_E$
$\langle 1_E \rangle\ \langle 1_T \rangle$	$a\ )\ \dashv$	$a \in \gamma = \{\ a\ (\ \}$	$0_E \stackrel{+}{\Rightarrow} (\ 0_E\ 2_T\ 1_E$
$\langle 1_E \rangle\ \langle 2_T \rangle\ \langle 0_E \rangle$	$a\ )\ \dashv$	$a \in \gamma = \{\ a\ (\ \}$	$0_E \stackrel{+}{\Rightarrow} (\ 0_T\ 1_E\ 2_T\ 1_E$
$\langle 1_E \rangle\ \langle 2_T \rangle\ \langle 1_E \rangle\ \langle 0_T \rangle$	$a\ )\ \dashv$	scan	$0_E \stackrel{+}{\Rightarrow} (\ a\ 3_T\ 1_E\ 2_T\ 1_E$
$\langle 1_E \rangle\ \langle 2_T \rangle\ \langle 1_E \rangle\ \langle 3_T \rangle$	$)\ \dashv$	$) \in \pi = \{\ a\ (\ )\ \dashv\}$	$0_E \stackrel{+}{\Rightarrow} (\ a\ \varepsilon\ 1_E\ 2_T\ 1_E$
$\langle 1_E \rangle\ \langle 2_T \rangle\ \langle 1_E \rangle$	$)\ \dashv$	$) \in \pi = \{\ )\ \dashv\}$	$0_E \stackrel{+}{\Rightarrow} (\ a\ \varepsilon\ 2_T\ 1_E$
$\langle 1_E \rangle\ \langle 2_T \rangle$	$)\ \dashv$	scan	$0_E \stackrel{+}{\Rightarrow} (\ a\ )\ 3_T\ 1_E$
$\langle 1_E \rangle\ \langle 3_T \rangle$	$\dashv$	$\dashv \in \pi = \{\ a\ (\ )\ \dashv\}$	$0_E \stackrel{+}{\Rightarrow} (\ a\ )\ \varepsilon\ 1_E$
$\langle 1_E \rangle$	$\dashv$	$\dashv \in \pi = \{\ )\ \dashv\}$ accept	$0_E \stackrel{+}{\Rightarrow} (\ a\ )\ \varepsilon$

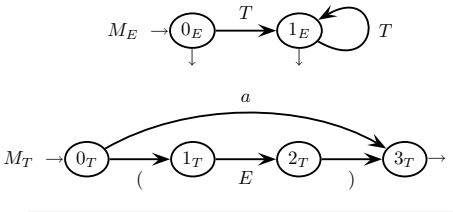
Parser Implementation by Recursive Procedures

We have seen in the introduction to predictive parsers on p. 283 that they are often implemented using recursive procedures. Each machine is transformed into a parameter-less *syntactic procedure*, having a control-flow graph matching the corresponding *PCFG* subgraph, so that the current state of a machine is encoded at run-time by the value of the program counter. Parsing starts in the axiom procedure and successfully terminates when the input has been exhausted, unless an error has occurred before. The standard run-time mechanisms of procedure invocation and return from procedure, respectively, implement the call and return moves. An example should suffice to show how the procedure pseudo-code is obtained from the *PCFG*.

Example 4.75 (*recursive descent parser*) From the *PCFG* of Fig. 4.47 on p. 310, we obtain the syntactic procedures shown in Fig. 4.48. The pseudo-code can be optimized in several ways, which we leave out. □

Any practical parser has to cope with input errors, to provide a reasonable error diagnosis, and to be able to resume parsing after an error. A few hints on error management in parsing are in Sect. 4.12.

machine network



recursive descent parser

```
program ELL_PARSER
        cc = next
        call E
        if cc ∈ { ⊣ } then accept
        else reject end if
end program

procedure E
        - - optimized
        while cc ∈ { a ( } do
                call T
        end while
        if cc ∈ { ) ⊣ } then
                return
        else
                error
        end if
end procedure
```

recursive descent parser

```
procedure T
        - - state 0_T
        if cc ∈ { a } then
                cc = next
        else if cc ∈ { ( } then
                cc = next
                - - state 1_T
                if cc ∈ { a ( ) } then
                        call E
                else
                        error
                end if
                - - state 2_T
                if cc ∈ { ) } then
                        cc = next
                else
                        error
                end if
        else
                error
        end if
        - - state 3_T
        if cc ∈ { a ( ) ⊣ } then
                return
        else
                error
        end if
end procedure
```

Fig. 4.48 Main program and syntactic procedures of a recursive descent parser (Example 4.75 and Fig. 4.47); function *next* is the programming interface to the lexical analyzer or scanner; function *error* is the messaging interface

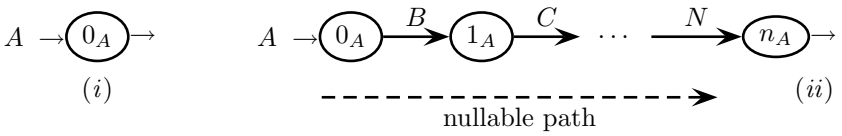
4.7.4 A Graphical Method for Computing Guide Sets

In this section we show how to compute by hand the guide sets by examining the net graph, for a machine net of small size. In this case, the present approach can profitably replace the preceding method based on the prospect set Eqs. (4.18) and (4.19), and on the guide set Eqs. (4.20)–(4.22). Since the equations require to know which nonterminals are nullable, and to have the set of the initials of the language accepted starting from a net state, we begin with these. Then we consider the prospect and guide sets, and we exemplify the rules on complete nets. Finally, we show how

this approach may also provide additional insight into the rules for computing the *ELR* pilot look-ahead.

Nullable Nonterminal

A nonterminal A or its machine M_A is nullable if (i) the initial state 0_A of M_A is final or if (ii) in M_A there is a path that connects state 0_A to a final state n_A ($n \neq 0$) and the path label consists only of nullable nonterminals B, C, …, N, all of which differ from A. See the two cases on this machine:



All the states on the path may have other outgoing arcs. If the grammar is *BNF*, an alternative is to use the procedure for identifying the nullable nonterminals (on p. 66).

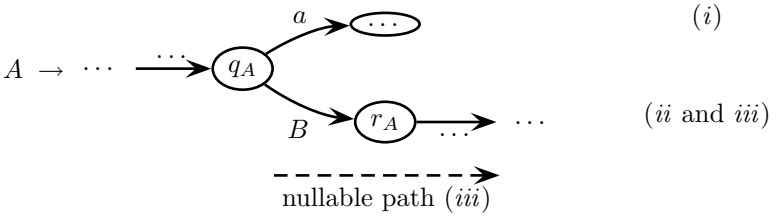
Set of Initials

In what follows, we abbreviate *Ini* (L (q_A)) in *Ini* (q_A) and *Ini* (L (B)) in *Ini* (B), as done in Definition 4.30, and we remind that *Ini* (B) = *Ini* (0_B). The case $B = A$ is permitted, unless differently stated.

The set *Ini* (q_A) of the initial terminals of the language of a state q_A of a machine M_A includes (see Definition 4.30):

(i) any terminal a that labels an arc from state q_A
(ii) the set *Ini* (B) of the initials of any nonterminal B that labels an arc from q_A, i.e., $q_A \xrightarrow{B} r_A$, excluding the case $0_A \xrightarrow{A} r_A$
(iii) if nonterminal B is *nullable*, also the set *Ini* (r_A) of the initials of the language of the destination state r_A; this implies to recursively go on adding initials as far as a nullable nonterminal path extends beyond B

See the three cases on this partial net graph:



$$Ini(q_A) = \{\ a\ \} \cup Ini(B) \quad \cup \underbrace{Ini(r_A)}_{\substack{\text{if nonterminal} \\ B \text{ is nullable}}}$$

A set of initials may be empty, this is the case for the set of the initials of the language of a final state without outgoing arcs.[25]

A pitfall to be avoided is to involve a prospect set (see below). For instance, if state q_A is final, this *is not* a reason for adding the prospect set π_A to set $Ini\,(q_A)$. In fact, the characters in the prospect set π_A are not generated by machine M_A, but by the machines that invoke M_A.

Prospect Set

For a nonterminal A or its machine M_A, all the final states have the same prospect set π_A, conventionally written on the exit arrows. Thus, it suffices to compute set π_A only once. Consider all the machines M_X (including the case $X = A$) that have arcs labeled by A, i.e., $q_X \xrightarrow{A} r_X$, and repeat these steps for each of such arcs, i.e., call sites, and machines (see (4.18) and (4.19)):

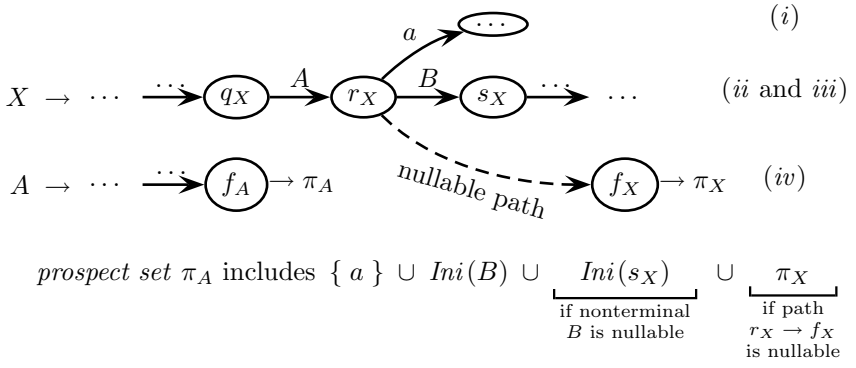
(*i*) any terminal a that labels an arc from state r_X

(*ii*) the set $Ini\,(B)$ of the initials of any nonterminal B that labels an arc from r_X, i.e., $r_X \xrightarrow{B} s_X$

(*iii*) if nonterminal B is nullable, also the set $Ini\,(s_X)$ of the initials of the language of state s_X; this implies to recursively go on adding initials as far as a nullable nonterminal path extends beyond B

(*iv*) if from state r_X there is a nullable nonterminal path to a final state f_X, also the prospect set π_X of M_X, which has to be already known

See the four cases on this partial net graph:



A particular instance of case (*iv*) is when state r_X itself is final. The prospect set π_A includes all the contributions collected as above for each arc $q_A \xrightarrow{A} r_X$. In addition,

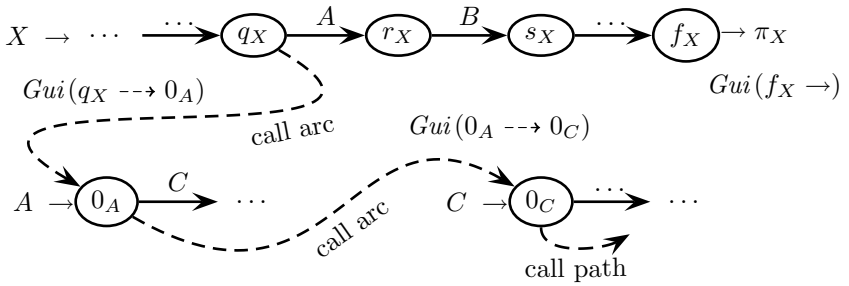
[25] More generally, we have an empty initial set for the language of a state connected to a final state without outgoing arcs, when the connection is exclusively through one (or more) nonterminal path(s) and every nonterminal on such path(s) can generate only the empty string ε. Indeed, this is a pathological case of no practical interest.

the axiomatic prospect set π_S always includes the terminator $\dashv$, which may propagate to other prospect sets. For this reason, a prospect set may never be empty, as at least it contains only the terminator $\dashv$.

Another alternative graphical approach for computing the prospect set π_A of machine M_A, if the pilot is available, is to collect from its graph all the look-ahead terminals of all the final states f_A of M_A.

Guide Set

Each terminal arc, exit arrow and call arc of a machine M_X has a non-empty guide set, conventionally written on the arc or arrow. The guide set on a terminal arc contains only the arc label. The guide set on an exit arrow of a final state f_X is the prospect set of M_X, i.e., $Gui\,(f_X \rightarrow) = \pi_X$. Thus, we only need the rules for computing the guide set $Gui\,(q_X \dashrightarrow 0_A)$ on a call arc of M_X, associated with a nonterminal arc $q_X \xrightarrow{A} r_X$ invoking machine M_A:



We distinguish four cases, which correspond to those listed in (4.20):

(*i*) put the initials $Ini\,(A)$ into $Gui\,(q_X \dashrightarrow 0_A)$

(*ii*) if nonterminal A *is nullable*, put also the initials $Ini\,(r_X)$ into $Gui\,(q_X \dashrightarrow 0_A)$; if nonterminal B is *nullable as well*, put also the initials $Ini\,(s_X)$; this implies to recursively go on adding initials as far as a *nullable path* extends beyond B

(*iii*) if nonterminal A *is nullable* and there is a *nullable path* B, …from state r_X to a *final* state f_X, put also the prospect set π_X of M_X into $Gui\,(q_X \dashrightarrow 0_A)$; this comprises the case that state r_X itself is final

(*iv*) if the call arc $q_X \dashrightarrow 0_A$ is *followed* by a call arc $0_A \dashrightarrow 0_C$, put also the guide set $Gui\,(0_A \dashrightarrow 0_C)$, which has to be already known, into $Gui\,(q_X \dashrightarrow 0_A)$; and so on similarly for all the call arcs that may follow

In case (*iv*), a pitfall to be avoided is to put the guide set of an exit arrow into a guide set on a call arc path; their name similarity may mislead. For instance, if state 0_A is final, this *is not* a reason for adding the guide set $Gui\,(0_A \rightarrow) = \pi_A$ to $Gui\,(q_X \dashrightarrow 0_A)$. In fact, set π_A contains all the terminals expected at the end of a computation of M_A started from *anywhere* in the net, not only from $q_X \xrightarrow{A} r_X$. Instead, in the case (*iii*), the prospect set to correctly add is π_X of the invoking machine M_X.

4.7.4.1 Examples of Graphical Solution

We conclude with four examples. First, we resume Example 4.70 and 4.71 (running example), already iteratively solved, and we show the graphical approach. Then, we solve three more examples, only graphically. We omit the (trivial) guide sets on terminal arcs and pre-compute the initials only if necessary.

Example 4.76 (*graphical solution of Example 4.70 and 4.71*) See Fig. 4.46 for the *PCFG* of Example 4.70:

nullability
: (i) 0_P is not final and (ii) it does not have any nullable path to a final state, thus P is non-nullable; 0_S is similar, since its outgoing P-arc is non-nullable, thus S is non-nullable as well

prospect sets
: S is not invoked anywhere, thus its prospect reduces to $\pi_S = \{ \dashv \}$; P is invoked on $0_S \xrightarrow{P} 1_S, 2_S \xrightarrow{P} 1_S$ and $1_P \xrightarrow{P} 2_P$, thus its prospect receives (i) terminals "," and a from 1_S and 2_P, respectively, and (iv) prospect π_S, since 1_S is final, thus it results $\pi_P = \{ a, \text{`,'}, \dashv \}$

guide sets
: that of call $0_S \dashrightarrow 0_P$ receives (i) only the initials a and c of P, since P is non-nullable, and similarly for calls $2_S \dashrightarrow 0_P$ and $1_P \dashrightarrow 0_P$, thus it results $\{ a, c \}$ for the three calls

See Fig. 4.47 for the *PCFG* of Example 4.71:

nullability
: (i) 0_E is final, thus E is nullable; (i) 0_T is non-nullable and (ii) does not have any nullable path to a final state, thus T is non-nullable

initials
: those of T are (i) only terminals a and "(" from 0_T, since 0_T does not have any nullable outgoing arc or path; those of E are (ii) only the initials a and "(" of T, since 0_E has a non-nullable outgoing T-arc

prospect sets
: E (axiom) is invoked on $1_T \xrightarrow{E} 2_T$, and its prospect receives $\dashv$ and (i) only the initial ")" of 2_T, since 2_T does not have any nullable outgoing arc or path, thus it results $\pi_E = \{ \text{`)'}, \dashv \}$; $T$ is invoked on both $0_E \xrightarrow{T} 1_E$ and $1_E \xrightarrow{T} 1_E$, and its prospect receives ($ii$) the initials $a$ and "(" of $T$ itself, since $1_E$ has an outgoing $T$-arc, and ($iv$) the prospect $\pi_E$, since $1_E$ is final, thus it results $\pi_T = \{ a, \text{`('}, \text{`)'}, \dashv \}$

guide sets
: that of call $0_E \dashrightarrow 0_T$ receives (i) only the initials a and "(" of T, since T is non-nullable, and similarly for call $1_E \dashrightarrow 0_T$, thus it results $\{ a, \text{`('} \}$; that of call $1_T \dashrightarrow 0_E$ receives (i) the initials a and "(" of E, (ii) the initial ")" of 2_T,

since E is nullable, and (*iv*) the guide set of the following call $0_E \dashrightarrow 0_T$, already computed, thus it results { a, '(', ')' } □

The next three examples focus on the details (and pitfalls) left uncovered by Example 4.76. None of them turns out to be deterministic *ELL* (1).

Example 4.77 (*graphical approach to guide sets—I*) See the *PCFG* in Fig. 4.49. Axiom S is non-nullable. Nonterminal B is nullable since state 0_B is final. Axiom S is invoked on arc $0_B \xrightarrow{S} 1_B$ and the initials of state 1_B are terminal b, thus the prospect set of S is { b, $\dashv$ }. Nonterminal B is invoked on arcs $1_S \xrightarrow{B} 2_S$ and $2_B \xrightarrow{B} 1_B$, and the initials of states 2_S and 1_B are terminals c and b, respectively, thus the prospect set of B is { b, c }.

The initials of axiom S, i.e., of state 0_S, are terminal a. The initials of nonterminal B, i.e., of state 0_B, are (recursively) those of S, i.e., a, and nothing else as S is non-nullable. The initials of B do not include the prospect set of B, despite (pitfall) state 0_B is final, as terminals c and b are not generated by the invoked machine M_B as its first symbols, i.e., starting from 0_B. Instead, they are, respectively, generated by the invoking machines M_S and (recursively) M_B, the latter yet starting from state 1_B (not from 0_B).

The guide set on the call $0_B \dashrightarrow 0_S$ associated with $0_B \xrightarrow{S} 1_B$ is (*i*) the initials a of S since S is non-nullable, thus it is { a }. The guide set on the call $1_S \dashrightarrow 0_B$ associated with $1_S \xrightarrow{B} 2_S$ includes (*i*) the initials a of B, (*ii*) the initials c of 2_S since B is nullable, and (*iv*) the guide { a } on the concatenated call $0_B \dashrightarrow 0_S$ (to the same effect as (*i*)), thus it is { a, c }. Similarly, the guide set on the call $2_B \dashrightarrow 0_B$ associated with $2_B \xrightarrow{B} 1_B$ includes (*i*) again the initials a of B, (*ii*) the initials b of 1_B since B is nullable, and (*iv*) again the guide { a } on call $0_B \dashrightarrow 0_S$, thus it is { a, b }.

The guide set on call $1_S \xrightarrow{a,\,c} 0_B$ does not include terminal b, despite (pitfall) b occurs on exit $0_B \xrightarrow{b,\,c}$, as such an occurrence is due to the other call $2_B \xrightarrow{a,\,b} 0_B$, and it includes terminal c because B is nullable and c follows the call to B (not because c is on the exit from 0_B). Similarly, call $2_B \xrightarrow{a,\,b} 0_B$ does not include c in its guide set, and it includes b instead. □

Example 4.78 (*graphical approach to guide sets—II*) See the *PCFG* in Fig. 4.50. Nonterminals A and B are nullable. Axiom S is non-nullable. The initials of A are a (pitfall—not b and c), those of B are b (pitfall—not c and $\dashv$), and those of S are a, *Ini* (A), i.e., again a, and (*iii*) *Ini* (1_S), i.e., c, since A is nullable, thus it is a and c.

The prospect set of S is just { $\dashv$ }, as S is not invoked anywhere. The prospect set of A includes b and c, since A is invoked on $1_A \xrightarrow{A} 2_A$ followed by b and on $0_S \xrightarrow{A} 1_S$ followed by c, thus it is { b, c }. The prospect set of B includes c and $\dashv$,

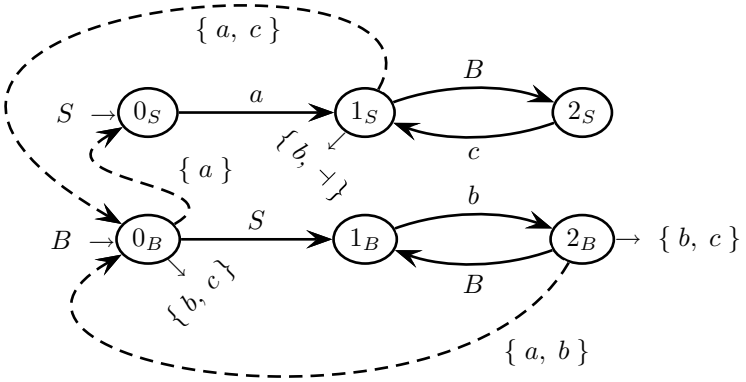


Fig. 4.49 *PCFG* for Example 4.77—state 2_B has a conflict on terminal b

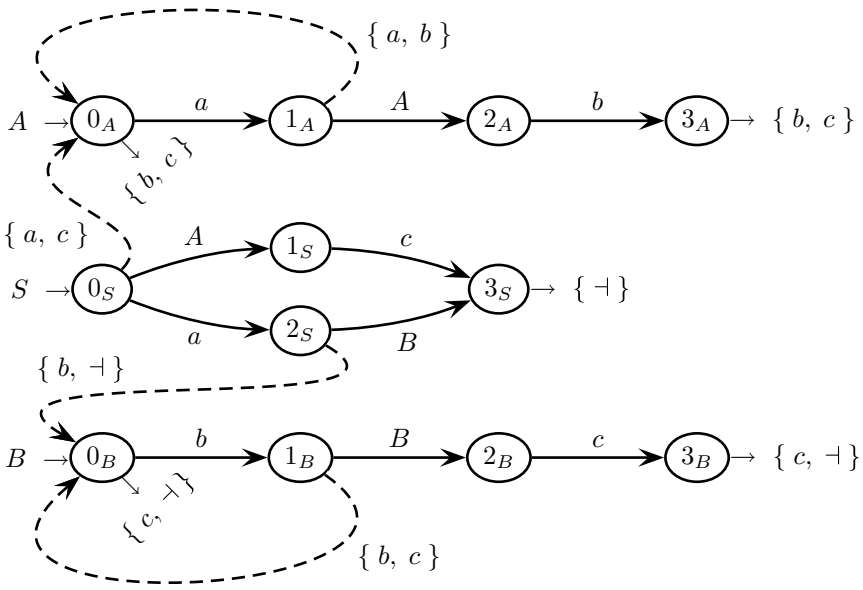


Fig. 4.50 *PCFG* for Example 4.78—state 0_S has a conflict on terminal a

since B is invoked on $1_B \xrightarrow{B} 2_B$ followed by c and (*iv*) on $2_S \xrightarrow{B} 3_S$, and state 3_S is final with prospect $\dashv$, thus it is $\{\ c,\ \dashv\ \}$.

The guide set on call $1_A \dashrightarrow 0_A$ includes *Ini* (A), i.e., a, and (*iii*) *Ini* (2_A), i.e., b, since A is nullable, thus it is $\{\ a,\ b\ \}$. The guide set on call $0_S \dashrightarrow 0_A$ includes a, *Ini* (A), i.e., again a, and (*iii*) *Ini* (1_S), i.e., c, since A is nullable, thus it is $\{\ a,\ c\ \}$. The guide set on call $2_S \dashrightarrow 0_B$ includes *Ini* (B), i.e., b, and (*iv*) the prospect set of S since state 3_S is final with prospect $\dashv$, thus it is $\{\ b,\ \dashv\ \}$. The guide set on call

$1_B \dashrightarrow 0_B$ includes *Ini* (B), i.e., b, and (*iii*) *Ini* (2_B), i.e., c, since B is nullable, thus it is $\{ b, c \}$.

Notice this subtle difference: symbol $\dashv$ is on call $2_S \xrightarrow{b,\dashv} 0_B$, since that call to B returns to state 3_S, final with $\dashv$ as prospect; instead, it is not on call $1_B \xrightarrow{b,c} 0_B$ (despite it is on the exit from 0_B), as state 2_B is not final and does not go to a final one through a nullable path. We see that other guide sets on calls do not necessarily include all the symbols on a chained exit, e.g., $1_A \xrightarrow{a,b} 0_A$ and $0_A \xrightarrow{b,c}$ with c not on call but on exit, for the reason that a guide set on a call is specific to that call site only (see also Example 4.77). □

Example 4.79 (*graphical approach to guide sets—III*) See the *PCFG* in Fig. 4.51. Nonterminal X is nullable. Nonterminal Y has a nullable path of length 1 to final state 1_Y, thus (*ii*) it is nullable. Axiom S is non-nullable. Most sets are easy to compute and here we do not detail all of them. The exit guide set of X includes b, since an X-call enters 1_Y (final) with prospect b, and includes c and e since X-calls are followed by such terminals (e is on a self-loop). The guide set on call $0_Y \dashrightarrow 0_X$ includes b since X is nullable and enters 1_Y (final) with prospect b, includes c since X is nullable and the call is followed by it, and includes d since it is the initial of X. This guide set propagates upstream to call $2_S \dashrightarrow 0_Y$.

Notice that terminal e is on call $1_X \xrightarrow{b,c,d,e} 0_X$, because X is nullable and after returning from that call the computation may find e (on self-loop $2_X \xrightarrow{e} 2_X$), not because it is also in the prospect of X. Furthermore, terminal e is on exit $0_X \xrightarrow{b,c,e}$,

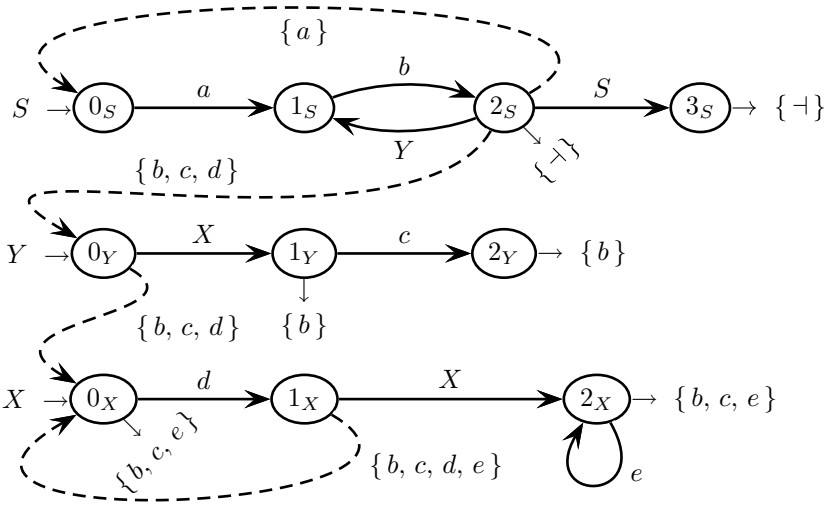


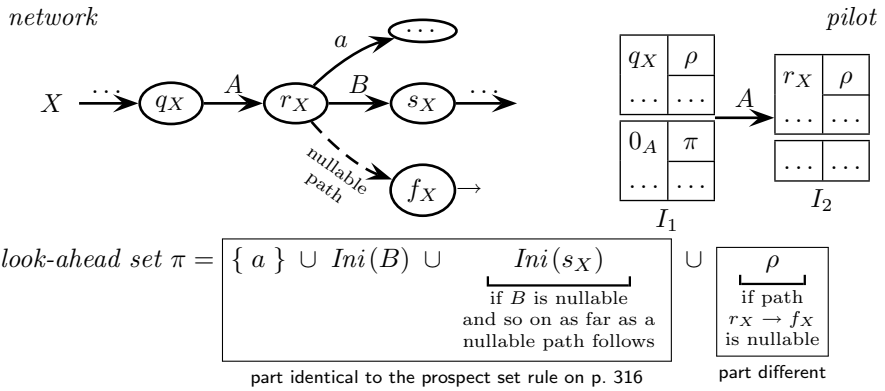
Fig. 4.51 *PCFG* for Example 4.79—state 2_X has a conflict on terminal e

yet it does not propagate upstream onto call path $2_S \xrightarrow{b,\,c,\,d} 0_Y \xrightarrow{b,\,c,\,d} 0_X$, as it is visibly unrelated to those calls. □

4.7.4.2 Relation Between Prospect Sets and Pilot Look-Ahead Sets

Returning to the look-ahead computation in the *ELR* pilot, already solved through the *closure* function (4.8) and Algorithm 4.35, we notice that a graphical approach reusing part of what said for the *PCFG* may help here as well:

Each nonterminal arc $q_X \xrightarrow{A} r_X$ in the net causes, in every pilot m-state I_1 that contains a candidate $\langle\, q_X,\ \rho \,\rangle$, the insertion by closure of a new initial candidate $\langle\, 0_A,\ \pi \,\rangle$ with a *look-ahead set* π computed by the rule above.



Notice the similarity with the rule that computes the contribution of $q_X \xrightarrow{A} r_X$ to the prospect set π_A of nonterminal A (on p. 316), but here the last term is ρ, i.e., the look-ahead inherited from the state q_X invoking A (not the prospect set π_X); the two rule parts are framed. However, the prospect set π_A receives all the look-ahead sets of all the final candidates of A.

A simple case: in Fig. 4.49, computing the closure of candidates $\langle\, 2_B,\ \ldots \,\rangle$ and $\langle\, 1_S,\ \ldots \,\rangle$, respectively, yields $\langle\, 0_B,\ b \,\rangle$ and $\langle\, 0_B,\ c \,\rangle$. Since nonterminal B has only these two call sites, the union of the two look-ahead singletons b and c yields the prospect set $\pi_B = \{\, b,\ c \,\}$.

4.7.5 Increasing Look-Ahead in Top-Down Parsers

A pragmatic approach for obtaining a deterministic top-down parser when the grammar does not comply with condition *ELL* (1) is to look-ahead of the current character and examine the following ones. This often suffices to make deterministic the choice at bifurcation points and has the advantage of not requiring annoying modifications of the original grammar.

Algorithm 4.73 on p. 311 (or its recursive descent version) has to be slightly modified, in order to examine in a bifurcation state the input characters located at

$k > 1$ positions ahead of the current one, before deciding the move. If such a test succeeds in reducing the choices to one, we say the state satisfies condition *ELL* (k).

A grammar has the *ELL* (k) property if there exists an integer $k \geq 1$ such that, for every net machine and for every state, at most one choice among the outgoing arrows is compatible with the characters that may occur ahead of the current character, at a distance less than or equal to k.

For brevity, we prefer not to formalize the definition of the guide set of order k, since it is a quite natural extension of the basic case, and we directly proceed to exemplify the computation of a few guide sets of length $k = 2$.

Example 4.80 (*conflict between instruction labels and variable names*) A small fragment of a programming language includes lists of instructions (variable assignments, *for* statements, etc.), with or without a label. Both labels and variable names are *identifiers*. The *EBNF* grammar of this language fragment, just sketched for the example, is as follows (axiom progr)[26]:

$$
\begin{cases}
\langle \text{progr} \rangle \rightarrow [\ \langle \text{label} \rangle\ \text{':'}\ ]\ \langle \text{stat} \rangle\ (\ \text{';'}\ \langle \text{stat} \rangle\ )^* \\
\langle \text{stat} \rangle \rightarrow \langle \text{assign_stat} \rangle \mid \langle \text{for_stat} \rangle \mid \ldots \\
\langle \text{assign_stat} \rangle \rightarrow \text{id}\ \text{'='}\ \langle \text{expr} \rangle \\
\langle \text{for_stat} \rangle \rightarrow \text{for id}\ \ldots \\
\langle \text{label} \rangle \rightarrow \text{id} \\
\langle \text{expr} \rangle \rightarrow \ldots
\end{cases}
$$

The net machines are depicted in Fig. 4.52 with the relevant guide sets. To avoid clogging, only two call arcs are represented with their guide sets of order $k = 1, 2$, the latter framed. The other guide sets are on the shift arcs and final darts exiting the furcation nodes. The states are simply enumerated, and the hooks around nonterminal names are omitted. Clearly, state 0 is not *ELL* (1): the guide sets of length $k = 1$ (not framed) on the two call arcs that leave state 0, sharing the terminal id, are not disjoint. By refining the analysis, we observe that if the identifier is an instruction label, i.e., arc $0 \xrightarrow{\text{label}} 1$, then it is followed by a colon ":" character. On the other hand, in the case of arc $0 \xrightarrow{\text{stat}} 3$, the identifier is the left variable of an assignment statement and is followed by an equal "=" character.[27] We have thus ascertained that inspecting the second next character suffices to deterministically resolve the choice of the move from state 0, which therefore satisfies condition *ELL* (2). In the *PCFG*, the call arcs leaving state 0 are labeled with the guide sets of length $k = 2$ (framed), which are disjoint. Such pre-computed sets will be used at run-time by the parser. On the other hand, in the bifurcation states 3 and 4, a look-ahead length $k = 1$ suffices because

[26] The symbols within brackets ⟨ ⟩ are nonterminals, all the others are terminals or are assumed to be terminals; and the square brackets mean their contents are optional.

[27] In the case of the for statement, for simplicity we assume that the lexeme for is a reserved keyword, which may not be used as a variable or label identifier.

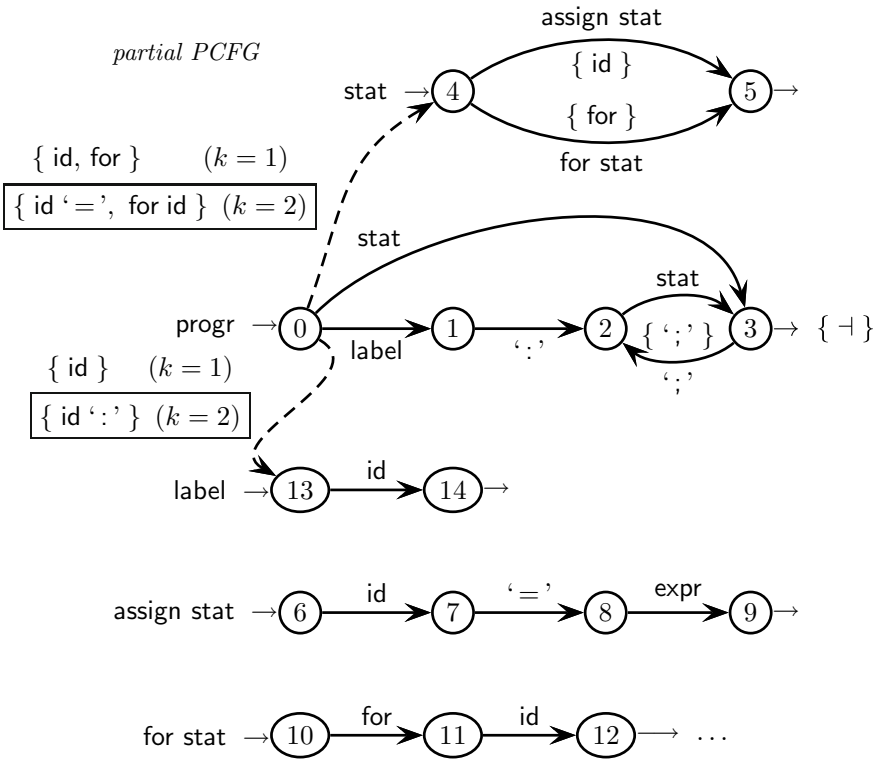


Fig. 4.52 Machine net and the relevant parts of the parser control-flow graph of Example 4.80. With look-ahead length $k = 2$ the graph is deterministic

the guide sets on the outgoing shift arcs and on the final dart are disjoint. It would be wasteful to use the maximal look-ahead length everywhere in the parser. □

In practice, parsers use different look-ahead lengths with an obvious economy criterion: in each bifurcation state, the minimum length $m \leq k$ needed to arbitrate the choice of the branches should be used.

Some top-down parsers that are more powerful, though computationally less efficient, have also been designed. A well-known example is the *ANTLR* parser [28], which uses a look-ahead of variable and unbounded length, depending on the current parser state. Other parsers, if the choice in a bifurcation state remains uncertain, take other cues into consideration. One possibility is to compute some semantic condition, an idea to be worked out in the last chapter. Of course, all such parsers can no longer be considered as deterministic pushdown automata, because they use other information pieces in addition to the stack contents and, in the case of unlimited look-ahead, they are able to perform multiple scans on the input.

4.8 Operator-Precedence Grammars and Parallel Parsing

In many application areas, algorithms that originally had been designed for sequential execution later have been transformed into parallel algorithms, to take advantage of the available multiple processors on current computers for reducing running time. For language parsing, a straightforward approach is to split the source text into as many *segments* (substrings) as the number of available processors, to parse each segment on a different processor by using a sequential parser, and then to recombine the resulting partial parse trees into the complete tree. In principle the speed-up obtained with respect to the original sequential algorithm scales with the number of processors, but there are conceptual and practical obstacles on this way that limit scalability.

First and foremost, neither the *LR* (1) nor the *LL* (1) parsing algorithms can be easily transformed into parallel programs because they need to scan the text sequentially from left to right and their decisions, whether to shift or to reduce, may depend on distant past decisions. Therefore, the parser of, say, the second segment may be unable to proceed (deterministically) until the parser of segment one has finished, which thus defeats the benefit of parallelization. To illustrate, observe the deterministic (*LR* (1) and also *LL* (1)) language below:

$$L = \overbrace{\left\{ c^* \, o \, c^* \, a^n \, b^n \mid \quad n \geq 1 \right\}}^{L_1} \cup \overbrace{\left\{ c^* \, t \, c^* \, a^n \, b^{2n} \mid \quad n \geq 1 \right\}}^{L_2}$$

such that the presence of letter o (respectively t) announces that each later letter a will be followed by *one* letter b (respectively *two* letters b). Imagine to parse the segmented text shown below:

$$\overbrace{c\,c\,\ldots\,c\,t\,c\,c\,\ldots\,c}^{\text{segment 1}} \; \overbrace{a\,a\,\ldots\,a\,b\,b\,\ldots\,b}^{\text{segment 2}}$$

Since the second parser does not know whether the first one has encountered a letter o or t, it faces the task of analyzing language $\{ a^n \, b^n \mid \quad n \geq 1 \} \cup \left\{ a^n \, b^{2n} \mid \quad n \geq 1 \right\}$, which is nondeterministic (as explained on p. 216 for a similar case). Notice that an arbitrary distance separates the positions of substrings o (or t) and $a \, b \, b$, where the choice of the reduction $a \, b \rightsquigarrow \ldots$ or $a \, b \, b \rightsquigarrow \ldots$ is due. This example shows that, if our goal is to deterministically parse separate text segments, we must exclude any deterministic language that exhibits such a long-distance dependence between operations performed by independent parsers. In other words, to be suitable for parallel parsing, a grammar, besides being *LR* (1), should not require parsing information to be carried to another substring over a long stretch of text. We loosely call *local parsability* such a grammar property.

We describe a very efficient sequential parsing algorithm that bases its parsing decisions on the local properties of the text, thereby it lends itself to parallel execution.

4.8.1 Floyd Operator-Precedence Grammars and Parsers

Floyd [29,30], a pioneer of compiler techniques, took inspiration from the traditional notion of precedence between arithmetic operators, in order to define a class of languages such that the shape of the parse tree is solely determined by a binary relation between terminals that are consecutive, or that become consecutive after a few bottom-up reduction steps. The Floyd approach assumes that the grammar has been normalized in the *operator form* (see Sect. 2.5.13.7 on p. 74); we recall that a rule is in the operator form if any two nonterminals that occur in the right part of the rule are separated by at least one terminal character. Many practical grammars are already in the operator form, e.g., the grammar of the Dyck language: $S \to b\, S\, e\, S \mid b\, S\, e \mid b\, e$. Otherwise, we have seen in Chap. 2 how to construct an equivalent grammar in the operator form.

All the grammars in this section are purely *BNF* in the operator form and free from empty rules.

Precedence Relations

We introduce the concept of *precedence relation*: this term actually means three partial binary relations over the terminal alphabet, named:

$\lessdot$	*yields precedence*
$\gtrdot$	*takes precedence*
$\doteq$	*equal in precedence*

For a given grammar and any pair of terminal characters $a,\ b \in \Sigma$, the precedence relation may take one or more of the above three *values*, or may be undefined. But for the operator-precedence grammars to be soon defined, for every character pair a, b, the relation, if it is defined, is unique.

To grasp the intended meaning of the relations, it helps to start from the language of the arithmetic expressions with the operators plus "$+$" (addition) and times "$\times$" (multiplication). Since traditionally times take precedence over plus, the following operator-precedence relations hold: $\times \gtrdot +$ and $+ \lessdot \times$. The former relation means that in an expression such as $5 \times 3 + 7$, the first reduction to apply is $5 \times 3 \rightsquigarrow \cdots$. Similarly, the fact that plus yields precedence to times, says that in an expression such as $5 + 3 \times 7$, the first reduction to apply is $3 \times 7 \rightsquigarrow \cdots$.

We are going to extend this traditional sense of precedence from the arithmetic operators to any terminal characters. Before that, we have to explain the meaning of the "equal in precedence" relation $a \doteq b$. It means that a grammar rule contains a substring $a\, A\, b$ where A is a nonterminal or is null, i.e., between a and b there is not another terminal. Thus in the grammar for parenthesized arithmetic expressions, the left and the right parentheses are in the relation "$($" $\doteq$ "$)$" due to the rule $E \to (\, F\,)$.

A word of warning: none of the three relations enjoys the mathematical properties of reflexivity, symmetry and transitivity, of the numerical order relations $<$, $>$ and $=$, notwithstanding the typographical similarity.

We observe that for an operator grammar, every string β that derives from a nonterminal has the property that any two nonterminals are separated by one or more terminal characters. More precisely we have:

$$\text{if } A \overset{*}{\Rightarrow} \beta \text{ then } \beta \in \Sigma^* \left(V_N \, \Sigma^+ \right)^* \cup \Sigma^* \left(V_N \, \Sigma^+ \right)^* V_N$$

We also say that such strings β are in the *operator form*. More explicitly, a string in the operator form takes one of the forms below:

$$a \in \Sigma \quad \text{or} \quad A \in V \quad \text{or} \quad B \, \sigma \, C \tag{4.23}$$

where B and C are nonterminals possibly null, and the string σ is in operator form and starts and ends with a terminal character.

To construct a precedence parser for a given grammar, we have to compute the precedence relations and to check that they do not conflict in a sense to be explained. We need to define two sets of terminal characters, which are similar to the sets of initial and final characters of a string, but differ because the nonterminal symbols are considered to be always nullable.

Definition 4.81 (*left/right terminal sets*) The *left* and *right terminal sets* of a string in the operator form, respectively, denoted $\mathcal{L}$ and $\mathcal{R}$, are recursively defined by:

$$\begin{aligned} \mathcal{L}(a) &= \{ a \} & \mathcal{R}(a) &= \{ a \} \quad \text{for } a \in \Sigma \\ \mathcal{L}(A) &= \bigcup_{\forall \text{ rule } A \to \alpha} \mathcal{L}(\alpha) & \mathcal{R}(A) &= \bigcup_{\forall \text{ rule } A \to \alpha} \mathcal{R}(\alpha) \\ \mathcal{L}(B \, \sigma \, C) &= Ini(\sigma) \cup \mathcal{L}(B) & \mathcal{R}(B \, \sigma \, C) &= Fin(\sigma) \cup \mathcal{R}(C) \end{aligned} \tag{4.24}$$

where B, C and σ are as in (4.23) above, and *Ini* and *Fin* denote the initial and final characters of a string. □

Example 4.82 (*operator-precedence grammar of arithmetic expressions*) A grammar G for certain arithmetic expressions is shown in Fig. 4.53, left. The left/right terminal sets of the strings that occur in the grammar as left- or right-hand sides, computed by Eq. (4.24), are listed in Table 4.8. □

To compute the precedence relations we need the following definition.

Definition 4.83 (*precedence relations*) Given an operator grammar, let B be a nonterminal symbol, and let a and b be terminal characters. The grammar operator-precedence (*OP*) relations over $\Sigma \times \Sigma$ are:

grammar G

$$S \to A \mid B$$
$$A \to A + B \mid B + B$$
$$B \to B \times n \mid n$$

operator precedence matrix M

	n	$+$	$\times$
n		$\gtrdot$	$\gtrdot$
$+$	$\lessdot$	$\gtrdot$	$\lessdot$
$\times$	$\doteq$		

Fig. 4.53 Operator-precedence grammar for certain arithmetic expressions and its operator-precedence matrix (read a matrix entry as *row rel. col*, e.g., $n \gtrdot +$, etc)

Table 4.8 Some left and right terminal sets for the grammar in Fig. 4.53

String	Left terminal set $\mathcal{L}$	Right terminal set $\mathcal{R}$
n	n	n
$B \times n$	$\{ n, \times \}$	n
B	$\{ n, \times \}$	n
$B + B$	$\{ n, \times, + \}$	$\{ n, + \}$
$A + B$	$\{ n, \times, + \}$	$\{ n, + \}$
A	$\{ n, \times, + \}$	$\{ n, + \}$
S	$\{ n, \times, + \}$	$\{ n, + \}$

takes precedence	$a \gtrdot b$ iff some rule contains a substring $B\, b$ and $a \in \mathcal{R}(B)$
yields precedence	$a \lessdot b$ iff some rule contains a substring $a\, B$ and $b \in \mathcal{L}(B)$
equal in precedence	$a \doteq b$ iff some rule contains a substring $a\, B\, b$ or $a\, b$

The *operator-precedence matrix* (*OPM*), M, of a grammar is a square array of size $|\Sigma|$ that associates to each ordered pair (a, b) the set $M_{a,b}$ of *OP* relations holding between the terminals a and b.

A grammar has the *operator-precedence property* (we also say it is an operator-precedence grammar) if, and only if, each matrix entry contains at most one relation. In that case, the matrix qualifies as *conflict-free*. □

We refer the reader to Fig. 4.54 for a schematic illustration of the meaning of precedence relations.

As said, the precedence relations are not necessarily symmetric, in the sense that relation $a \gtrdot b$ does not imply that relation $b \lessdot a$ holds; similarly relation $a \doteq b$ does not imply that relation $b \doteq a$ holds.

For the current Example 4.82, we compute the *OPM* in Fig. 4.53, right, by using the left/right terminal sets listed in Table 4.8, and by inspecting the substrings that

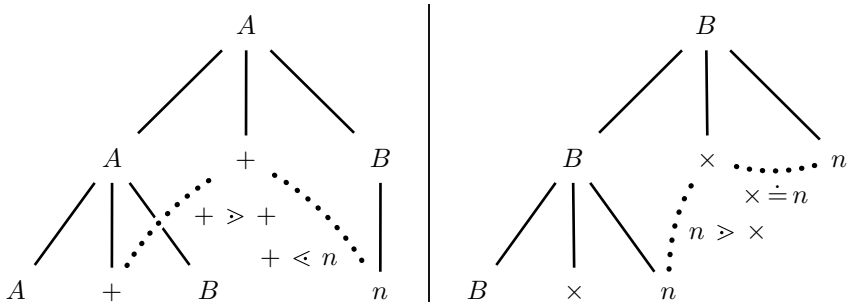


Fig. 4.54 Precedence relations made apparent in the derivations of Example 4.82

occur in the following rules:

rule	*terminal set*	*precedence relations*
$\cdots \to A\ +\ \ldots$	$\mathcal{R}(A) = \{\, n,\ + \,\}$	$n \gtrdot +$ and $+ \gtrdot +$
$\cdots \to B\ +\ \ldots$	$\mathcal{R}(B) = \{\, n \,\}$	$n \gtrdot +$
$\cdots \to \ldots\ +\ B$	$\mathcal{L}(B) = \{\, n,\ \times \}$	$+ \lessdot n$ and $+ \lessdot \times$
$\cdots \to \ldots\ \times\ n$		$\times \doteq n$

The grammar has the *OP* property since its *OPM* is conflict-free.

We comment the meaning of a few relations. The *OPM* entries $+ \lessdot n$ and $n \gtrdot +$ indicate that, when parsing a string containing the pattern $\ldots + n + \ldots$, the right part n of rule $B \to n$ has to be reduced to nonterminal B. The *OPM* entry $(\, n,\ n\,)$ is empty because in a valid string, terminal n is never adjacent to another n, or separated by a nonterminal as in the patterns $\ldots\ n\ A\ n\ \ldots$ or $\ldots\ n\ B\ n\ \ldots$; in other words no sentential form may contain such patterns. Similarly, there is no relation between the operators $\times$ and $+$, because none of the patterns $\ldots \times + \ldots$, $\ldots \times A + \ldots$ and $\ldots \times B + \ldots$ ever occurs in a sentential form. We observe that relation $\doteq$ is connected with an important grammar parameter, namely the length of the rule right-hand sides. Clearly a rule $A \to A_1\, a_1\ \ldots\ A_t\, a_t\, A_{t+1}$, where each symbol A_i is a possibly missing nonterminal, is associated with the relations $a_1 \doteq a_2 \doteq \cdots \doteq a_t$.

The next example (Example 4.84) refers to a real language, which turns out to have a syntax ideal for precedence parsing.

Example 4.84 (*grammar and OP matrix of JSON*) The *JavaScript Object Notation* (*JSON* language) has been introduced in Chap. 2 in Example 2.43 on p. 48, where its official grammar is listed. We reproduce in Fig. 4.55 the grammar with a minor change needed for the operator form; in the same figure we show the operator-precedence matrix, which is conflict-free. The very large size of the typical data files represented in *JSON*, and the fact that the official grammar is essentially in the operator form and without precedence conflicts, make parallel parsing attractive. Notice that in the *OPM* a majority of cases is void. This is typical of practical grammars, whose *OPM* matrix is frequently quite sparse. □

$$
\begin{aligned}
\text{S} &\to \text{OBJECT}\\
\text{OBJECT} &\to \{\ \} \mid \{\ \text{MEMBERS}\ \}\\
\text{MEMBERS} &\to \text{PAIR} \mid \text{PAIR}\ ,\ \text{MEMBERS}\\
\text{PAIR} &\to \text{STRING}\ :\ \text{VALUE}\\
\text{VALUE} &\to \text{STRING} \mid \text{OBJECT} \mid \text{ARRAY} \mid \boldsymbol{num} \mid \boldsymbol{bool}\\
\text{STRING} &\to ''\,'' \mid ''\ \text{CHARS}\ ''\\
\text{ARRAY} &\to [\ ] \mid [\ \text{ELEMENTS}\ ]\\
\text{ELEMENTS} &\to \text{VALUE} \mid \text{VALUE}\ ,\ \text{ELEMENTS}\\
\text{CHARS} &\to \underline{\boldsymbol{char} \mid \boldsymbol{char}\ \text{CHARS}}
\end{aligned}
$$

operator precedence matrix

	{	}	,	:	num	bool	"	char	[	]
{		$\doteq$	$\lessdot$	$\lessdot$			$\lessdot$			
}		$\gtrdot$	$\gtrdot$							$\gtrdot$
,	$\lessdot$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$		$\lessdot$	$\gtrdot$
:	$\lessdot$	$\gtrdot$	$\gtrdot$		$\lessdot$	$\lessdot$	$\lessdot$		$\lessdot$	
num		$\gtrdot$	$\gtrdot$							$\gtrdot$
bool		$\gtrdot$	$\gtrdot$							$\gtrdot$
"		$\gtrdot$	$\gtrdot$	$\gtrdot$			$\doteq$	$\lessdot$		$\gtrdot$
char							$\gtrdot$	$\gtrdot$		
[	$\lessdot$		$\lessdot$		$\lessdot$	$\lessdot$	$\lessdot$		$\lessdot$	$\doteq$
]		$\gtrdot$	$\gtrdot$							$\gtrdot$

Fig. 4.55 Operator grammar and *OP* matrix of *JSON*. Uppercase and lowercase words denote nonterminals and terminals, respectively. To cast the grammar into operator form, only the underlined rule has been changed with respect to the official *JSON* grammar (listed in Example 2.43 on p. 48)

4.8.2 Sequential Operator-Precedence Parser

Precedence relations precisely control whether a substring that matches the right part of a rule (also referred to as the *handle*) has to be reduced to the nonterminal on the rule left part. This control is very efficient and, unlike the conditions examined by *LR* (1) parsers, it does not rely on state information. A minor difficulty remains however, if the grammar includes two rules such as $A \to x$ and $B \to x$ with the same right part: how to choose between nonterminals A and B when string x is reduced. A possibility is to represent such uncertainty as a set of nonterminals, i.e., $\{ A, B \}$, and to propagate it across the following reductions, until just one choice remains

(otherwise the grammar would be ambiguous). But, to simplify matters, we assume that the grammar is free from repeated right parts, i.e., it is *invertible*, a property that can be always satisfied by the transformation presented in Chap. 2 on p. 69.

Operator-precedence grammars are an ideal choice for parallel parsing. For sequential parsing, they are occasionally adopted, in spite of their limitations with respect to *ELR* (1) grammars, because the parsers are simpler to construct and equally efficient. As the parallel algorithm stems directly from the sequential one, we start from the latter.

The key idea driving the algorithm is simple. The algorithm scans the input, and wherever a series of $\doteq$ precedence relations enclosed by the relation pair $\lessdot$, $\gtrdot$ is found between consecutive terminals, the enclosed substring is a reduction handle. To find handles, the parser uses the *OPM* and stores the current input and the precedence relations on a pushdown stack $\mathcal{S}$, until the next $\gtrdot$ relation occurs. Then the handle is the portion of stack encompassed by the most recent $\lessdot$ and the top.

For convenience, the input string is enclosed by the special character #, acting as end-markers. This is tantamount to adding the new axiom S_0 and the axiomatic rule $S_0 \to \# \; S \; \#$. Therefore, character # yields precedence to every character in the left terminal set of S and every character in the right terminal set of S takes precedence over #.

To prepare for its parallel use, we enable the parsing algorithm to analyze also the strings that may contain nonterminal characters. Such strings begin and end with terminal characters (or with character #) and are in the operator form. This setting is only needed when the parallel algorithm is called to parse internal segments of the input text.

Each stack symbol contained in $\mathcal{S}$ is a pair of the form $(\, x, \; p \,)$, where $x \in \Sigma \cup V$ and p is one of the precedence relations $\{ \lessdot, \doteq, \gtrdot \}$ or is undefined (denoted by $\perp$). Component p is used to encode the precedence relation existing between two consecutive symbols; conventionally, we set $p = \perp$ for nonterminals. In the following representations, we assume that the stack grows rightward.[28]

Algorithm 4.85 (*operator-precedence sequential parser*)

OPparser ($\mathcal{S}$, u, *head*, *end*)

$\mathcal{S}$: stack of couples u: string to analyze *head*, *end*: pointers in u

conventions and Settings

denote $u = u_1 \, u_2 \, \ldots \, u_m$ and $s = u_2 \, \ldots \, u_{m-1}$ with $m > 2$

denote $u_1 = a$ and $u_m = b$ where a and b are terminal

for sequential use only:

denote the initial stack contents as $\mathcal{S} = (a, \perp)$

set the initial pointers *head* := 2 and *end* := m

[28] See [31] for further reading on this algorithm, and [18] for the simpler version, which is however unfit for parallel execution.

begin

1. let $x = u_{head}$ be the symbol currently pointed by *head* and consider the precedence relation between x and the topmost terminal y in the stack $\mathcal{S}$
2. **if** x is a nonterminal **then**
   push $(x, \perp)$ $head++$
3. **else if** $y \lessdot x$ **then**
   push $(x, \lessdot)$ $head++$
4. **else if** $y \doteq x$ **then**
   push $(x, \doteq)$ $head++$
5. **else if** $y \gtrdot x$ **then**

   a. **if** stack $\mathcal{S}$ does not contain $\lessdot$ **then**
      push $(x, \gtrdot)$ $head++$
   b. **else**
      let the stack $\mathcal{S}$ be $(z_0, p_0) \ldots (z_{i-1}, p_{i-1})(z_i, \lessdot) \ldots (z_n, p_n)$ where z_i (more precisely p_i) is the topmost $\lessdot$ relation, i.e., for every stack element on the right of z_i the precedence relation is $\doteq$ or $\perp$

      i. **if** z_{i-1} is a nonterminal **and** $\exists$ rule $A \to z_{i-1}\ z_i\ \ldots\ z_n$ **then**
         replace $(z_{i-1}, p_{i-1})(z_i, \lessdot) \ldots (z_n, p_n)$ in $\mathcal{S}$ with $(A, \perp)$
      ii. **else if** z_{i-1} is a terminal or # **and** $\exists$ rule $A \to z_i\ \ldots\ z_n$ **then**
         replace $(z_i, \lessdot) \ldots (z_n, p_n)$ in $\mathcal{S}$ with $(A, \perp)$
      iii. **else**
         start an error recovery procedure[29]
      end if

   end if

end if

6. **if** $head < end$ **or** $\big(head = end$ **and** $\mathcal{S} \neq (a, \perp)(B, \perp)\big)$ for any nonterminal B **then goto** step (1) **else return** $\mathcal{S}$ **end if**

end □

[29] An error is detected either when the handle, i.e., the string included between relations $\lessdot$ and $\gtrdot$, does not match any right-hand side, or when no precedence relation holds between two consecutive terminal characters.

stack	*input string*	*relation or reduction*		*step*
$(\#, \bot)$	$\overset{head}{\#}\ \overset{}{n}\ + n \times n\ \overset{end}{\#}$	$\# \lessdot n$		1
$(\#, \bot)(n, \lessdot)$	$\overset{head}{+}\ n \times n \#$	$n \gtrdot +$	$n \rightsquigarrow B$	2
$(\#, \bot)(B, \bot)$	$\overset{head}{+}\ n \times n \#$	$\# \lessdot +$		3
$(\#, \bot)(B, \bot)(+, \lessdot)$	$\overset{head}{n}\ \times n \#$	$+ \lessdot n$		4
$(\#, \bot)(B, \bot)(+, \lessdot)(n, \lessdot)$	$\overset{head}{\times}\ n \#$	$n \gtrdot \times$	$n \rightsquigarrow B$	5
$(\#, \bot)(B, \bot)(+, \lessdot)(B, \bot)$	$\overset{head}{\times}\ n \#$	$+ \lessdot \times$		6
$(\#, \bot)(B, \bot)(+, \lessdot)(B, \bot)(\times, \lessdot)$	$\overset{head}{n}\ \#$	$\times \doteq n$		7
$(\#, \bot)(B, \bot)(+, \lessdot)(B, \bot)(\times, \lessdot)(n, \doteq)$	$\overset{head}{\#}$	$n \gtrdot \#$	$B \times n \rightsquigarrow B$	8
$(\#, \bot)(B, \bot)(+, \lessdot)(B, \bot)$	$\overset{head}{\#}$	$+ \gtrdot \#$	$B + B \rightsquigarrow A$	9
$(\#, \bot)(A, \bot)$	$\overset{head\ end}{\#}$			10

Fig. 4.56 Steps of the operator-precedence sequential parser for the input $a + a \times a$ (grammar and *OPM* in Fig. 4.53)

If no error occurs, the algorithm terminates in a configuration that cannot be changed either by shift or by reduce actions, and we say that it is *irreducible*.

When used sequentially and not as a part of the parallel version to be next presented, the algorithm is called with parameters $u = \# s \#$ and $\mathcal{S} = (\#, \bot)$ (so $a = \#$), it never performs step 2, never pushes $(x, \gtrdot)$ onto the stack, and accepts the input string only if the stack is $\mathcal{S} = (\#, \bot)(S, \bot)$.

We comment a few steps of the parsing trace shown in Fig. 4.56. Step 1 shifts token n on the stack, then step 2 detects the pattern $\lessdot n \gtrdot$ and reduces the handle n to B. At step 3, the topmost terminal on stack is #, which yields precedence to the current token $+$ that is shifted. Skipping to step 7, the stack top $\times$ equals in precedence token n, which is therefore shifted. Step 8 identifies the handle $B \times n$ in the pattern $\lessdot B \times \doteq n \gtrdot$; notice that the handle includes a nonterminal B, which occurs between the terminal characters related by $+ \lessdot \times$.

4.8.3 Comparisons and Closure Properties

Let *OP* denote the family of languages defined by *OP* grammars. We start with the obvious remark that family *OP* is included into the family *DET*, since Algorithm 4.85 clearly implements a deterministic pushdown machine.

The following examples permit to compare the languages of *OP* and those of *LL* (1) and *LR* (1) grammars.

A language that is deterministic but is known not to be in *OP* is $\{\ a^n\ b\ a^n \mid\ \ n \geq 0\ \}$, generated by the grammar:

$$S \to a\ S\ a\ \mid\ b \qquad \text{with matrix} \qquad \begin{array}{c|cc} & a & b \\ \hline a & \lessdot\ \gtrdot\ \doteq & \lessdot \\ b & \gtrdot & \end{array}$$

There are conflicts in the matrix entry $[\,a,\ a\,]$, yet the grammar meets the *LL* (1) and therefore also the *LR* (1) condition.

On the other hand there exist *OP* languages that are not in the *LL* (1) family. An example is the language $\{\ a^*\ a^n\ b^n \mid\ \ n \geq 1\ \}$ (see Fig. 4.35 on p. 285). It is generated by the conflict-free *OP* rules $S \to a\ S\ \mid\ B$ and $B \to a\ B\ b\ \mid\ a\ b$.

Each *OP* grammar has the peculiar property of *symmetry with respect to mirror reversal*, in the sense that the grammar obtained by reversing the right parts of the rules is also an *OP* grammar. Its precedence matrix simply interchanges the "yield" and "take" relations of the original grammar, and reverses the "equal in precedence" relation. To illustrate we list another grammar G:

$$G \begin{cases} S \to X\ b \\ X \to a\ X\ b\ \mid\ a\ b \end{cases} \qquad \text{with } \mathit{OPM} \qquad \begin{array}{c|cc} & a & b \\ \hline a & \lessdot & \doteq \\ b & & \gtrdot \end{array}$$

The reversed language $(\ L\ (G)\ )^R$ is defined by the "reversed" *OP* grammar G_R:

$$G_R \begin{cases} S \to b\ X \\ X \to b\ X\ a\ \mid\ b\ a \end{cases} \qquad \text{with } \mathit{OPM} \qquad \begin{array}{c|cc} & a & b \\ \hline a & \gtrdot & \\ b & \doteq & \lessdot \end{array}$$

On the other hand, in general, neither an *LL* (1) nor an *LR* (1) grammar remains such when it is reversed. For instance, grammar G above has the *LL* (1) property, but the reversed grammar G_R does not, though it remains *LR* (1). The case of a language L that ceases to be deterministic upon reversal is illustrated by:

$$L = \{\ o\,a^n\,b^n \mid\ \ n \geq 1\ \} \cup \{\ t\,a^{2n}\,b^n \mid\ \ n \geq 1\ \}$$

Here the first character (o or t) determines whether the string contains one letter a per b or two letters a per b. The reversed language L^R contains sentences of two types, say, $b^4\,a^4\,o$ and $b^4\,a^8\,t$. A pushdown machine cannot deterministically decide when to pop a stack symbol standing for a letter b upon reading an a.

We collect the preceding findings into the next property, which also deals with the families of regular and of input-driven languages (Definition 4.20 on p. 225).

Property 4.86 (Comparison of OP and other deterministic families)

1. *Family* OP *is strictly included in the family* DET *of deterministic languages, i.e., the family of* LR (1) *languages, see Property 4.93 on p. 348.*
2. *The families of* OP *languages and of* LL (1) *languages are not comparable.*
3. *Family* OP *strictly includes the family of input-driven languages, therefore also the family of regular languages.*
4. *The reversal* L^R *of every* OP *language* L *has the* OP *property. More precisely, if* L *is generated by the grammar* G *then* L^R *is generated by the reversed grammar* G^R*. The* OPM *of* G^R *interchanges the "yield" and "take" relations of* G*, and reverses the "equal in precedence" relation.* □

Since we have already commented statements 1, 2 and 4, and the inclusion of regular languages within input-driven ones has been discussed in Sect. 4.3.4.2 on p. 227, it remains to discuss statement 3, i.e., that every language recognized by an input-driven *PDA* is also an *OP* language.

We start from the elementary case that R is a regular language over alphabet $\Sigma_{internal}$, and we assume, without any loss of generality, that it is defined by a strictly left-linear grammar G (Table 4.4 on p. 227), which clearly is in the operator form. Each rule of G has one of the forms (disregard for simplicity the end-markers #) $A \to B\,a$ and $A \to a$, where A and B are nonterminal symbols and a is terminal. By applying Definition 4.83, we easily discover that grammar G cannot have any "yield" or "equal" relation, thus leaving us with the "takes" precedence relation. It follows that the precedence matrix of any strictly left-linear grammar is conflict-free. Notice that the same property holds for a strictly right-linear grammar, except that the only possible relations are "yield" precedence.

After showing the (obviously strict) inclusion of the language families $REG \subset OP$, we move to the general case of an *ID* language L over the 3-partitioned alphabet $\Sigma_{call} \cup \Sigma_{return} \cup \Sigma_{internal}$. We are going to rely in part on the reader intuition[30] in the next presentation of the correspondence between the input-driven subalphabets of two terminal characters and the precedence relations that may occur in an equivalent *OP* grammar.

For each input-driven language L with alphabet $\Sigma_{call} \cup \Sigma_{return} \cup \Sigma_{internal}$ there exists an *OP* grammar that generates language L and has an *OPM* with the following structure:

	$c \in \Sigma_{call}$	$i \in \Sigma_{internal}$	$r \in \Sigma_{return}$
$c \in \Sigma_{call}$	$\lessdot$	$\lessdot$	$\doteq$
$i \in \Sigma_{internal}$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$r \in \Sigma_{return}$	$\gtrdot$	$\gtrdot$	$\gtrdot$

(4.25)

The *OPM* encodes the following constraints:

[30] We refer to [32] for a complete proof.

1. for each $c_1,\ c_2 \in \Sigma_{call}$, the only possible relation is $c_1 \lessdot c_2$
2. for each $c \in \Sigma_{call}$ and $i \in \Sigma_{internal}$, the only possible relation is $c \lessdot i$
3. for each $c \in \Sigma_{call}$ and $r \in \Sigma_{return}$, the only possible relation is $c \doteq r$
4. for each $i \in \Sigma_{internal}$ and $c \in \Sigma_{call}$, the only possible relation is $i \gtrdot c$
5. for each $i_1,\ i_2 \in \Sigma_{internal}$, the only possible relation is $i_1 \gtrdot i_2$
6. for each $i \in \Sigma_{internal}$ and $r \in \Sigma_{return}$, the only possible relation is $i \gtrdot r$
7. for each $r \in \Sigma_{return}$ and $c \in \Sigma_{call}$, the only possible relation is $r \gtrdot c$
8. for each $r \in \Sigma_{return}$ and $i \in \Sigma_{internal}$, the only possible relation is $r \gtrdot i$
9. for each $r_1,\ r_2 \in \Sigma_{return}$, the only possible relation is $r_1 \gtrdot r_2$

To understand the reason of such constraints, it helps to mentally equate the characters in Σ_{call} and Σ_{return} to open and closed parentheses, respectively. Therefore, the cases $c \lessdot c, c \doteq r$ and $r \gtrdot r$ should be clear by comparison with the structure of parentheses languages. Now consider the relation $i \gtrdot i$: it says that a string of internal characters has to be parsed using a left-linear grammar, as we just did to show the inclusion of regular languages into *OP*.

The other cases of matrix (4.25) can be justified by observing in Fig. 4.57 the parsing of a string over a 3-partitioned alphabet. Thus, relations $c \lessdot i$ and $i \gtrdot c$ say that a substring over $\Sigma_{internal}$, bordered on the left and on the right by c, must be handled as before for a regular language string bordered by #. For brevity, we do not discuss the remaining cases of matrix (4.25).

To end the proof of statement 3 of Property 4.86 we show an *OP* language that is not input-driven. The language is:

$$L_{XYZ} = \left\{ a^n\, b^n \mid \ n \geq 1 \right\} \cup \left\{ c^n\, d^n \mid \ n \geq 1 \right\} \cup \left\{ e^n\, (a\, c)^n \mid \ n \geq 1 \right\}$$

The straightforward operator-precedence grammar is in Example 4.88 on p. 338. On the other hand, the language cannot be input-driven for the following reasons. The strings of type $a^n\, b^n$ impose that a is a call and b a return; for similar reasons, c must be a call and d a return. But the strings of type $e^n\, (a\, c)^n$ impose that at least one of a and c must be a return, which is a contradiction for a 3-partitioned alphabet.

$$\begin{array}{l}
\bot \text{ - } \bot \\
\bot \lessdot \text{ - } c \gtrdot \text{ - } \bot \\
\bot \lessdot \text{ - } i \gtrdot c \lessdot \text{ - - - - - - - } c \gtrdot \text{ - - - - - - - - - - - - - - } \bot \\
\bot \lessdot \text{ - - - - - - - - - - - } c \doteq \text{ - - - - - - - } r \gtrdot i \gtrdot c \lessdot \text{ - - - - - - - } c \lessdot \text{ - - - - - - - - - - - } i \gtrdot \bot \\
\bot \lessdot \text{ - - - - - - - - - } r \gtrdot c \doteq \text{ - - - - - - - } r \gtrdot i \gtrdot c \lessdot \text{ - - - - - - - } c \lessdot c \doteq \text{ - - - - - - - } r \gtrdot i \gtrdot \bot \\
\bot \lessdot \text{ - - } c \doteq \text{ - - } r \gtrdot r \gtrdot c \lessdot \text{ - - - - } i \gtrdot r \gtrdot i \gtrdot c \lessdot c \doteq \text{ - - } r \gtrdot c \lessdot c \lessdot \text{ - - } c \doteq r \gtrdot r \gtrdot i \gtrdot \bot \\
\bot \lessdot i \gtrdot c \lessdot i \gtrdot r \gtrdot r \gtrdot c \lessdot c \doteq r \gtrdot i \gtrdot r \gtrdot i \gtrdot c \lessdot c \lessdot i \gtrdot r \gtrdot c \lessdot c \lessdot i \gtrdot c \doteq r \gtrdot r \gtrdot i \gtrdot \bot
\end{array}$$

Fig. 4.57 Parsing the string of an input-driven language using the *OP* matrix (4.25). The symbols of the form c, i and r, respectively, denote arbitrary call/internal/return characters

It would be possible to show that, if an *OP* grammar G has a partition of the terminal alphabet into three sets, i.e., $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \Sigma_3$, and its precedence matrix has the same structure as matrix (4.25) has, then the language $L(G)$ is input-driven. More precisely, its 3-partition is $\Sigma_{call} = \Sigma_1$, $\Sigma_{internal} = \Sigma_2$ and $\Sigma_{return} = \Sigma_3$.

To sum up, the family of input-driven languages coincides with the family of languages generated by operator-precedence grammars having an *OPM* of the 3-partitioned form (4.25).

4.8.3.1 Closure Properties of Compatible *OP* Languages

We say that two *OP* grammars G' and G'' are *compatible* if the union of their *OP* matrices, i.e., $OPM(G') \cup OPM(G'')$, is conflict-free. Notice that if the terminal alphabets of G' and G'' differ, the two *OP* matrices are normalized by considering the union of the two alphabets.

Within the family of all the *OP* grammars, let us fix a particular *OP* matrix M. Then we focus on the family of *OP* grammars and corresponding languages that are compatible with matrix M, to be denoted by $OP(M)$, by means of the following definition:

$$OP(M) = \{\, G \mid \ G \text{ is an } OP \text{ grammar and } OPM(G) \subseteq M\}$$

Clearly, every such family of grammars and corresponding languages is a subset of *OP*, with $OP(M) \subset OP$.

We observe that if the relation $\doteq$ in M is circular, i.e., if matrix M contains a chain $a_1 \doteq a_2 \doteq \ldots \doteq a_t \doteq a_1$ where $t \geq 1$, then the $OP(M)$ family includes some unusual grammars. Such grammars would have rules with right parts of unbounded length, which are strings or substrings contained in the regular language $(a_1\, a_2\, \ldots\, a_t)^+$. For both practical and mathematical reasons, we prefer to exclude such situation by assuming that matrix M does not contain any cyclic $\doteq$ chain.

For any matrix M, the corresponding language family has the remarkable property that the fundamental operations (union, intersection, set difference and concatenation) that, when they are applied to general *OP* languages, may produce a language that loses the *OP* property, now produce a language that remains in the same family $OP(M)$.

Property 4.87 (closure properties of compatible OP languages) *For each* OP *matrix* M *(assumed to be* $\doteq$*-acyclic), the family of compatible languages, i.e.,* $\mathrm{OP}(M)$*, is closed under the following operations: union, set difference and therefore intersection.*

Moreover, for any compatible languages $L' = L(G')$ *and* $L'' = L(G'')$*, the concatenation* $L = L(G') \cdot L(G'')$ *and the star* $\left(L(G')\right)^*$ *are generated by* OP *grammars, respectively,* $G_{(\cdot)}$ *and* $G_{(*)}$*, such that:*

$$\mathrm{OPM}\left(G_{(\cdot)}\right) \supseteq \mathrm{OPM}(G') \ \textit{ and } \ \mathrm{OPM}\left(G_{(*)}\right) \supseteq \mathrm{OPM}(G') \tag{4.26}$$

□

The proof of the above properties, including the constructions of the grammars, is in [32], and we just show an example.

Example 4.88 (*operations on compatible grammars*) The following languages L_X, L_Y and L_Z:

$$L_X = \{\, a^n\, b^n \mid \ n \geq 1 \,\} \qquad L_Y = \{\, c^n\, d^n \mid \ n \geq 1 \,\}$$

$$L_Z = \{\, e^n\, (a\, c)^n \mid \ n \geq 1 \,\}$$

are, respectively, defined by the *OP* grammars:

$$X \to a\, X\, b \mid a\, b \qquad Y \to c\, Y\, d \mid c\, d \qquad Z \to e\, Z\, a\, c \mid e\, a\, c$$

The three grammars are conflict-free and compatible as we see from the three individual *OP* matrices and from the resulting one M shown below:

M_X

	a	b
a	$\lessdot$	$\doteq$
b		$\gtrdot$

	c	d
c	$\lessdot$	$\doteq$
d		$\gtrdot$

	a	c	e
a		$\doteq$	
c	$\gtrdot$		
e	$\doteq$		$\lessdot$

$M =$

	a	b	c	d	e
a	$\lessdot$	$\doteq$	$\doteq$		
b		$\gtrdot$			
c	$\gtrdot$		$\lessdot$	$\doteq$	
d				$\gtrdot$	
e	$\doteq$				$\lessdot$

Therefore, if we apply the operations listed in Property 4.87 to the above three languages, the languages that we obtain are compatible with matrix M. We show some cases.

Language $L_X \cup L_Y \cup L_Z$ has the grammar $S \to X \mid Y \mid Z$, plus the preceding rules for the nonterminal symbols X, Y and Z. The matrix is M, shown above.

The concatenation $L_X\, L_X$ contains strings of type $a^m\, b^m\, a^n\, b^n$, with $m,\ n \geq 1$, where the character b can be followed by a. Since in the matrix M the case $(b,\ a)$ is void, we can assign any relation to it. We discard the choice $b \doteq a$ that would create the circularity $b \doteq a \doteq b$, and we arbitrarily choose $b \gtrdot a$ (if the case already contained a relation, we would have to stick to the same). In accordance with the chosen relation, the grammar and *OPM* for $L_X\, L_X$ are:

$$\begin{cases} S \ \to \ X\, a\, X\, b \mid X\, a\, b \\ X \ \to \ \text{as above} \end{cases} \qquad M_X \ \cup \ \{\, b \gtrdot a \,\}$$

The last operation we illustrate is $(L_X)^+$. This time we arbitrarily assign to the case $(b,\ a)$ the relation $b \lessdot a$, thus obtaining the grammar and matrix:

$$\begin{cases} S \ \to \ a\,X\,b\,S \ \mid \ a\,b\,S \\ X \ \to \ \text{as above} \end{cases} \qquad M_X \ \cup \ \{\, b \lessdot a \,\} \qquad \square$$

Returning to the input-driven languages, we may ask what effect the operations considered in Property 4.87 have on such languages, but we already know the answer from Table 4.4 on p. 227: the input-driven family is closed under the same operations. This is not surprising since we have just seen that an input-driven language is an *OP* language that has a particular form of precedence matrix, and by applying, say, the union operation to two input-driven languages with the same 3-partition, we obtain an *OP* language that has an *OPM* fitting with such a particular form.

4.8.4 Parallel Parsing Algorithm

Here we describe how to use the sequential parsing algorithm to efficiently parse large texts in parallel. More information on the algorithm, its implementation and some experimental results can be found in [31].

At the level of abstraction that is usually adopted in compiling, the input text consists of a stream of symbols (also known as lexemes or tokens), which are terminal elements of the grammar. As we know, the sequential parser invokes the lexical analyzer (scanner) to obtain the next input token. To accommodate multiple parsers that operate in parallel, such an organization, which relies on a scanning subroutine invoked by the parser, has to be slightly revised in order to avoid that a token, say *CYCLE27*, stretching across consecutive text segments assigned to different parsers, may be split into two tokens, say *CYCL* and *E27*. Since this problem is marginal for the parallel parsing method we are interested in, we assume that the input text has been entirely scanned into tokens before parallel parsing starts.

Then, our data-parallel algorithm splits the load among workers by dividing the text into $k \geq 1$ segments, where k is the number of physical processors. How the text is split is irrelevant, provided that the substring lengths are approximately equal to ensure load balancing. In contrast, other parallel parsers have been proposed that perform a language dependent split, and require the text to be segmented at specific positions where a token occurs (such as `begin`), which parts a large syntactic constituent of the language.

We apply Algorithm 4.85 to each segment, by using a separate stack per worker. Each worker returns a stack, which is guaranteed to be correct independently of the stacks of the other workers. In practice, the reductions operated by each parser make a *final* fragment of the parse tree. The local parsability property of *OP* grammars manifests itself in that each parser bases its decisions on a look-ahead/look-back of length one, to evaluate the precedence relations between consecutive terminal characters. Therefore, in the split text, we have to overlap by one token any two

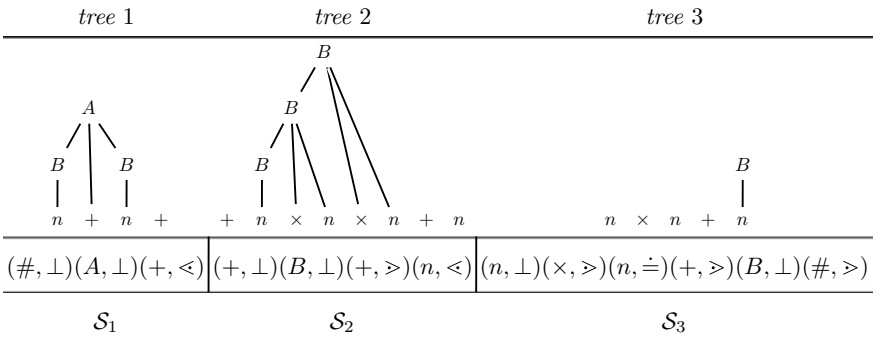


Fig. 4.58 Partial trees and corresponding stacks after the first parsing phase of text $n + n + n \times n \times n + n \times n + n$ by using three workers

consecutive segments, i.e., the last terminal of a segment coincides with the first terminal of the next segment.

Example 4.89 (*parallel parsing*) Reconsider the grammar of Example 4.82 on p. 327, and the source text below:

$$\# \; n \; + \; n \; + \; n \; \times \; n \; \times \; n \; + \; n \; \times \; n \; + \; n \; \#$$

Assume there are $k = 3$ workers and the segmented text is as below:

$$\overbrace{n \; + \; n}^{1} \; + \; \overbrace{n \; \times \; n \; \times \; n \; +}^{2} \; n \; \overbrace{\times \; n \; + \; n}^{3}$$

where the marks # are left implicit, and the symbols + and n not embraced are shared by the two adjacent segments.

After each sequential parser has processed its segment, the partial trees and the stacks are shown in Fig. 4.58. Notice that stack $\mathcal{S}_1$ contains the shortest possible string: a nonterminal symbol embraced by the two terminals that are always present on the stack ends. Such a stack is called *quasi-empty*. Stacks $\mathcal{S}_2$ and $\mathcal{S}_3$ are not quasi-empty since they have also terminal symbols in their inner part. □

Stack Combination

To complete parsing, we need an algorithm that joins the k existing stacks and launches one or more sequential parsers on the combined stack(s). Different strategies are possible:

serial
: One new stack is produced by joining all the stacks (and by taking care of canceling one of the duplicated symbols at each overlapping point). The next phase uses just one worker, i.e., it is sequential.

dichotomic
: A new stack is created by joining every pair of consecutive stacks at positions 1–2, 3–4, etc., so that the number of stacks of phase one is halved, as well as the number of workers.

opportunistic
: Two or more consecutive stacks are joined, depending on their length. In particular a quasi-empty stack is always joined to a neighboring stack. The number of stacks is guaranteed to decrease by one, or more rapidly if more stacks are quasi-empty.

If the output of phase two consists of more than one stack, then the last two strategies require further phases, which can employ the serial or again a parallel strategy. Of course, at the end all the strategies compute the same syntax tree and they only differ with respect to their computation speed.

The serial strategy is convenient if the stacks after phase one are short, so that it would be wasteful to parallelize phase two. We do not need to describe phase two since it can reuse the sequential parsing algorithm, which we had the foresight to design to process also nonterminal symbols. The dichotomic strategy fixes the number of workers of phase two without considering the remaining amount of work, which is approximately measured by the length of each combined stack; this may cause useless overhead when a worker is assigned a short or quasi-empty stack to analyze. The opportunistic approach, to be described, offers more flexibility in the combination of stacks and their assignment to the workers of phase two.

We recall that Algorithm 4.85 has two main parameters: stack $\mathcal{S}$ and input string u. Therefore, we describe how to initialize these two parameters for each worker of phase two (no matter the strategy). Each stack $\mathcal{S}$ returned by phase one, if it is not quasi-empty, is split into a *left* and a *right part*, respectively, denoted as $\mathcal{S}^L$ and $\mathcal{S}^R$, such that $\mathcal{S}^L$ does not contain any $\lessdot$ relation and $\mathcal{S}^R$ does not contain any $\gtrdot$ relation; of course, $\doteq$ relations may occur in both parts.

Thus, stack $\mathcal{S}_2$ is divided into $\mathcal{S}_2^L = (+, \bot)\,(B, \bot)\,(+, \gtrdot)$ and $\mathcal{S}_2^R = (n, \lessdot)$. If several cut points satisfy the condition, then we arbitrarily choose one of them. Notice that the sequential parser can maintain a pointer to the stack cut point, with a negligible overhead, so we can assume that the stack returned by Algorithm 4.85 is bipartite.

To initialize the two parameters (stack and input string) for a worker, the parallel algorithm combines two consecutive bipartite stacks, as next explained. Notice that the input string thus obtained contains terminal and nonterminal symbols (computed in the reductions of phase one). Let $\mathcal{W}$ and $\mathcal{W}'$ be consecutive workers of the previous phase, and let their bipartite stacks be $\mathcal{S}^L\ \mathcal{S}^R$ and $\mathcal{S}'^L\ \mathcal{S}'^R$, respectively. The initial configuration for a worker of the next phase is obtained as shown in Fig. 4.59. More precisely we define the following functions:

1. Stack initialization:

$$\mathcal{S}_{combine}\,(\mathcal{S}^L,\ \mathcal{S}^R) = (a, \bot)\ \mathcal{S}^R \qquad \text{where } a \text{ is the top symbol of } \mathcal{S}^L$$

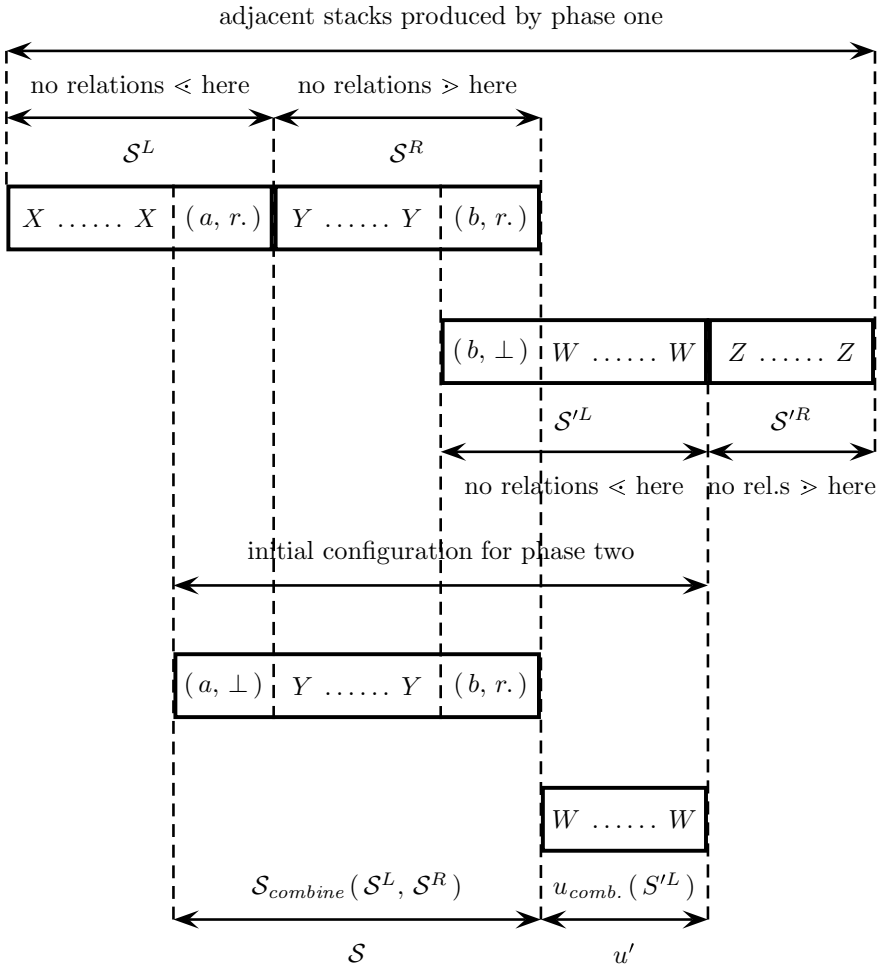


Fig. 4.59 Scheme of the initial configuration of a worker of the next phase: the stack is $\mathcal{S}$ and the input string is u'

Notice that the precedence value listed with a becomes undefined because in the new stack the a is not preceded by a terminal symbol.

2. Input string initialization. The function $u_{combine}(\mathcal{S}'^L)$ returns the string u' obtained from the stack part $\mathcal{S}'^L$ by dropping the precedence signs and deleting the first element (which is redundant as it is on top of $\mathcal{S}^R$).

There are a few simple special but important cases. When $\mathcal{S}'$ contains only $\lessdot$ and $\doteq$ precedence relations, the number of stacks can be reduced. Since in the neighbouring stack, part $\mathcal{S}^R$ too contains only relations $\lessdot$ and $\doteq$, we concatenate three pieces to obtain the new stack: $\mathcal{S}^R\ \mathcal{S}'^R$, and the left part, denoted $\mathcal{S}''^L$, of the worker that

comes after W'. As said, we need to keep only one of the overlapping elements: the last symbol of $\mathcal{S}^R$, but not the identical first symbol of $\mathcal{S}'$.

Similarly, if $\mathcal{S} = (a, \perp)\,(A, \perp)\,(b, \gtrdot)$ or $\mathcal{S} = (a, \perp)\,(A, \perp)\,(b, \doteq)$ is a quasi-empty stack, the next stack part ${\mathcal{S}'}^L$ is appended to it. Notice that the case of a quasi-empty stack $(a, \perp)\,(A, \perp)\,(b, \lessdot)$ belongs to the previous discussion.

The outline of the parallel parsing algorithm comes next. Notice that for brevity the special cases discussed above are not encoded in this outline.

Algorithm 4.90 (*operator-precedence parallel parser*)

1. **split** the input string u into k substrings: $\#\, u_1\, u_2\, \ldots\, u_k\, \#$
2. **launch** k instances of Algorithm 4.85 where for each index i with $1 \leq i \leq k$ the parameters of the i-th instance are as below:

   denote as a the last symbol of u_{i-1}
   denote as b the first symbol of u_{i+1}
   denote $u_0 = u_{k+1} = \#$
   $\mathcal{S} := (a, \perp)$
   $u := u_i\, b$
   $head := |\, u_1\, u_2\, \ldots\, u_{i-1}\, |$
   $end := |\, u_1\, u_2\, \ldots\, u_i\, | + 1$

   the result of this phase are k bipartite stacks $\mathcal{S}_i^L\ \mathcal{S}_i^R$
3. **repeat**

   for each non-quasi-empty bipartite stacks $\mathcal{S}_i^L\ \mathcal{S}_i^R$ and $\mathcal{S}_{i+1}^L\ \mathcal{S}_{i+1}^R$ **do**

   launch an instance of Algorithm 4.85 with these parameters:

   $\mathcal{S} := \mathcal{S}_{combine}\,(\mathcal{S}_i^L,\ \mathcal{S}_i^R)$
   $s := u_{combine}\,(\mathcal{S}_{i+1}^L)$
   $head := 1$
   $end := |u|$

   end for

   until (there is only one stack $\mathcal{S}$ **and** the configuration is irreducible) **or** an error is detected (triggering recovery actions not specified here)
4. **return** stack $\mathcal{S}$ □

Example 4.91 (*parallel parsing steps*) We resume from the configuration shown in Fig. 4.58 where three bipartite stacks, computed by three workers, are present:

$(\#, \bot)\ (A, \bot)\ (+, \lessdot)$	$\mathcal{S}_1$
$(+, \bot)\ (B, \bot)\ (+, \gtrdot)\ \overbrace{(n, \lessdot)}^{\mathcal{S}_2^R}$	$\mathcal{S}_2$
$(n, \bot)\ (\times, \gtrdot)\ (n, \doteq)\ (+, \gtrdot)\ (B, \bot)\ (\#, \gtrdot)$	$\mathcal{S}_3 = \mathcal{S}_3^L$

Notice that stack $\mathcal{S}_1$ is quasi-empty. In reality, the stacks are so short that the serial strategy should be applied, which simply concatenates the three stacks and takes care of the overlaps. For the sake of illustrating the other strategies, we combine stacks 2 and 3. We observe that the stack part $\mathcal{S}_3^R$ is void, hence $\mathcal{S}_3 \equiv \mathcal{S}_3^L$ holds. Step 3, i.e., **repeat**, launches Algorithm 4.85 on every initial configuration constructed as shown in Fig. 4.59. Here there is just one instance, namely the parser that combines the outcomes of workers 2 and 3, starting from the initial configuration:

$$\underbrace{\mathcal{S}_{combine}\,(\mathcal{S}_2^L, \mathcal{S}_2^R)}_{(+, \bot)\ (n, \lessdot)} \qquad \underbrace{u_{combine}\,(\mathcal{S}_3^L)}_{\times\ n\ +\ B\ \#}$$

In the order, the following reductions are performed:

$$n \rightsquigarrow B \qquad B \times n \rightsquigarrow B$$

thus obtaining the stack: $\mathcal{S} = (+, \bot)\,(B, \bot)\,(+, \gtrdot)\,(B, \bot)\,(\#, \gtrdot)$. The situation before the final round is below:

$(\#, \bot)\ (A, \bot)\ (+, \lessdot)\ (B, \bot)\ (+, \gtrdot)$	$\mathcal{S}_1\ \mathcal{S}_2^L$
$(+, \bot)\ (B, \bot)\ (+, \gtrdot)\ (B, \bot)\ (\#, \gtrdot)$	$\mathcal{S}$

The parser can use as input string simply a # and the stack obtained by concatenating the existing stacks:

$$(\#, \bot)\ (A, \bot)\ (+, \lessdot)\ (B, \bot)\ (+, \gtrdot)\ (B, \bot)\ (+, \gtrdot)\ (B, \bot)$$

Three successive reductions $A + B \rightsquigarrow A$, followed by $A \rightsquigarrow S$, complete the analysis. □

We mention a possible improvement to the strategy, which mixes the serial and parallel approaches. At step 3, if the sum of the lengths of a series of consecutive stacks is deemed short enough, then the algorithm can join all of them at once into one stack, thus applying the serial strategy to that portion of the remaining work. This strategy is advantageous when the remaining work needed to join the stacks and to synchronize the workers outbalances the useful work effectively spent to complete the construction of the syntax tree.

4.8.4.1 Parsing Complexity

Parallel parsing techniques are mainly motivated by possible speed gains and electric power saving over sequential algorithms. We briefly analyze the performance of the parallel parser, assuming that, after the first phase of segment parsing, all the resulting stacks are joined at once into one stack (as discussed above).

In terms of asymptotic complexity, we are going to argue that the parallel algorithm achieves these requirements:

1. a best-case linear speedup with respect to the number of processors
2. a worst-case performance that does not exceed the complexity of a complete sequential parsing

To meet these requirements, it is essential that the combination of stacks $\mathcal{S}_i$ and $\mathcal{S}_{i+1}$ inside step 3 of Algorithm 4.90 takes $\mathcal{O}(1)$ time, hence $\mathcal{O}(k)$ overall time for k workers. This goal is easily achieved by maintaining at run-time a marker that keeps track of the separation between the two parts $\mathcal{S}^L$ and $\mathcal{S}^R$ of each bipartite stack. The marker is initialized at the position where the first relation $\lessdot$ is detected and then updated every time a reduction is applied and a new element is shifted on the stack as a consequence of a new relation $\lessdot$. For instance in the case of $\mathcal{S}_2$ in Fig. 4.58, the marker is initialized at the position of the first $+$ symbol and there remains after the reductions $n \rightsquigarrow B$, $B \times n \rightsquigarrow B$ and $B \times n \rightsquigarrow B$, because $+ \lessdot n$ and $+ \lessdot \times$ hold. Then, when the second $+$ within this segment is shifted (the first $+$ remains because the $\gtrdot$ between the two $+$ is not matched by a corresponding $\lessdot$ at its left), the marker is advanced to the position of the second $+$, because $+ \lessdot n$ holds. In such a position the marker indicates the beginning of $\mathcal{S}_2^R$.

These operations require a time $\mathcal{O}(1)$, irrespectively of whether we implement the stacks by means of arrays or of more flexible linked lists. It follows that the overhead due to stack combination (linearly) depends on k but not on the source text length, thus proving that the worst-case asymptotic complexity is the same as in the sequential algorithm.

Truly, such analysis neglects the synchronization and communication costs between workers, which may deteriorate the performances of parallel algorithms. Such costs tend to outbalance the benefit of parallelization when the load assigned to a worker becomes too small. In our case, this happens when the text, i.e., the input stack assigned to a worker, is too small. The cost also depends on the implementation techniques used for synchronization and communication, and on their support by the hardware operations provided by the computer. Experiments [31] indicate that parsing large texts in parallel on the current multiprocessor and multi-core computers, achieves a significant speed-up.

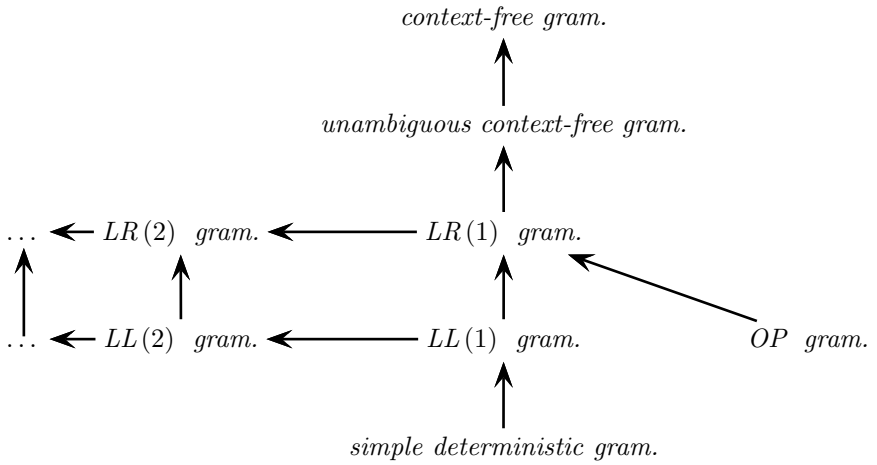


Fig. 4.60 Inclusion relationships between families of grammars

4.9 Deterministic Language Families: A Comparison

The family *DET* of *deterministic* (context-free) *languages*, characterized by the deterministic pushdown automaton as their recognizer, was defined in Sect. 4.3 on p. 215, as well as their subclasses: *simple deterministic* (p. 223) and *input-driven* (p. 223), the latter including *parenthesis languages*. Next, to complete the theoretical picture of deterministic families, we include in the comparison the families of deterministic bottom-up and top-down parsable languages, and the operator-precedence languages. In this section we do not use *EBNF* grammars and machine nets systematically, because in the literature language theoretic properties have been mostly investigated for the basic *BNF* grammar model. Notice however that, since the families *ELR* and *ELL* of extended grammars strictly include those of non-extended grammars (*LR* and *LL*), some properties for the latter immediately extend to the former.

As we proceed, we have to clarify what we actually compare: grammar or language families? An instance of the first case is the statement that every *LL* (1) grammar is also an *LR* (1) grammar; for the second case, an instance is that the family of the languages generated by *LL* (1) grammars is included in the *DET* family.

The inclusion relationships between grammar families and between language families are depicted in Figs. 4.60 and 4.61, respectively, and are now discussed case by case. Notice that some facts already known from previous sections and chapters are not repeated here.

We briefly comment the inclusions between grammar families.

In the *simple deterministic* case, for every nonterminal A the grammar alternatives must start with distinct terminals, i.e., they are $A \to b\,\beta$ and $A \to c\,\gamma$ with $b \neq c$. Therefore, the guide sets at the corresponding bifurcation from the initial state of machine M_A are disjoint, thus proving that the grammar is *LL* (1).

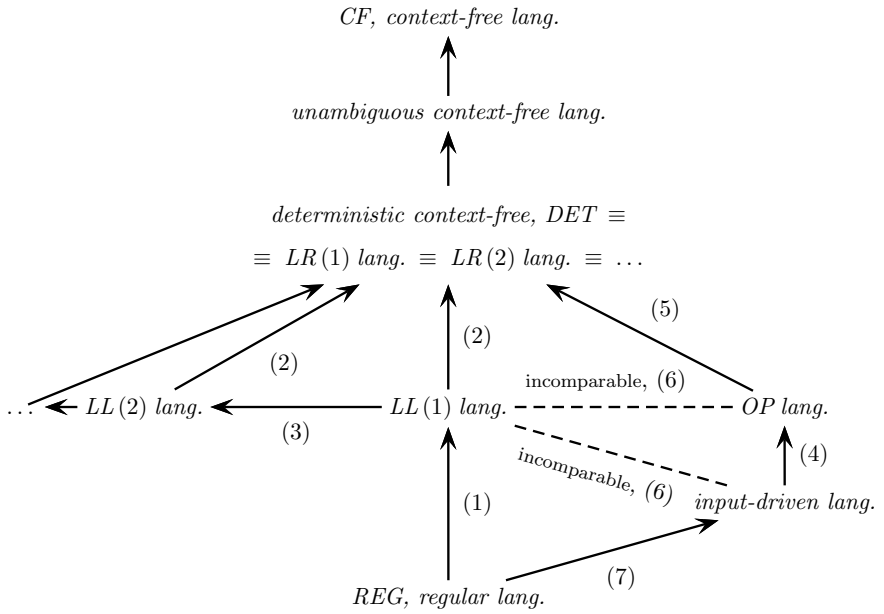


Fig. 4.61 Inclusion relationships between language families; all inclusions are strict; numbers refer to the comments

The fact that every $LL(1)$ grammar is also $LR(1)$ is obvious, since the former condition is a restriction of the latter. Moreover, the inclusion is strict: it suffices to recall that a grammar with left-recursive rules surely cannot be $LL(1)$, but it may be $LR(1)$. It is also obvious that every $LR(1)$ grammar is non-ambiguous.

It remains to comment the inclusions between the subfamilies $LL(k)$ and $LR(k)$, with $k \geq 1$. Although they have not been formally defined in this book, it is enough to think of them as obtained by extending the length of look-ahead from 1 to k characters, as we did in Sects. 4.6.6 and 4.7.5 for the $LR(2)$ and the $LL(2)$ case, respectively. In this way, we step up from the original conditions for $k = 1$, to the augmented conditions that we name $LR(k)$ and $LL(k)$ (dropping the "E" of *ELR/ELL* since the grammars are not extended). The fact that every $LR(k)$ (respectively $LL(k)$) grammar is also $LR(k+1)$ (respectively $LL(k+1)$) is trivial; moreover, the grammar in Fig. 4.52 on p. 324, witnesses the strict inclusion $ELL(1) \subset ELL(2)$, and after the natural conversion to *BNF*, also $LL(1) \subset LL(2)$.

We state without justification (the books [33,34] cover such properties in depth) that the inclusion of $LL(k)$ within $LR(k)$, which we already know to hold for $k = 1$, is valid for any value of k.

Next we exhibit examples to support the fact that every *OP* grammar meets the *LR* (1) condition.[31] Let G be an operator grammar (without copy rules and ε-rules) such that G presents an *LR* (1) conflict; does G also have *OP* conflicts? The answer is positive, as suggested by two old examples, respectively, exhibiting a shift-reduce and a reduce-reduce conflict.

Example 4.92 (*from LR* (1) *conflicts to OP conflicts*) Figure 4.18 on p. 256 details the shift-reduce conflict in the grammar: $S \to A\,b\,c \mid a\,b$ and $A \to a$. The *OP* relations contain the conflict: $a \doteq b$ and $a \gtrdot b$.

For a reduce-reduce conflict, see the grammar in Fig. 4.19 on p. 257:

$$\begin{cases} S \to A \mid B \\ A \to a\,b \mid a\,A\,b \\ B \to a\,a\,b \mid a\,a\,B\,b \end{cases}$$

Notice that this language is not deterministic. The *OP* matrix contains the conflict: $a \doteq a$ and $a \lessdot a$. □

Language Family Comparisons

We compare several language families in Fig. 4.61 (disregarding the marginal *simple deterministic* family, which fits between families *REG* and *LL* (1)). For two generic families $\mathcal{X}$ and $\mathcal{Y}$, which are characterized by certain grammar or automaton models, the proof that family $\mathcal{X}$ is included in $\mathcal{Y}$, consists of showing that every grammar (or automaton) of type $\mathcal{X}$ can be converted to a grammar (or automaton) of type $\mathcal{Y}$. For a broad introduction to such comparison methods, we refer to [1].

Before we analyze the relationships among language families, we have to state more precisely a fundamental property of deterministic languages, which we have vaguely anticipated.

Property 4.93 (LR and DET families) *The family* DET *of deterministic context-free languages coincides with the family of the languages generated by* LR (1) *grammars.* □

This important statement (due to Knuth [16]) essentially says that the shift-reduce parsing approach with look-ahead length $k = 1$ perfectly captures the idea of deterministic recognition, using as automaton a machine with a finite memory and an unbounded stack. Obviously, this does not say that every grammar generating a deterministic language enjoys the *LR* (1) property. For instance, the grammar might be ambiguous or require a look-ahead longer than one. But for sure there exists an equivalent grammar that is *LR* (1). Notice that, since the grammar family *LR* (1) is

[31] A formal comparison of precedence and *LR* (1) grammars is in Aho and Ullman [35]. Fischer [36] shows that any OP grammar is also a *simple precedence* grammar, which is a grammar type, not considered in this book, included in the *LR* (1) type.

included in the grammar family *ELR* (1), the latter too coincides with *DET*. On the other hand, any context-free language that is not deterministic, as the cases in Sect. 4.3.2 on p. 218, cannot be generated by an *LR* (1) grammar.

Consider now extending the look-ahead length, as we did in Sect. 4.6.6. Trusting the reader's intuition (or going back to Sect. 4.6.6 for a few enlightening examples), we talk about *ELR* (*k*) grammars and parsers without formally defining them. Such a parser is driven by a pilot with look-ahead sets that contain strings of length $k > 1$. Clearly, the pilot may have more m-states than the *LR* (1) pilot, because some m-states of the latter pilot are split, due to the look-ahead sets with $k > 1$ becoming different. However, since after all an *ELR* (*k*) parser is nothing more than a deterministic pushdown automaton, no matter how large parameter *k* is, from Property 4.93 we have the following one:

Property 4.94 (ELR families) *For every integer* $k > 1$*, the family of the languages generated by* ELR (*k*) *grammars coincides with the language family generated by* ELR (1) *grammars, hence also with the* DET *family.* □

Now a question arises: since every deterministic language can be defined by means of an *LR* (1) grammar, what is the use of taking higher values of the length parameter *k*? The answer comes from the consideration of the families of grammars in Fig. 4.60, where we see that there exist grammars having the *LR* (2) property, but not the *LR* (1) one, and more generally that there exists an infinite inclusion hierarchy between the *LR* (*k*) and *LR* ($k + 1$) grammar families ($k \geq 1$). Although Property 4.94 ensures that any *LR* (*k*) grammar, with $k > 1$, can be replaced by an equivalent *LR* (1) grammar, the latter grammar is sometimes less natural or simply bigger, as seen in Sect. 4.6.6 where a few grammar transformations from *LR* (2) down to *LR* (1) are illustrated. The following comments are keyed to the numbered relationships among languages, shown in Fig. 4.61:

1. Since every regular language is recognized by a *DFA*, consider its state-transition graph and normalize it so that no arcs enter the initial state: such a machine only has terminal shifts and is clearly *ELL* (1). To show that it is *LL* (1), it suffices to take the grammar that encodes each arc $p \xrightarrow{a} q$ as a right-linear rule $p \to a\, q$, and each final state f as $f \to \varepsilon$. Such a grammar is easily *LL* (1) because every guide set coincides with the arc label, except the guide set of a final state, which is the end-marker ⊣.
2. Every *LL* (*k*) language with $k \geq 1$ is deterministic, hence also *LR* (1) by Property 4.93. Of course, this is consistent with the fact that the top-down parser of an *LL* (*k*) language is a deterministic pushdown automaton. It remains to argue that the inclusion is strict, i.e., for every value of *k* there exist deterministic languages that cannot be defined with an *LL* (*k*) grammar. A good example is the deterministic language $\{\, a^* a^n b^n \mid \ n \geq 0 \,\}$, studied in Fig. 4.38 on p. 289, where the grammar with rules $S \to a^* N$ and $N \to a\, N\, b \mid \varepsilon$, and the equivalent machine net, have been found to be *ELR* (1), but to violate the *ELL* (1) condition. The reason is that, in the initial state of the net with character *a* as current token, the parser

is unable to decide whether to simply scan character a or to invoke machine M_N. Quite clearly, we see that lengthening the look-ahead does not solve the dilemma, as witnessed by the sentences $a \ \ldots \ a\, a^n\, b^n$ where the balanced part of the string is preceded by unboundedly many letters a. A natural but more difficult question is: can we find an equivalent $ELL\,(k)$ grammar for this language? The answer, proved in [37], is negative.

3. The collection of language families defined by $LL\,(k)$ grammars, for $k = 1,\ 2,\ \ldots$, forms an infinite inclusion hierarchy, in the sense that, for any integer $k \geq 1$, there exists a language that meets the $LL\,(k+1)$ condition, but not the $LL\,(k)$ one (we refer again to [33,34,37]). Notice the contrast with the $LR\,(k)$ collection of language families, which all collapse into DET for every value of the parameter k.
4. See Property 4.86 on p. 335.
5. An OP parser is a deterministic pushdown machine, see Property 4.86 on p. 335.
6. It follows from Property 4.86 on p. 335 and the strict containment of ID within OP.
7. See the remark after Definition 4.20 of input-driven automata on p. 225.

To finish with Fig. 4.61, the fact that a deterministic language is unambiguous is stated in Property 4.17 on p. 220.

For completeness, we end with a negative statement concerning decidability of the $LR\,(k)$ property.

Property 4.95 (undecidability) *For a generic context-free grammar, it is undecidable if there exists an integer $k \geq 1$ such that the grammar has the* LR (k) *property. For a generic nondeterministic pushdown automaton, it is undecidable if there exists an equivalent deterministic pushdown automaton; this is the same as saying that it is undecidable if a context-free language (presented by means of a generic grammar or pushdown automaton) is deterministic.* □

However, if the look-ahead length k is fixed, then it is possible to check whether the grammar is $LR\,(k)$ by constructing the pilot and verifying that there are not any conflicts: this is what we have done for $k = 1$ in a number of cases. Furthermore, it is not excluded that for some specific nondeterministic pushdown automaton, it can be decided if there exists a deterministic one accepting the same language; or (which is the same thing) that for some particular context-free language, it can be decided if it is deterministic; but general algorithms for answering these decision problems in all cases do not exist.

4.10 Discussion of Parsing Methods

We briefly discuss the practical implications of choosing top-down or bottom-up deterministic methods, then we raise the question of what to do if the grammar is nondeterministic and neither parsing method is adequate.

For the primary technical languages, compilers exist that use both top-down and bottom-up deterministic algorithms, which proves that the choice between the two is not really critical. In particular, the speed differences between the two algorithms are small and often negligible. The following considerations provide a few hints for choosing one or the other technique.

We know from Sect. 4.9 (p. 346 and following) that the family *DET* of deterministic languages, i.e., the *LR* (1) ones, is larger than the family of *LL* (*k*) languages, for any $k \geq 1$. Therefore there are *LR* (1) grammars that cannot be converted into *LL* (*k*) grammars, no matter how large the guide set length *k* may be. In the practice this case is unfrequent. Truly, quite often the grammar listed in the official language reference manual violates the *ELL* (1) condition because of the presence of left-recursive derivations, or in other ways, and choosing top-down parsing forces the designer to use a longer look-ahead, or to transform the grammar. The first option is less disturbing, if it is applicable. Otherwise, some typical transformations frequently suffice to obtain an *LL* (*k*) grammar: moving left recursions to the right and refactoring the alternative grammar rules, so that the guide sets become disjoint. For instance, the rules $X \rightarrow A \mid B$, $A \rightarrow a \mid c\,C$ and $B \rightarrow b \mid c\,D$ are refactored as $X \rightarrow a \mid b \mid c\,(C \mid D)$, thus eliminating the overlap between the guide sets of the alternatives of X. Unfortunately the resulting grammar is typically quite different from the original one and often less readable. Add to it that having two grammar versions to manage carries higher maintenance costs when the language evolves over time.

Another relevant difference is that a software tool (such as *Bison*) is needed to construct a shift-reduce parser, whereas recursive descent parsers can be easily coded by hand, since the layout of each syntax procedure essentially mirrors the graph of the corresponding machine. For this reason the code of such compilers is easier to understand and may be preferred by the non-specialist. Actually, tools have been developed (but less commonly used) for helping in designing *ELL* (1) parsers: they compute the guide sets and generate the stubs of the recursive syntactic procedures.

With respect to the form of the grammar, *EBNF* or pure *BNF*, we have presented parser generation methods that handle extended *BNF* rules, not just for top-down as done since many years, but also for bottom-up algorithms. Parser generator tools for *ELR* (1) grammars exist [19], but are less consolidated than those for pure *BNF* grammars, such as *Bison*.

A parser is not an isolated application, but it is always interfaced with a translation algorithm (or semantic analyzer) to be described in the next chapter. Anticipating some discussion, top-down and bottom-up parsers differ with respect to the type of translation they can support. A formal property of the syntax-directed translation methodology, to be presented later, makes top-down parsers more convenient for designing simple translation procedures, obtained by inserting output actions into the body of parsing procedures. Such one-pass translators are less expressive and less convenient to implement on top of bottom-up parsers.

If the reference grammar of the source language violates the *ELR* (hence also the *ELL*) condition, the compiler designer has several choices:

1. to modify the grammar
2. to adopt a more general parsing algorithm
3. or to leverage on semantic information

Actually this situation is more common for natural language processing (computational linguistic) than for programming and technical languages, because natural languages are much more ambiguous than the artificial ones. So skipping case 1, we discuss the remaining ones.

In case 2, the parsing problem for general context-free grammars (including the ambiguous case) has attracted much attention, and well-known algorithms exist that are able to parse any string in cubic time. The leading algorithms are named *CYK* (from its inventors Cocke–Younger–Kasami) and Earley (see for instance [18]); the latter is the object of the next section.

We should mention that other parsing algorithms exist that fall in between deterministic parsers and general parsers with respect to generality and computational performance. Derived from the *ELL* (k) top-down algorithms, parser generators exist, like *ANTLR*, that produce more general parsers: they may perform unbounded lookahead in order to decide the next move. For bottom-up parsing, a practical almost deterministic algorithm existing in several variants is due to Tomita [38]. The idea is to carry on in parallel only a bounded number of alternative parsing attempts. But the designer should be aware that any general parser more general than the deterministic ones will be slower and more memory demanding.

In case 3, sometimes an entirely different strategy for curtailing the combinatorial explosion of nondeterministic attempts is followed, especially when the syntax is highly ambiguous, but only one semantic interpretation is possible. In that case, it would be wasteful to construct a number of syntax trees, just to delete, at a later time, all but one of them using semantic criteria. It is preferable to anticipate semantic checks during parsing, and thus to prevent meaningless syntactic derivations to be carried on. Such a strategy, called *semantics-directed parsing*, is presented in the next chapter within the semantic model named attribute grammars.

4.11 A General Parsing Algorithm

The *LR* and *LL* parsing methods are inadequate for dealing with ambiguous and nondeterministic grammars. In his seminal work [39], J. Earley introduced an algorithm for recognizing the strings of arbitrary context-free grammars, which has a cubic time complexity in the worst case. His original algorithm does not construct the (potentially numerous) syntax trees of the source string, but later certain efficient representations of parse forests have been invented, which avoid to duplicate the common subtrees and thus achieve a polynomial time complexity for the tree construction problem [40]. Until very recently [41], the Earley parsers computed the syntax tree in a second pass, after string recognition [18].

Concerning the use of *EBNF* grammars, J. Earley already gave some hints about how to do, and later a few parser generators have been implemented.

Another long-standing discussion concerns the pros and cons of using look-ahead to filter parser moves. Since experiments have indicated that the look-ahead-less algorithms can be faster, at least for programming languages [42], we decided to present the simpler version that does not use look-ahead at all. This is in line with the classical theoretical presentations of the Earley parsers for non-extended grammars (*BNF*), e.g., in [18,43].

We present a version of the Earley parser for machine networks representing arbitrary *EBNF* grammars, including the nondeterministic and ambiguous ones. But for the computation of the syntax tree, we restrict our presentation to unambiguous grammars, because this book focuses on programming languages, which are well-defined unambiguous formal notations, unlike the natural languages. Thus, our procedure for building syntax trees works for nondeterministic unambiguous grammars, and it does not deal with multiple trees or parse forests.

The section continues with an introductory presentation by example, then it defines and illustrates the pure recognizer algorithm, and it finishes with the algorithm for building the syntax tree. At the section end, we mention some extensions of the Earley algorithm and other related issues.

4.11.1 Introductory Presentation

The data structures and operations of the Earley and *ELR*(1) parsers are in many ways similar. In particular, the vector stack used by the parser implementation in Sect. 4.6.5 on p. 271 is a good analogy to the *Earley vector* to be next introduced. When analyzing a string $x = x_1 \ldots x_n$ or $x = \varepsilon$ of length $n \geq 0$, the algorithm uses a vector $E\,[0 \ldots n]$ of $n + 1$ elements. Every element $E\,[i]$ is a set of *pairs*, $\langle\ q_X,\ j\ \rangle$, where q_X is a *state* of machine M_X, and j $(0 \leq j \leq i \leq n)$ is an integer *pointer* that indicates an element $E\,[j]$, preceding or coinciding with $E\,[i]$; such $E\,[j]$ contains a pair $\langle\ 0_X,\ j\ \rangle$, with the same machine M_X and the same integer j. The intended meaning of this pointer j is to mark the position $\uparrow$ in the input string x from where the current derivation of nonterminal X has started, as shown below:

$$x_1 \ldots x_j \uparrow \underbrace{x_{j+1} \ldots x_i}_{X} \ldots x_n$$

The pointer thus stretches from character x_j, excluded, to character x_i, included. If the current derivation of X is ε, we have the case $j = i$.

Thus, a pair has the form $\langle\ q_X,\ j\ \rangle$ and is named *initial* or *final* depending on the state q_X being, respectively, initial or final; a pair is called *axiomatic* if nonterminal X is the grammar axiom.

An intuitive example familiarizes us with the Earley method. It embodies the essential concepts leaving out details to be later formalized.

Example 4.96 (*introduction to the Earley method*) The language:

$$\{\, a^n\, b^n \mid \ n \geq 1 \,\} \cup \{\, a^{2n}\, b^n \mid \ n \geq 1 \,\}$$

is nondeterministic and is defined by the grammar and machine net in Fig. 4.62, which violate the *ELR* (k) condition, as any other equivalent grammar would do (the *ELR* (1) pilot is in Fig. 4.19 on p. 257. Suppose the string to be analyzed is $x = x_1\, x_2\, x_3\, x_4 = a\, a\, b\, b$ with length $n = 4$. We prepare the Earley vector E, with the elements $E\,[0]$, …, $E\,[4]$, all initialized to empty, except $E\,[0] = \langle\, 0_S,\ 0\,\rangle$. In this pair, the *state* is set to 0_S (the initial state of machine M_S), and the *pointer* is set to 0, which is the position that precedes the first character x_1 of string x. As parsing proceeds, when the current character is x_i, the current element $E\,[i]$ will be filled with one or more pairs. The final vector $E\,[0] \ldots E\,[4]$ is shown in Table 4.9.

In broad terms, operations of three types, to be next explained on the example, can be applied on the current element $E\,[i]$: *closure*, *terminal shift* and *nonterminal shift*; their names are remindful of similar *ELR* operations.

Closure The operation applies to a pair such that from its state an arc with nonterminal label X originates. Let the pair $\langle\ p,\ j\ \rangle$ be in the current element $E\,[i]$ and suppose that the net has an arc $p \xrightarrow{X} q$ with nonterminal label X; the target state q is not relevant. The operation adds to the current element $E\,[i]$ a new pair $\langle\ 0_X,\ i\ \rangle$ such that: the state is the initial one 0_X of the machine M_X of nonterminal X; the pointer has value i, to indicate that the pair was created at step i from a pair already present in $E\,[i]$.

The effect of closure is to add to element $E\,[i]$ all the pairs with the initial states of the machines that can recognize a substring starting from the next character x_{i+1}.

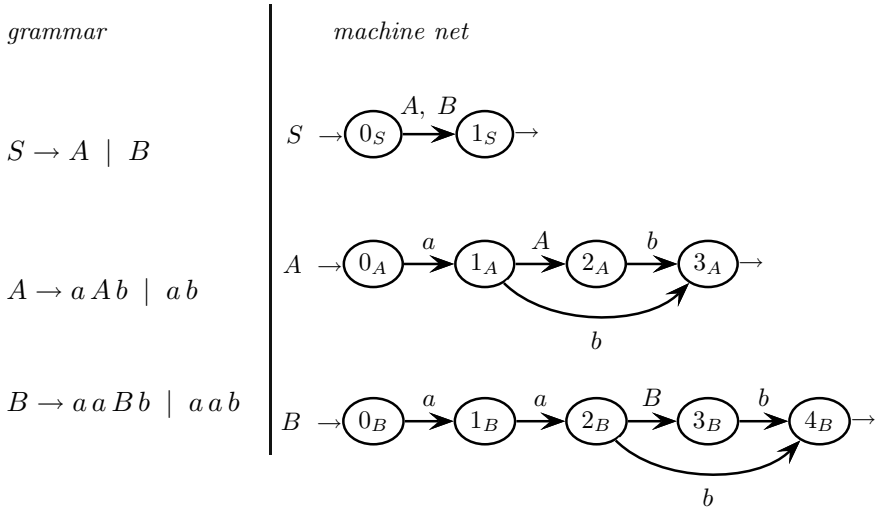


Fig. 4.62 Grammar and machine net of Example 4.96

Table 4.9 Earley parsing trace of string *a a b b* for Example 4.96

Earley vector $E\,[0\ldots4]$

$E\,(0)$		x_1	$E\,(1)$		x_2	$E\,(2)$		x_3	$E\,(3)$		x_4	$E\,(4)$	
q	j	a	q	j	a	q	j	b	q	j	b	q	j
0_S	0		1_A	0		2_B	0		4_B	0		3_A	0
0_A	0		1_B	0		1_A	1		3_A	1		1_S	0
0_B	0		0_A	1		0_B	2		1_S	0			
						0_A	2		2_A	0			

In our example, the closure writes into $E\,[0]$ the pairs:

$$\langle\, 0_A,\ 0\,\rangle \qquad \langle\, 0_B,\ 0\,\rangle$$

Since a further application of the closure to the new pairs does not create new pairs, the algorithm moves to the second operation.

Terminal shift The operation applies to a pair with a state from where a terminal shift arc originates. Suppose that the pair $\langle\, p,\ j\,\rangle$ is in element $E\,[i-1]$ and that the net has the arc $p \xrightarrow{x_i} q$ labeled by the current token x_i. The operation writes into element $E\,[i]$ the pair $\langle\, q,\ j\,\rangle$, where the state is the destination of the arc and the pointer equals that of the pair in $E\,[i-1]$, to which terminal shift applies. Having scanned x_i, the current token becomes x_{i+1}.

Here, with $i = 1$ and $x_1 = a$, the terminal shift from element $E\,[0]$ puts into $E\,[1]$ the new pairs:

$$\langle\, 1_A,\ 0\,\rangle \qquad \langle\, 1_B,\ 0\,\rangle$$

Then, closure applies again to the new pairs, with the effect of adding to $E\,[1]$ the pair (notice the pointer has value 1):

$$\langle\, 0_A,\ 1\,\rangle$$

If the current element $E\,[i]$ does not contain any pair to which terminal shift, under the current token, applies, the next element $E\,[i+1]$ remains empty, and the algorithm stops and rejects the string (in the example this never happens.).

Continuing from element $E\,[1]$ scanning character $x_2 = a$, a terminal shift puts into $E\,[2]$ the pairs:

$$\langle\, 2_B,\ 0\,\rangle \qquad \langle\, 1_A,\ 1\,\rangle$$

Through their closure, we add to $E\,[2]$ the pairs:

$$\langle\ 0_B,\ 2\ \rangle \qquad \langle\ 0_A,\ 2\ \rangle$$

Scanning character $x_3 = b$, a terminal shift writes into $E\,[3]$ the pairs:

$$\langle\ 4_B,\ 0\ \rangle \qquad \langle\ 3_A,\ 1\ \rangle$$

Now, the third type of operation, which is more complex, comes into play.

Nonterminal shift The operation is triggered by the presence in $E\,[i]$ of a pair $\langle\ f_X,\ j\ \rangle$, where f_X is a final state of machine M_X: such pair is qualified as *enabling*. Then, to locate the pairs to be shifted, the operation jumps to the element $E\,[j]$ pointed to by the pointer of the enabling pair. In this example, the pointer always directs to an element at a preceding position $j < i$. However, if the grammar contains nullable nonterminals, the pointer j may also direct to the same position $E\,[i]$, as we will see in other examples.

In the element $E\,[j]$, the nonterminal shift operation searches for a pair $\langle\ p,\ l\ \rangle$ such that the net contains an arc $p \xrightarrow{X} q$, where label X matches the machine M_X of state f_X (in the enabling pair). The pointer l may range in the interval $0 \leq l \leq j$. It can be proved that the search will find at least one such pair, and the nonterminal shift applies to it. The operation writes the pair $\langle\ q,\ l\ \rangle$ into $E\,[i]$; notice that state q is the target of the arc and the pointer value l is copied. If the search in $E\,[j]$ finds two or more pairs that fit, then the operation is applied to each one, in any order.

Here, the final pairs $\langle\ 4_B,\ 0\ \rangle$ and $\langle\ 3_A,\ 1\ \rangle$ in $E\,[3]$ enable the nonterminal shifts on symbols B and A, respectively. The pointer in pair $\langle\ 4_B,\ 0\ \rangle$ links to element $E\,[0]$, which, intuitively, is the point where the parser began to recognize nonterminal B. In $E\,[0]$ we find pair $\langle\ 0_S,\ 0\ \rangle$, and we see that the net has arc $0_S \xrightarrow{B} 1_S$ from state 0_S with nonterminal label B. Then, we add to element $E\,[3]$ the pair:

$$\langle\ 1_S,\ 0\ \rangle$$

Similarly, for the other enabling pair $\langle\ 3_A,\ 1\ \rangle$, we find the pair $\langle\ 1_A,\ 0\ \rangle$ in the element $E\,[1]$ pointed to by 1; since the net has arc $1_A \xrightarrow{A} 2_A$, we write into $E\,[3]$ the pair:

$$\langle\ 2_A,\ 0\ \rangle$$

We pause to compare terminal and nonterminal shift: the former works from $E\,[i]$ to $E\,[i+1]$, two adjacent elements, and needs an arc $p \xrightarrow{x_{i+1}} q$, which scans token x_{i+1}; the latter works from element $E\,[j]$ to $E\,[i]$, which can be separated by a long distance, and uses arc $p \xrightarrow{X} q$, which corresponds to analyze the string derived from nonterminal X, i.e., all the tokens between the two elements.

We resume the example from the point we stopped. If string x ended with x_3, then its prefix *a a b* scanned hitherto would be accepted: acceptance is demonstrated by the presence in element $E\,[3]$ of the final axiomatic pair $\langle\ 1_S,\ 0\ \rangle$, with the pointer set to

zero. In reality, one more character $x_4 = b$ has to be scanned, and from element $E\,[3]$ the terminal shift writes into element $E\,[4]$ the pair $\langle\ 3_A,\ 0\ \rangle$. Since the pair is final, it enables a nonterminal shift that goes back to element $E\,[0]$, finds pair $\langle\ 0_S,\ 0\ \rangle$ such that the net has arc $0_S \xrightarrow{A} 1_S$, and finally puts the pair $\langle\ 1_S,\ 0\ \rangle$ into element $E\,[4]$. This pair causes the whole string $x = a\,a\,b\,b$ to be accepted. The whole computation trace is in Table 4.9. The divider between vector elements separates the pairs obtained from the preceding element through terminal shift (top group), from those obtained through closure and nonterminal shift (bottom group); such a partitioning is just for visualization (an element is an unordered set); of course in the initial element $E\,(0)$ the top group is always missing. The final states are highlighted as visual help. □

4.11.2 Earley Algorithm

The Earley algorithm for machine networks representing *EBNF* grammars uses two procedures, *Completion* and *TerminalShift*. The input string is denoted $x = x_1\ \ldots\ x_n$ or $x = \varepsilon$, with $|x| = n \geq 0$. Here is *Completion*:

Completion $(E,\ i)$ - - with index $0 \leq i \leq n$

do

- - loop that computes the *closure* operation
- - for each pair that launches machine M_X

for $\Big($each pair $\langle\ p,\ j\ \rangle \in E\,[i]$ and $X,\ q \in V,\ Q$ s.t. $p \xrightarrow{X} q\Big)$ **do**

add pair $\langle\ 0_X,\ i\ \rangle$ to element $E\,[i]$

end for

- - nested loops that compute the *nonterminal shift* operation
- - for each final pair that enables a shift on nonterminal X

for $\big($each pair $\langle\ f,\ j\ \rangle \in E\,[i]$ and $X \in V$ such that $f \in F_X\big)$ **do**

- - for each pair that shifts on nonterminal X

for $\Big($each pair $\langle\ p,\ l\ \rangle \in E\,[j]$ and $q \in Q$ s.t. $p \xrightarrow{X} q\Big)$ **do**

add pair $\langle\ q,\ l\ \rangle$ to element $E\,[i]$

end for

end for

while $\big($some pair has been added$\big)$

The *Completion* procedure adds new pairs to the current vector element $E\,[i]$, by applying one or more times *closure* and *nonterminal shift* as long as pairs can be added to it. The outer loop (do-while) runs once at least, exits after the first run

if it cannot add any pair, or more generally exits after a few runs when closure and nonterminal shift cease to produce effect. Notice that *Completion* processes the nullable nonterminals by applying to them a combination of closures and nonterminal shifts. Here is *TerminalShift*:

TerminalShift $(E,\ i)$ - - with index $1 \leq i \leq n$

- - loop that computes the *terminal shift* operation
- - for each preceding pair that shifts on terminal x_i

for $\Big($each pair $\langle\ p,\ \ j\ \rangle \in E\,[i-1]$ and $q \in Q$ s.t. $p \xrightarrow{x_i} q\Big)$ **do**

add pair $\langle\ q,\ \ j\ \rangle$ to element $E\,[i]$

end for

The *TerminalShift* procedure adds to the current element $E\,[i]$, the new pairs obtained from the previous element $E\,[i-1]$ through a *terminal shift* scanning token x_i $(1 \leq i \leq n)$.

Remarks. *TerminalShift* may fail to add any pair to the element, which so remains empty. A nonterminal that exclusively generates the empty string ε never undergoes terminal shift (it is processed only by completion).

Notice that both procedures correctly work also when the element $E\,[i]$ or, respectively, $E\,[i-1]$, is empty, in which case the procedures do nothing.

Algorithm 4.97 (*Earley syntactic analysis*)

- - analyze the terminal string x for possible acceptance
- - define the Earley vector $E\,[0 \ldots n]$ with $|x| = n \geq 0$

$E\,[0] := \{\ \langle\ 0_S,\ \ 0\ \rangle\ \}$ - - initialize the first elem. $E\,[0]$

for $i := 1$ **to** n **do** - - initialize all elem.s $E\,[1 \ldots n]$

$E\,[i] := \emptyset$

end for

Completion $(E,\ 0)$ - - complete the first elem. $E\,[0]$

$i := 1$

- - while the vector is not finished and the previous elem. is not empty

while $\big(\ i \leq n\ \wedge\ E\,[i-1] \neq \emptyset\ \big)$ **do**

TerminalShift $(E,\ i)$ - - put into the current elem. $E\,[i]$

Completion $(E,\ i)$ - - complete the current elem. $E\,[i]$

$i++$

end while □

At start, the algorithm initializes $E\,[0]$ with the axiomatic pair $\langle\ 0_S,\ 0\ \rangle$ and sets the other elements (if any) to empty. Soon after the algorithm completes element $E\,[0]$. Then, if $n \geq 1$, i.e., string x is not empty, the algorithm loops on each subsequent element from $E\,[1]$ to $E\,[n]$, puts pairs (if any) into the current element $E\,[i]$ through *TerminalShift*, and finishes element $E\,[i]$ through *Completion* ($1 \leq i \leq n$). If *TerminalShift* fails to put any pairs into $E\,[i]$, then *Completion* runs idle on it. Usually, the loop iterates as far as the last element $E\,[n]$, but it may terminate prematurely if an element was left empty in a loop iteration; then it leaves empty all the following elements. We do not comment the case $n = 0$, which is simpler.

Notice the analogy between Algorithm 4.35 on p. 252, which constructs the pilot graph and the present algorithm. The former builds a pilot m-state: it shifts candidates into the base and computes closure. The latter builds a vector element: it shifts pairs into the element and computes completion, which includes closure. The analogy is partial and there are differences, too: a candidate has a look-ahead and a pair does not; a pair has a pointer and a candidate does not; and processing nonterminals through shift is not identical either. But the major difference is conceptual: Algorithm 4.35 is a tool that constructs the data structures which will control the parser, whereas the present algorithm constructs similar data structures during parsing. Therefore, such construction time does not impact on the *ELR* parser performance, but is the reason of the higher computational complexity of the Earley parser.

We specify how Algorithm 4.97 decides whether the input string x is legal for language $L\,(G)$. If the algorithm terminates prematurely, before filling the whole vector E, then string x is obviously rejected. Otherwise, the acceptance condition below applies.

Property 4.98 (Earley acceptance mode) *When the Earley algorithm (Algorithm 4.97) terminates, the string x is accepted if, and only if, the last element $E\,[n]$ of vector E contains a final axiomatic pair with zero pointer, i.e., a pair of the form $\langle\ f_S,\ 0\ \rangle$ where it holds $f_S \in F_S$.* □

We have seen a complete run of the algorithm in Table 4.9, but that grammar is too simple to illustrate other interesting points: iterations in the *EBNF* grammar (circuits in the net) and nullable nonterminals.

Example 4.99 (*grammar with iterative operators and nullable nonterminal*) Figure 4.63 lists an unambiguous nondeterministic grammar (*EBNF*) with the corresponding machine net. The string *a a b b a a* is analyzed and its Earley vector is shown in Fig. 4.64. Since the last element, $E\,[6]$, contains the axiomatic final pair $\langle\ 4_S,\ 0\ \rangle$, with pointer 0, the string is accepted as from Property 4.98. The arcs represented in Fig. 4.64 permit to follow the operations that lead to acceptance. A dotted arc indicates a closure, e.g., $\langle\ 2_S,\ 0\ \rangle \rightarrow \langle\ 0_B,\ 3\ \rangle$. A solid arc can indicate a terminal shift, e.g., $\langle\ 0_B,\ 3\ \rangle \rightarrow \langle\ 1_B,\ 3\ \rangle$ on b. A dashed arc and a solid arc that join into the same destination together indicate a nonterminal shift: the source of the dashed arc is the enabling pair, that of the solid arc is the pair to be shifted, and the common destination of the two arcs is the operation result; e.g., $\langle\ 3_B,\ 3\ \rangle \rightarrow \langle\ 3_S,\ 0\ \rangle$

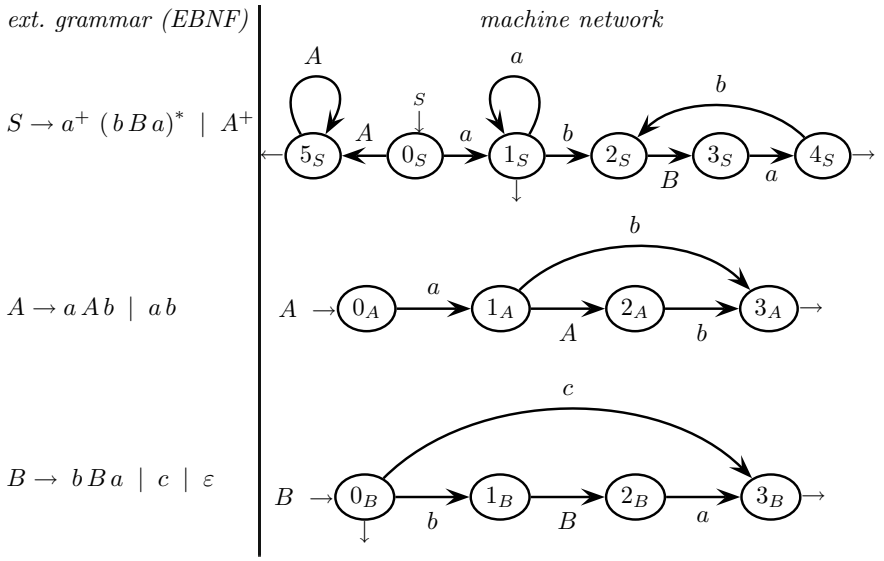


Fig. 4.63 Nondeterministic *EBNF* grammar G and network of Example 4.99

(dashed) along with $\langle\ 2_S,\ 0\ \rangle \rightarrow \langle\ 3_S,\ 0\ \rangle$ on B (solid). The label on each solid arc identifies the shift type, terminal or nonterminal.

An arc path starting with a (dotted) closure arc, continued by one or more (solid) shift arcs (terminal and non), and ended by a (dashed) nonterminal arc, represents

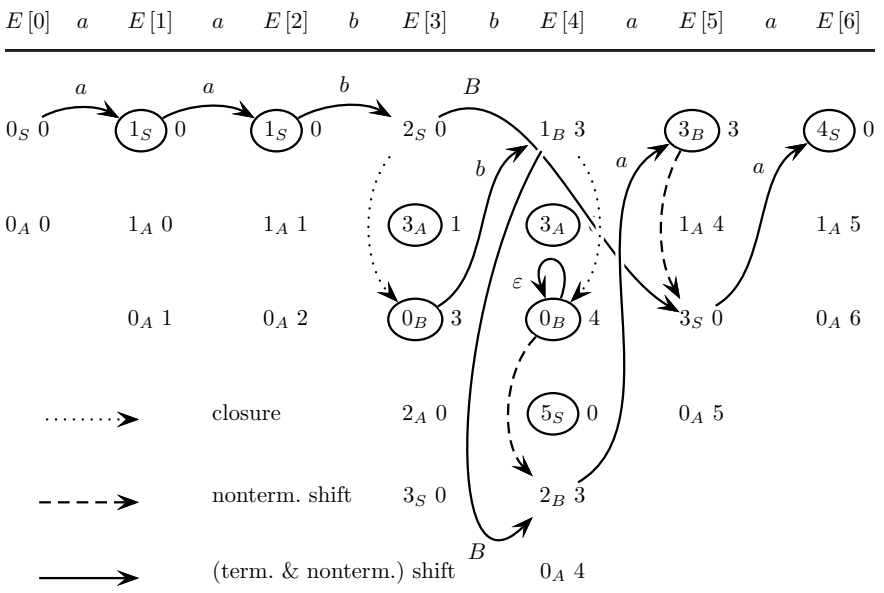


Fig. 4.64 Parse trace of string *a a b b a a* with the machine net in Fig. 4.63

a complete recognizing path within the machine of the nonterminal. The solid nonterminal shift arc that has the same origin and destination as the path cuts short the whole derivation. See for instance:

$$\langle 2_S,\ 0\rangle \overset{closure}{\underset{(dotted)}{\longrightarrow}} \langle 0_B,\ 3\rangle \underset{(sl.)}{\overset{b}{\rightarrow}} \langle 1_B,\ 3\rangle \underset{(sl.)}{\overset{B}{\rightarrow}} \langle 2_B,\ 3\rangle \underset{(sl.)}{\overset{a}{\rightarrow}} \langle 3_B,\ 3\rangle \overset{n.t.\ shift}{\underset{(dashed)}{\longrightarrow}} \langle 3_S,\ 0\rangle$$

$$\langle 2_S,\ 0\rangle \qquad \underset{nonterminal\ shift\ on\ B}{\overset{(solid)}{\longrightarrow}} \qquad \langle 3_S,\ 0\rangle$$

Notice the self-loop on pair $\langle\, 0_B,\ 4\,\rangle$ with label ε, as state 0_B is both initial and final or equivalently nonterminal B is nullable. It means that here machine M_B starts and immediately finishes the computation. Equivalently, it indicates that the corresponding instance of B immediately generates the empty string, like for instance the inner B in the arc above does.

A (solid) shift arc labeled by a nonterminal that generates the empty string, ultimately, i.e., immediately or after a few derivation steps, has its source and destination in the same vector element. See for instance in E [4]:

$$\langle\, 1_B,\ 3\,\rangle \overset{closure}{\underset{(dotted)}{\longrightarrow}} \langle\, 0_B,\ 4\,\rangle \underset{(solid)}{\overset{\varepsilon}{\longrightarrow}} \langle\, 0_B,\ 4\,\rangle \overset{n.t.\ shift}{\underset{(dashed)}{\longrightarrow}} \langle\, 2_B,\ 3\,\rangle$$

$$\langle\, 1_B,\ 3\,\rangle \qquad \underset{nonterminal\ shift\ on\ B}{\overset{(solid)}{\longrightarrow}} \qquad \langle\, 2_B,\ 3\,\rangle$$

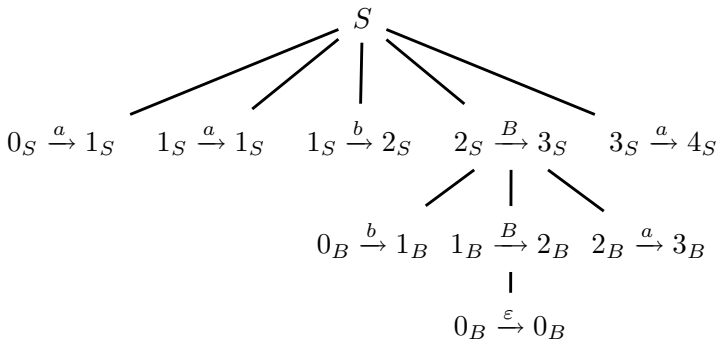
In fact, if the nonterminal shift arc $\langle\, 1_B,\ 3\,\rangle \overset{B}{\rightarrow} \langle\, 2_B,\ 3\,\rangle$ spanned a few vector elements, then nonterminal B would generate the terminals in between, not the empty string. In practice, the complete thread of arcs in Fig. 4.64 represents the syntax tree of the string, which is explicitly shown in Fig. 4.65. The nodes of the top tree are the machine transitions that correspond to the solid shift arcs (terminal and non-), while the bottom tree is the projection of the top one over the total alphabet of the grammar (terminals and nonterminals) and is the familiar syntax tree of the string for the given grammar. □

Correctness and Completeness of Earley Algorithm

Property 4.100, below, establishes a connection between the presence of certain pairs in the Earley vector, and the existence of a leftmost derivation for the string prefix analyzed so far. Along with the consequential Property 4.101, it states the *correctness* of the method: if a string is accepted by Algorithm 4.97 for an *EBNF* grammar G, then it belongs to language $L(G)$. Both properties are simple and it suffices to illustrate them by short examples.

Property 4.100 (Earley—I) *If it holds* $\langle\, q,\ j\,\rangle \in E[i]$*, which implies inequality* $j \leq i$*, with* $q \in Q_X$*, i.e., state* q *belongs to the machine* M_X *of nonterminal* X*, then*

representation with shift arcs or net transitions



projection over the total alphabet of the grammar

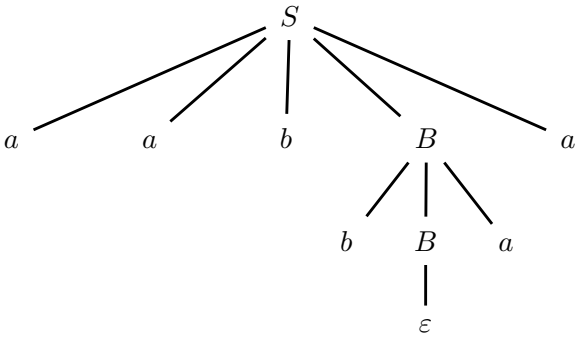


Fig. 4.65 Syntax tree for the string *a a b b a a* of Example 4.99

it holds $\langle\, 0_X,\ j \,\rangle \in E\,[j]$ *and the right-linearized grammar* $\hat{G}$ *(see Sect. 4.5.2.1 on p. 239) admits a leftmost derivation* $0_X \stackrel{*}{\Rightarrow} x_{j+1} \ldots x_i\, q$ *if* $j < i$ *or* $0_X \stackrel{*}{\Rightarrow} q$ *if* $j = i$. □

For instance: in Example 4.96 it holds $\langle\, 3_A,\ 1 \,\rangle \in E\,[3]$ with $q = 3_A$ and $1 = j < i = 3$, then it holds $\langle\, 0_A,\ 1 \,\rangle \in E\,[1]$ and grammar $\hat{G}$ admits a derivation $0_A \Rightarrow a\, 1_A \Rightarrow a\, b\, 3_A$; in Example 4.99 it holds $\langle\, 0_B,\ 4 \,\rangle \in E\,[4]$ with $q = 0_B$ and $j = i = 4$, then the property holds because state q itself is initial already.

Property 4.101 (Earley—II) *If the Earley acceptance condition (Property 4.98) is satisfied, i.e.,* $\langle\, f,\ 0 \,\rangle \in E\,[n]$ *with* $f \in F_S$*, then the* EBNF *grammar* G *admits a derivation* $S \stackrel{+}{\Rightarrow} x$*, i.e.,* $x \in L\,(S)$*, and string* x *belongs to language* $L\,(G)$. □

For instance, again in Example 4.96, it holds $\langle\, 1_S,\ 0 \,\rangle \in E\,[4]$ $(n = 4)$ with $f = 1_S \in F_S$, then the right-linearized and *EBNF* grammars $\hat{G}$ and G admit derivations

$0_S \stackrel{*}{\Rightarrow} a\,a\,b\,b\,1_S$ and $S \stackrel{*}{\Rightarrow} a\,a\,b\,b$, respectively, hence it holds $a\,a\,b\,b \in L\,(S)$ and string $a\,a\,b\,b$ belongs to language $L\,(G)$. In fact, condition $\langle\, 1_S,\ 0\,\rangle \in E\,[4]$ is the Earley acceptance one, as 1_S is a final state for the axiomatic machine M_S. This concludes the proof that Algorithm 4.97 is correct.

Property 4.102, below, contains two points 1 and 2 that are the converse of Properties 4.100 and 4.101, respectively, and states the *completeness* of the method: given an *EBNF* grammar G, if a string belongs to language $L\,(G)$, then it is accepted by Algorithm 4.97 executed for G.

Property 4.102 (Earley—III) *Take an* EBNF *grammar G and a string $x = x_1 \ldots x_n$ of length n that belongs to language $L\,(G)$. In the right-linearized grammar $\hat{G}$, consider any leftmost derivation d of a prefix $x_1 \ldots x_i$ $(i \leq n)$ of x, that is:*

$$d\colon 0_S \stackrel{+}{\Rightarrow} x_1 \ldots x_i\, q\, W$$

with $q \in Q_X$ and $W \in Q^*$, where Q is the state set of the whole net. The two points below apply:

1. *if it holds $W \neq \varepsilon$, i.e., $W = r\,Z$ for some $r \in Q_Y$, then it holds $\exists\, j\ 0 \leq j \leq i$ and $\exists\, p \in Q_Y$ such that the machine net has an arc $p \stackrel{X}{\rightarrow} r$ and grammar $\hat{G}$ admits two leftmost derivations $d_1\colon 0_S \stackrel{+}{\Rightarrow} x_1 \ldots x_j\, p\, Z$ and $d_2\colon 0_X \stackrel{+}{\Rightarrow} x_{j+1} \ldots x_i\, q$, so that derivation d decomposes as follows:*

$$\begin{aligned} d\colon 0_S &\stackrel{d_1}{\Rightarrow} x_1 \ldots x_j\, p\, Z && \stackrel{p \rightarrow 0_X\, r}{\Longrightarrow} x_1 \ldots x_j\, 0_X\, r\, Z \\ &\stackrel{d_2}{\Rightarrow} x_1 \ldots x_j\, x_{j+1} \ldots x_i\, q\, r\, Z \quad = \quad x_1 \ldots x_i\, q\, W \end{aligned}$$

   as an arc $p \stackrel{X}{\rightarrow} r$ in the net maps to a rule $p \rightarrow 0_X\, r$ in grammar $\hat{G}$
2. *this point is split into two steps, the second being the crucial one:*
   a. *if it holds $W = \varepsilon$, then it holds $X = S$, i.e., nonterminal X is the axiom, $q \in Q_S$ and $\langle\, q,\ 0\,\rangle \in E\,[i]$*
   b. *if it also holds $x_1 \ldots x_i \in L\,(G)$, i.e., the prefix also belongs to language $L\,(G)$, then it holds $q \in F_S$, i.e., state q is final for the axiomatic machine M_S, and the prefix is accepted (Property 4.98) by the Earley algorithm*

Limit cases: if it holds $i = 0$ then it holds $x_1 \ldots x_i = \varepsilon$; if it holds $j = i$ then it holds $x_{j+1} \ldots x_i = \varepsilon$; and if it holds $x = \varepsilon$ (so $n = 0$) then both cases hold, i.e., $j = i = 0$. We skip over Property 4.102 as entirely evident.

If the prefix coincides with the whole string x, i.e., $i = n$, then step 2b implies that string x, which by hypothesis belongs to language $L\,(G)$, is accepted by the Earley algorithm, which therefore is complete.

We have already observed that the Earley algorithm unavoidably accepts also all the prefixes that belong to the language, whenever in an element $E\,[i]$ before the

last one $E\,[n]$ $(i < n)$ *the acceptance condition of Property 4.98 is satisfied. For instance in Example 4.99 it holds* $\langle\ 5_S,\ 0\ \rangle \in E\,[4]$ *with state* 5_S *final, so the corresponding prefix a a b b of length* 4 *is accepted and it actually belongs to language* $L\,(G)$.

In conclusion, according to Properties 4.100–4.102, Algorithm 4.97 is both correct and complete. Hence it performs the syntax analysis of EBNF *grammars with no exceptions.* □

Computational Complexity

We have seen that the Earley parser of Algorithm 4.97 does much more work than the deterministic top-down and bottom-up parsers do, which we know to have a time complexity linear with respect to the length of the input string. But how much more does it work? It is not difficult to compute the asymptotic time complexity in the worst case. We stepwise examine the algorithm.

For the Earley parser, the chosen grammar is fixed and the input to analyze is a string. For a generic string of length $n \geq 1$, we estimate the numbers of pairs $\langle\ state,\ pointer\ \rangle$ in the Earley vector E and of basic operations performed on them to fill the vector. A basic operation consists of checking if a pair has a certain state or pointer, or of adding a new pair. We can assume all the basic operations to have a constant computational cost, i.e., $\mathcal{O}\,(1)$. Then:

1. A vector element $E\,[i]$ $(0 \leq i \leq n)$ contains a number of pairs $\langle\ q,\ j\ \rangle$ that is linear in i, as the number of states in the machine net is a constant and it holds $j \leq i$. Since it also holds $i \leq n$, we conservatively assume the number of pairs in $E\,[i]$ is linearly bounded by n, i.e., $\mathcal{O}\,(n)$.
2. For a pair $\langle\ p,\ j\ \rangle$ checked in the element $E\,[i-1]$, the terminal shift operation adds one pair to $E\,[i]$. So, for the whole $E\,[i-1]$, the *TerminalShift* procedure needs no more than $\mathcal{O}\,(n) \times \mathcal{O}\,(1) = \mathcal{O}\,(n)$ basic operations.
3. The *Completion* procedure iterates the operations of closure and nonterminal shift as long as they can add some new pair, that is until the element saturates. So, to account for all loop iterations, we examine the two operations on the whole set $E\,[i]$:

   a. For a pair $\langle\ q,\ j\ \rangle$ checked in $E\,[i]$, the closure adds to $E\,[i]$ no more pairs than the number $|\ Q\ |$ of states in the machine net, i.e., $\mathcal{O}\,(1)$. So, for the whole $E\,[i]$, the closure needs no more than $\mathcal{O}\,(n) \times \mathcal{O}\,(1) = \mathcal{O}\,(n)$ basic operations.
   b. For a final pair $\langle\ f,\ j\ \rangle$ checked in $E\,[i]$, the nonterminal shift first searches pairs $\langle\ p,\ l\ \rangle$ with certain states p through $E\,[j]$, which has a size $\mathcal{O}\,(n)$, and then adds to $E\,[i]$ as many pairs as it has found, which are no more than the size of $E\,[j]$, i.e., $\mathcal{O}\,(n)$. So, for the whole $E\,[i]$, it needs no more than $\mathcal{O}\,(n) \times \mathcal{O}\,(n) + \mathcal{O}\,(n) = \mathcal{O}\,(n^2)$ basic operations.

   Therefore, for the whole set $E\,[i]$, the *Completion* procedure in total needs no more than $\mathcal{O}\,(n) + \mathcal{O}\,(n^2) = \mathcal{O}\,(n^2)$ basic operations.

4. By summing up the numbers of basic operations performed by the procedures *TerminalShift* and *Completion* for i from 0 to n, we obtain:

$$\begin{aligned}\text{Earley cost} &= \mathit{TerminalShift} \times n + \mathit{Completion} \times (n+1) \\ &= \mathcal{O}(n) \times n + \mathcal{O}(n^2) \times (n+1) = \mathcal{O}(n^3)\end{aligned}$$

   The time cost estimate is also conservative as the algorithm may stop, before completely filling the vector, when it rejects the string.

The limit case of analyzing the empty string ε, i.e., $n = 0$, is trivial. The Earley algorithm just calls the *Completion* procedure once, which only works on the element $E[0]$. Since all the pairs in $E[0]$ have a pointer of value 0, their number is no more than $|Q|$. So the cost of the whole algorithm only depends on the network size.

Property 4.103 (Earley complexity) *The asymptotic time complexity of the Earley algorithm in the worst case is cubic, i.e., $\mathcal{O}(n^3)$, where n is the length of the string analyzed.* □

4.11.2.1 Remarks on Complexity and Optimizations

In practice, the Earley algorithm (Algorithm 4.97) performs faster in some relevant situations. For the non-extended (*BNF*) grammars that are not ambiguous, the worst-case time complexity was proved to be quadratic, i.e., $\mathcal{O}(n^2)$, by Earley himself [39]. For the non-extended (*BNF*) grammars that are deterministic, the complexity approaches a linear time, i.e., $\mathcal{O}(n)$, in most cases [18].

It is worth hinting how the non-ambiguous case works. If the grammar is not ambiguous, then a pair $\langle q, j \rangle$ is never added twice or more times to a vector element $E[i]$ $(0 \leq j \leq i \leq n)$. In fact, if it happened, the grammar would have different ways to generate a string and it would be ambiguous. Then an optimization is possible. Suppose to rearrange each element and merge all the pairs contained in it that have the same state q, into a single tuple of $t+1$ items: a state q and a series of $t \geq 1$ pointers, i.e., $\langle q, j_1, \ldots, j_t \rangle$. Since while the element builds, each pair adds only once to it, a tuple may not have any identical pointers and therefore the inequality $t \leq n+1$ holds. A shifting operation now simply means to search and find a tuple in an element, to update the tuple state and to copy the modified tuple into a subsequent element. If the destination element already has a tuple with the same state, both pointer series must be concatenated but, as explained before, the resulting tuple will not contain any duplicated pointers; the pointer series may be unordered, but this does not matter. So the rearrangement keeps and makes a sense.

Now, we can estimate the complexity. Say constant $s = |Q|$ is the number of states in the machine net. Since in a vector element each tuple has a different state, the number of tuples in there is bounded by s, i.e., a constant. Thus, the number of pointers in an element is bounded by $s \times t \leq s \times (n+1)$, i.e., linear. The *terminal shift* and *closure* operations take a linear time per element, as before. But the *nonterminal shift* operation gets lighter. Suppose to process an element: searching tuples

in the previous elements needs a number of state checks bounded by $s^2 \times (n+1)$, i.e., linear; and copying the modified tuples cannot add to the element more than a linear number of pointers, for else duplication occurs. So, the three operations altogether take a linear time per element, and quadratic for the vector. Hence, the Earley algorithm for non-ambiguous non-extended (*BNF*) grammars has a worst-case time complexity quadratic in the string length, while, without optimization, it was cubic (Property 4.103). [32]

We also mention the issue of nullable nonterminals. As explained before, in the Earley algorithm these are processed through a cascade of *closure* and *nonterminal shift* operations in procedure *Completion*. We can always put an *EBNF* grammar into non-nullable form and get rid of this issue, yet the grammar may get larger and less readable; see Sect. 2.5.13.3. But an optimized algorithm can be designed to process the nullables of *BNF* grammars in only one step; see [42]. The same authors also define optimized procedures for building the syntax tree with nullables, which are adjustable for our version of the Earley algorithm and could work with *EBNF* grammars as well.

4.11.3 Syntax Tree Construction

The next function *BuildTree BT* builds the syntax tree of an accepted string, using the vector E computed by Earley algorithm. For simplicity, *BT* works under the assumption that the grammar is unambiguous, and returns the unique syntax tree. The tree is represented as a parenthesized string, where two matching parentheses delimit a subtree rooted at some nonterminal node (such representation is described in Chap. 2 on p. 47).

Take an *EBNF* grammar $G = (V,\ \Sigma,\ P,\ S)$ and machine net $\mathcal{M}$. Consider a string $x = x_1\ \ldots\ x_n$ or $x = \varepsilon$ of length $n \geq 0$ that belongs to language $L(G)$, and suppose its Earley vector E with $n+1 \geq 1$ elements is available.

Function *BT* is recursive and has four formal parameters: nonterminal $X \in V$, state f, and two non-negative indices j and i. Nonterminal X is the root of the (sub)tree to be built. State f is final for machine M_X; it is the end of the computation path in M_X that corresponds to analyzing the substring generated by X. Indices j and i satisfy the inequality $0 \leq j \leq i \leq n$; they, respectively, specify the left and right ends of the substring generated by X:

$$X \underset{G}{\overset{+}{\Longrightarrow}} x_{j+1}\ \ldots\ x_i \quad \text{if } j < i \qquad\qquad X \underset{G}{\overset{+}{\Longrightarrow}} \varepsilon \quad \text{if } j = i$$

Grammar G admits derivation $S \overset{+}{\Rightarrow} x_1\ \ldots\ x_n$ or $S \overset{+}{\Rightarrow} \varepsilon$, and the Earley algorithm accepts string x. Thus, element $E[n]$ contains the final axiomatic pair $\langle\ f,\ 0\ \rangle$. To build the tree of string x with root node S, function *BT* is called with parameters

[32] Since this analysis is independent of the grammar being non-extended, it reasonably holds for non-ambiguous extended (*EBNF*) grammars as well. But remember that the element contents need a suitable rearrangement.

BuildTree ($S,\ f,\ 0,\ n$); then, the function will recursively build all the subtrees and will assemble them in the final tree. The commented code follows.

Algorithm 4.104 (*Earley syntax tree construction*)

BuildTree ($X,\ f,\ j,\ i$)

- - X is a nonterminal, f is a final state of M_X and $0 \leq j \leq i \leq n$
- - return as parenthesized string the syntax tree rooted at node X

- - node X will have a list $\mathcal{C}$ of terminal and nonterminal child nodes
- - either list $\mathcal{C}$ will remain empty or it will be filled from right to left

$\mathcal{C} := \varepsilon$ - - set to ε the list $\mathcal{C}$ of child nodes of X

$q := f$ - - set to f the state q in machine M_X

$k := i$ - - set to i the index k of vector E

- - walk back the sequence of term. & nonterm. shift oper.s in M_X

while ($q \neq 0_X$) **do** - - while current state q is not initial

- - try to backwards recover a terminal shift move $p \xrightarrow{x_k} q$, i.e.,
- - check if node X has terminal x_k as its current child leaf

(a) **if** $\left(\begin{array}{l} \exists\, h = k-1 \geq j\ \ \exists\, p \in Q_X \ \text{ such that} \\ \langle\ p,\ j\ \rangle \in E\,[h]\ \wedge\ \text{net has } p \xrightarrow{x_k} q \end{array} \right)$ **then**

$\mathcal{C} := x_k \cdot \mathcal{C}$ - - concatenate leaf x_k to list $\mathcal{C}$

end if

- - try to backwards recover a nonterm. shift oper. $p \xrightarrow{Y} q$, i.e.,
- - check if node X has nonterm. Y as its current child node

(b) **if** $\left(\begin{array}{l} \exists\, Y \in V\ \ \exists\, e \in F_Y\ \ \exists\, h\ j \leq h \leq k \leq i\ \ \exists\, p \in Q_X \ \text{ s.t.} \\ \langle\ e,\ h\ \rangle \in E\,[k]\ \wedge\ \langle\ p,\ j\ \rangle \in E\,[h]\ \wedge\ \text{net has } p \xrightarrow{Y} q \end{array} \right)$ **then**

- - recursively build the subtree of the derivation:
- - $Y \overset{+}{\underset{G}{\Rightarrow}} x_{h+1}\ \ldots\ x_k$ if $h < k$ or $Y \overset{+}{\underset{G}{\Rightarrow}} \varepsilon$ if $h = k$
- - and concatenate to list $\mathcal{C}$ the subtree of node Y

$\mathcal{C} := \mathit{BuildTree}\ (\ Y,\ e,\ h,\ k\ ) \cdot \mathcal{C}$

end if

$q := p$ - - shift the current state q back to p

$k := h$ - - drag the current index k back to h

end while

return ($\mathcal{C}$)$_X$ - - return the tree rooted at node X □

Essentially, BT walks back on a computation path in machine M_X and jointly scans back the vector from $E\,[n]$ to $E\,[0]$. During the traversal, BT recovers the *terminal* and *nonterminal shift* operations to identify the children (terminal leaves and internal nonterminal nodes) of the same parent node X. In this way, BT reconstructs in reverse order the shift operations performed by the Earley algorithm (Algorithm 4.97).

The while loop is the kernel of function BT: it runs zero or more times and recovers exactly one shift operation per iteration. Conditions (a) and (b) in the loop body, respectively, recover a terminal and nonterminal shift. For a terminal shift (case (a)), the function appends the related leaf. For a nonterminal shift (case (b)), it recursively calls itself and thus builds the related node and subtree. The actual parameters in the call are as prescribed: state e is final for machine M_Y, and inequality $0 \leq h \leq k \leq n$ holds.

A limit case: if the parent nonterminal X immediately generates the empty string, then leaf ε is the only child, and the loop is skipped from start (see also Fig. 4.67).

Function BT uses two local variables in the while loop: the current state q of the machine M_X; and the index k of the current Earley vector element $E\,[k]$; both variables are updated at each iteration. In the first iteration, state q is the final state f of M_X. When the loop terminates, state q is the initial state 0_X of M_X. At each iteration, state q shifts back from state to state, along a computation path of M_X. Similarly, index k is dragged back from element i to j from loop entry to exit, through a few backward long or short hops. Sometimes however, index k may stay on the same element for a few iterations; this happens if, and only if, the function processes a series of nonterminals that ultimately generate the empty string.

Function BT creates the list $\mathcal{C}$, initially set to empty, which stores the children of X, from right to left. At each iteration, the while loop concatenates (on the left) one new child to $\mathcal{C}$. A child can be a leaf (point (a)) or a node with its subtree recursively built (point (b)). At loop exit, the function returns list $\mathcal{C}$ encapsulated between labeled parentheses: $(\ \mathcal{C}\ )_X$ (see Fig. 4.67). In the limit case when the loop is skipped, the function returns the empty pair $(\ \varepsilon\ )_X$, as the only child of X is leaf ε.

Our hypothesis that the grammar is non-ambiguous has the important consequence that the conditional statements (a) and (b) are mutually exclusive at each loop iteration. Otherwise, if the grammar were ambiguous, function BT, as encoded in Algorithm 4.104, would be nondeterministic.

Two Examples

We show how function *BuildTree* constructs the syntax trees for two grammars and we discuss in detail the first case. The second case differs in that the grammar has nullable nonterminals.

Example 4.105 (*construction of syntax tree from Earley vector*) We reproduce the machine net of Example 4.96 in Fig. 4.66, top left. For input string $x_1 \ldots x_4 = a\,a\,b\,b$ the figure displays, from top to bottom, the syntax tree, the call tree of the invocations of function *BuildTree*, and the vector E returned by Earley algorithm

(Algorithm 4.97). The Earley vector (the same as in Table 4.9) is commented with three boxes, which contain the conditions of the two nonterminal shifts, and the conditions of one (out of four) terminal shift. The arrows are reversed with respect to those in Fig. 4.64, and trace the terminal and nonterminal shifts. The string is accepted thanks to pair $\langle\ 1_S,\ 0\ \rangle$ in element E [4] (Property 4.98). The initial call to function *BuildTree* is $BT\ (\ S,\ 1_S,\ 0,\ 4\ )$, which builds the tree for the whole string $x_1\ \ldots\ x_4 = a\ a\ b\ b$.

The axiomatic machine M_S has an arc $0_S \xrightarrow{A} 1_S$. Vector E contains the two pairs: $\langle\ 3_A,\ 0\ \rangle \in E$ [4], which has a final state of machine M_A and so enables a shift operation on nonterminal A; and $\langle\ 0_S,\ 0\ \rangle \in E$ [0], which has a state with an arc directed to state 1_S and labeled with nonterminal A. Thus, condition (b) in the while loop of Algorithm 4.104 is satisfied. Therefore, the first recursive call $BT\ (\ A,\ 3_A,\ 0,\ 4\ )$ is executed, which builds the subtree for the (still whole) (sub)string $x_1\ \ldots\ x_4 = a\ a\ b\ b$.

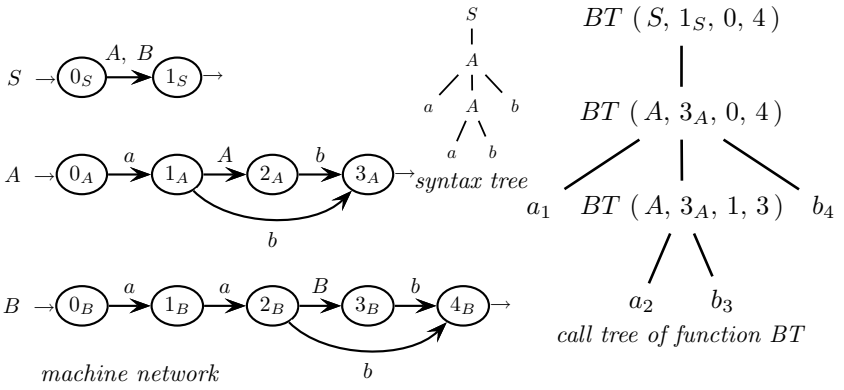
This invocation of function *BuildTree* walks back vector E from element E [4] to E [1] and recovers the terminal shift on b, the nonterminal shift on A and the terminal shift on a. When processing the nonterminal shift on A with reference to element E [3], another recursive call occurs because vector E contains the two pairs $\langle\ 3_A,\ 1\ \rangle \in E$ [3] and $\langle\ 1_A,\ 0\ \rangle \in E$ [1], and machine M_A has an arc $1_A \xrightarrow{A} 2_A$. Therefore, condition (b) in the while loop is satisfied and the second recursive call $BT\ (\ A,\ 3_A,\ 1,\ 3\ )$ is executed, which builds the subtree for substring $x_2\ x_3 = a\ b$.

The two boxes at the bottom of Fig. 4.66 explain the nonterminal shifts (point (b) in Algorithm 4.104), pointed to by a double arrow. In detail, the formal parameters X and j (upper side) have the actual values of the current invocation of *BuildTree*. Local variables q and k (lower side) have the values of the current iteration of the while loop. Inside a box we find in order: the values assumed by nonterminal Y, final state e, index h and state p; second, some logical conditions on pairs and arc; and, last, the call to BT with its actual parameters. Each box so justifies one iteration, where condition (b) is satisfied, whence a call to BT occurs and a subtree is constructed.

For terminal shifts the situation is simpler: consider the comment box in Fig. 4.66 and compare it with the if-then statement (a) in Algorithm 4.104. Three more terminal shifts take place, but we do not describe them. □

Example 4.106 (*syntax tree with nullable nonterminals*) Figure 4.67 shows the call tree of function *BuildTree* of Example 4.99. The corresponding parse trace is in Fig. 4.64 on p. 360: to see how function BT walks back the Earley vector and constructs the syntax tree, imagine to reverse the direction of solid and dashed arcs (the dotted arcs of closure are not relevant). Figure 4.67 also shows the parenthesized subtrees returned by each call to BT.

The innermost call to function *BuildTree* in Fig. 4.67 is $BT\ (\ B,\ 0_B,\ 4,\ 4\ )$, which constructs the subtree of an instance of nonterminal B that immediately generates the empty string ε. In fact, the two indices j and i are identical to 4 and their difference is zero; this means that the call is on something nullable inserted between terminals b_4 and a_5. □



Earley vector and how function BT walks it back for constructing the syntax tree above

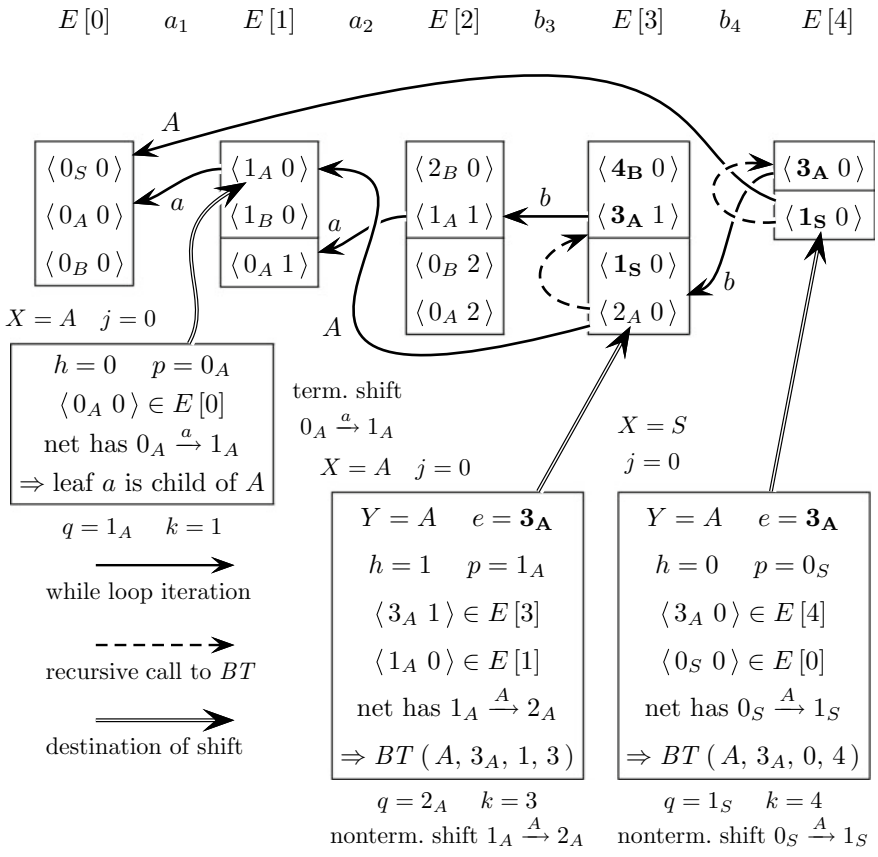


Fig. 4.66 Actions and calls of function *BuildTree* on the shifts for Example 4.96

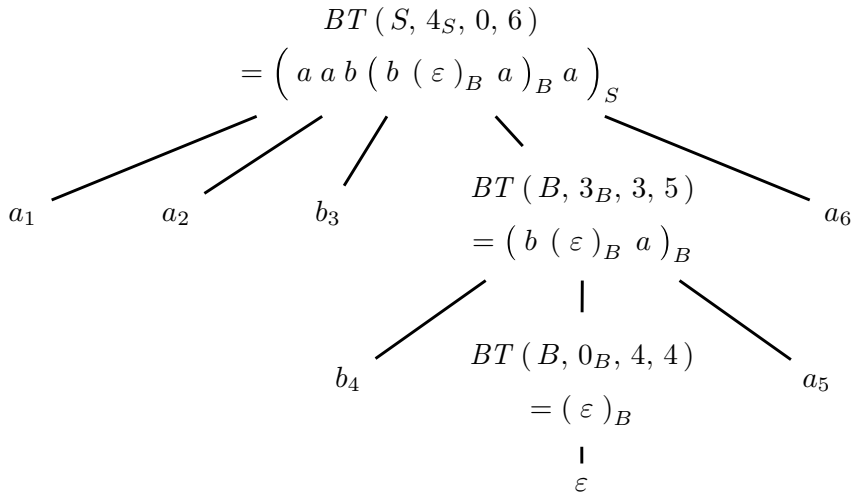


Fig. 4.67 Calls and return values of *BuildTree* for Example 4.99

Computational Complexity

The worst-case computational complexity of function *BuildTree* can be estimated by computing the size of the constructed syntax tree, the total number of nodes and leaves. Such total is directly related to the number of terminal and nonterminal shifts that function *BT* has to recover.

The tree to be built is only one for the string accepted, as the grammar is non-ambiguous. Since the grammar is also clean and free from circular derivations, for a string of length n the number of tree nodes is linearly bounded by n, i.e., $\mathcal{O}(n)$. The basic operations are those of checking the state or pointer of a pair, and of concatenating one leaf or node (i.e., its parenthesized representation) to the child list; both operations take a constant time, for a suitable parenthesized representation of the tree. The following analysis mirrors the complexity analysis of the Earley algorithm (function *BT* uses k as current index to vector E):

1. A vector element $E\,[k]$ $(0 \leq k \leq n)$ contains a number of pairs of magnitude $\mathcal{O}\,(n)$.
2. Checking the condition of the if-then statement (a) of Algorithm 4.104 takes a constant time, i.e., $\mathcal{O}\,(1)$. The possible enlisting of one leaf takes a constant time as well. So, processing the whole $E\,[k-1]$ takes a time of magnitude $\mathcal{O}\,(n) \times \mathcal{O}\,(1) + \mathcal{O}\,(1) = \mathcal{O}\,(n)$.
3. Checking the condition (b) needs to identify the pairs: (1) $\langle\; e,\;\; h\;\rangle$ in $E\,[k]$; and (2) $\langle\; p,\;\; j\;\rangle$ in $E\,[h]$. For each potential candidate pair (1) in $E\,[k]$, the whole $E\,[h]$ has to be searched to find the corresponding candidate pair (2) (if any); the search takes a time of magnitude $\mathcal{O}\,(n)$. The possible enlisting of one (nonterminal) node takes a constant time, i.e., $\mathcal{O}\,(1)$. So, processing the whole $E\,[k]$ takes a time of magnitude $\mathcal{O}\,(n) \times \mathcal{O}\,(n) + \mathcal{O}\,(1) = \mathcal{O}\,(n^2)$.

4. Since the total number of terminal plus nonterminal shifts (a) and (b) to be recovered is bounded by the number of tree nodes, i.e., $\mathcal{O}(n)$, the total time cost of function *BT* is no more than:

$$\begin{aligned} BT\,\text{cost} &= \big(\text{if-then (a)} + \text{if-then (b)}\big) \times \#\,\text{of tree nodes} \\ &= \big(\mathcal{O}(n) + \mathcal{O}(n^2)\big) \times \mathcal{O}(n) = \mathcal{O}(n^3) \end{aligned}$$

Property 4.107 (tree building complexity) *The asymptotic time complexity of function* BuildTree *to construct a syntax tree for a non-ambiguous grammar (Algorithm 4.104), is cubic in the worst case, i.e.,* $\mathcal{O}(n^3)$*, where n is the length of the string accepted.* □

If we consider that function *BuildTree* (Algorithm 4.104) just reads the Earley vector E and does not write it, the above complexity can be lowered by pre-ordering the vector. Suppose every element $E[k]$ is independently ordered: pairs $\langle\, q,\ j \,\rangle$ are ranked by the value of pointer j ($0 \le j \le n$); ranking by state q does not matter as the number of states is a constant. Ordering each element takes a time $\mathcal{O}(n \log n)$, e.g., by the QuickSort algorithm. So, ordering all the elements takes a time $(n+1) \times \mathcal{O}(n \log n) = \mathcal{O}(n^2 \log n)$. After ordering the elements, function *BT* can regularly run as before.

The benefit of pre-ordering is mainly for the time cost of the if-then statement (b) in function *BuildTree*. Now, finding a pair with some final state e and pointer h in $E[k]$ takes a time $\mathcal{O}(n)$; and consequently searching the related pair with a fixed pointer j in $E[h]$ takes a time $\mathcal{O}(\log n)$, e.g., by the Binary Search algorithm. So, statement (b) takes a time $\mathcal{O}(n \log n)$. Similarly, in the if-then statement (a) we can use Binary Search for searching a pair with fixed pointer j in $E[h]|_{h=k-1}$, so statement (a) takes a time $\mathcal{O}(\log n)$. Therefore, the total time cost of function *BT* is no more than as below:

$$\begin{aligned} BT\ \text{cost} &= \text{pre-ordering}\ E + \big(\text{if-then (a)} + \text{if-then (b)}\big) \times \#\,\text{of tree nodes} \\ &= \mathcal{O}(n^2 \log n) + \big(\mathcal{O}(\log n) + \mathcal{O}(n \log n)\big) \times \mathcal{O}(n) = \mathcal{O}(n^2 \log n) \end{aligned}$$

This new complexity of *BuildTree*, for non-ambiguous grammars, is lower than the previous cubic one without pre-ordering (Property 4.107), and it approaches that of the Earley algorithm rearranged for non-ambiguous grammars, too.

We conclude by warning again that the version of function *BuildTree* presented here is not designed to properly work with ambiguous grammars. Anyway, since the grammars of programming languages are usually non-ambiguous, such a limitation is not relevant for compilation. However, ambiguous grammars are unavoidable in natural language processing. There, the main difficulty is how to compactly represent all the syntax trees the string may have, the number of which may be exponential—and even unbounded if there are nullable nonterminals—in the string length. This can be done by using a so-called Shared Packed Parse Forest (*SPPF*), which is a graph type more general than a tree but that still takes a worst-case cubic time for building; see [40] for how to proceed in such more general cases.

A practical parser for a programming language usually works on a syntax tree represented as a graph, with variables that model nodes and leaves, and pointers that connect them; not on a tree represented as a parenthesized string. The function *BuildTree* in Algorithm 4.104 can be easily adapted to build such a tree, without changing its algorithmic structure. Here is how to recode the three crucial statements of *BT* (the rest is unmodified):

original encoding	*recoded for a pointer-connected tree*
$\mathcal{C} := x_k \cdot \mathcal{C}$	*create leaf* x_k *and attach it to node* X
$\mathcal{C} :=$ *BuildTree* $(\ Y,\ e,\ h,\ k\ ) \cdot \mathcal{C}$	*create node* Y*, call BuildTree* $(\ Y,\ \ldots\ )$ *and attach node* Y *to node* X
return $(\ \mathcal{C}\ )_X$	*return the pointer to node* X

If upon return node X does not have any child nodes or leaves yet, leaf ε has to be attached to it. It is not difficult to encode the instructions above by means of a library for managing of graph structures, as many are available. The worst-case time complexity of this variant of *BuildTree* is still cubic. See also [18] for an overview of a few parsing methods and practical variants.

4.12 Managing Syntactic Errors and Changes

In this section we address two additional features that are requested from a parser: the capability to manage user errors in a text, and the ability to efficiently reparse the modified text when the errors are fixed, or more generally when some changes have been made.

4.12.1 Errors

In human communication the hearer usually tries to correct the errors that quite frequently occur in utterances, to assign a likely meaning to the sentence; if he fails, he may prompt the utterer for a correction. On the other hand, for artificial languages the communication partners can be human beings and machines, and it helps to separate three cases: machine-to-human (as in automatic answering systems), machine-to-machine (e.g., *XML*) and human-to-machine, where programming languages are the typical case. Since "errare humanum est", but machines are supposed not to make mistakes, we have to deal only with the last case: a handwritten document or program is likely to contain errors and the machine has to be capable to suitably react.

The simplest reaction is to stop processing the text as soon as the machine *detects* an error, and to display a diagnostic message for the author. A more complete reaction involves *recovery* from error, i.e., the capability to resume processing the text. Yet more ambitious but harder to obtain is the capability to automatically *correct* the error.

Historically, in the old times when a central computer slowly processed batches of texts submitted by many users, the ability to detect all or most errors in a single compilation was a must, in order to avoid time-consuming delays caused by repeated compilations after a single error had been corrected by the user. Nowadays, compilers are interactive and faster, and a treatment of all the errors at once is no longer requested, while it remains important to provide an accurate diagnostic for helping the user to fix the error.

In the following, first we classify the error types, then we outline some simple strategies for error recovery in parsers.

4.12.1.1 Error Types

The errors a person makes in writing can be *subjective* or *objective*. In a subjective error, the text remains formally correct both syntactically and semantically, but it does not reflect the author's intended meaning. For instance, the typo $x - y + z$ instead of $x \times y + z$ is a subjective error, which will alter the program behavior at run-time in rather unpredictable ways, yet it cannot be catched at compile time. A human error is objective if it causes a violation of the language syntax or semantic specifications.

Sometimes, undetected errors cause the compiled program to raise fault conditions at run-time, such as memory errors (segmentation, overflow, etc.) or nontermination. In such cases we say the program violates the *dynamic* (or run-time) *semantic* of the language. On the other hand, the errors that can be detected by the compiler or by the static semantic analyzer (described in the next chapter) are called *static*. It pertains to the run-time support (interpreter, virtual machine or operating system) of the language to detect and recover from dynamic errors, perhaps by invoking suitable exception handlers. The methods used to detect subjective errors and to manage the dynamic errors they induce belong to software engineering, to operating system and virtual machine design and are out of scope for this book.

Therefore, we only deal with objective static errors, which can be further classified as *syntactic* and *semantic*. A syntactic error violates the context-free grammar that defines the language structure, or the regular expression that defines the lexicon. A semantic error violates some prescription included in the language reference manual, but not formulated as a grammar rule. Typically, such prescriptions are specified in English; less frequently, they are formalized using a formal or semi-formal notation (the attribute grammars of Chap. 5 are an example). For languages such as Java, the so-called *type violations* are a primary kind of semantic error: e.g., using as *real* a variable that was declared to be *boolean*; or using three subscripts to access a one-dimensional array; and many others. To detect and manage static semantic errors, the compiler builds certain data structures, called *symbol tables* that encode the properties of all the objects that are declared or used in a program.

Since this chapter deals with parsing, we present some basic ideas for the detection, diagnosis and recovery of syntactic errors. A general discussion of error processing in parsers is in the book [18].

4.12.1.2 Syntactic Errors

We examine how a parser detects a syntactic error. To keep our analysis general enough, we avoid implementation details and we represent the parser as a pushdown automaton, which scans the input from left to right until either the end-of-text is reached or the move is not defined in the current configuration, to be then called *erroneous*. The current token, for which the move is undefined, is named the *error token*. From a subjective point of view, the mistake or typo the author made in writing the text may have occurred at a much earlier moment. A negative consequence is that the parser diagnosis of the erroneous configuration may have little resemblance with the author's subjective mistake. In fact, the input string ending with the token that the human erroneously typed, though different from the one the author had in mind, may be the initial part (prefix) of a legal sentence of the language, as next illustrated.

Example 4.108 (*human error and error token*) In the *FORTRAN* language a loop such as DO 110 J = 9, 20 is defined by the following rule:

$$\langle \text{iterative instruction} \rangle \rightarrow \text{DO } \langle \text{label} \rangle \text{ ' = ' } \langle \text{initial value} \rangle \text{ ',' } \langle \text{step} \rangle$$

Suppose the human error is to omit the blank spaces around the label and so to write DO110J = 9, 20; now token DO110J is a legal variable identifier and the string DO110J = 9 hitherto analyzed is a legal assignment of value 9 to it. The parser will detect the error on reading the comma. The delay between the first human error and the recognized error token is: DO $\underbrace{\text{110J = 9}}_{delay}$, 20.

A more abstract example from the regular language below:

$$L = d\,b^*\,a\,a^* \;\cup\; e\,b^*\,c\,c^*$$

happens with the illegal string $x = d\,b\,b\,\ldots\,b\overset{\downarrow}{c}\,c\,c\,c\,c\,c$, where the first error token is marked by the arrow, whereas it is more likely that the human error was to type d for e, because for the two ways to correct x into $x' = e\,\underbrace{b\,b\,\ldots\,b}_{delay}\,c\,c\,c\,c\,c\,c$ or $x'' = d\,b\,b\,\ldots\,b\,a\,a\,a\,a\,a\,a$, string x' requires a single change, but string x'' needs as many as six changes. Assuming the more likely correction, the delay between human error and error token is unbounded. □

The second example has introduced the idea of measuring the differences between the illegal string and the presumably correct one, and thus of obtaining a metric called *editing* or *Levenshtein distance*. In the example, the distance of x' from x is one and of x'' from x is six. Moreover, we say that string x' is the *interpretation at minimal distance* from x, as no other sentence of language L has distance one from x. To give a precise definition of editing distance, one has to indicate which editing operations are considered.

A typical choice for typed texts includes the following operations, which correspond to frequent typing mistakes: *substituting* a character for another;[33] *erasing* a character; and *inserting* a character. After detecting an erroneous token, it would be desirable to compute a minimal distance interpretation, which is a legal sentence as similar as possible to the scanned illegal string. Such a sentence is a reconstruction of what the typist intended to write, in accordance with a model that assigns a lower probability to the mutations involving many editing changes.

Unfortunately, the problem of computing a minimal distance interpretation for context-free languages is computationally complex because of the unlimited spatial distance between the error token and the position where the editing corrections are due. A further difficulty comes from that it is not known how the string will continue, thus what is a minimal distance interpretation so far, may turn out to necessitate too many editing changes when the input is further scanned. Therefore the typical error recovery techniques used by compilers, renounce to the minimal distance goal, yet they attempt to stay away from the opposite case necessitating as many corrections as the string length.[34]

Error Messages and Error Recovery

In Fig. 4.68 we represent the parser as a deterministic *PDA*. By definition of error token, the parser cannot move ahead, i.e., the transition function is undefined for the input b_0 and the top symbol A_n. A straightforward (but not very informative) diagnostic message is: "token b_0 is not acceptable at line r". This message can be made more precise by reporting the parser configuration, as we next explain for the case of a deterministic top-down parser.

As we know, each symbol A_i in the stack carries two pieces of information: a nonterminal symbol of the grammar (i.e., a net machine or a recursive procedure); and an internal state of that machine (i.e., the label of a procedure statement). We assume that the active machine is associated with A_n. For the current instruction two cases are possible: the instruction scans a token e; or the instruction invokes a

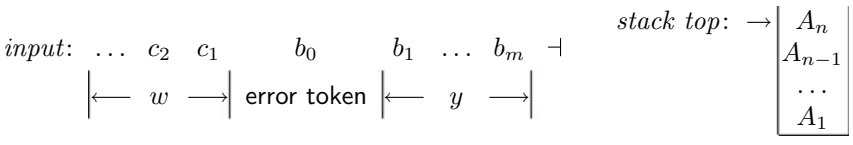


Fig. 4.68 *PDA* parser configuration when an error is detected. The input string is $z = w\, b_0\, y\ \dashv$, character b_0 is the error token, string w is the already scanned prefix, and string $y = b_1 \ldots b_m$ is the suffix still to be scanned. The pushdown stack contains $\alpha = A_1\, A_2 \ldots A_n$, where A_n is the top symbol

[33] The probability of a substitution error can be made more accurate by taking into account the key distances on the keyboard: two adjacent keys are more likely than two separated keys to be pressed one for the other.

[34] An example of minimum distance method for error handling is in [44].

machine/nonterminal E with guide set $Gui(E)$. Then the message says: "at line r I expected token e (or the tokens in $Gui(E)$ in the latter case) instead of token b_0".

Even with this improvement, such error messages can be misleading. Imagine that the subjective error omitted the first *begin* token in a string made of nested *begin* ... *end* blocks. The first error configuration of the parser occurs when symbol b_0 is the last *end* token of the series, while the parser expects to be outside of any block. The previous message tells only that token *end* is out of place, but gives no cue for finding the subjective error: the missing token *begin* could be in many possible positions in the text. To produce a more useful diagnostic help, some parsers incorporate a language-specific knowledge, such as statistics about the most frequent errors.

Error Recovery

To complete this brief discussion, we outline a simple technique for error recovery. To resume parsing after an error, we need to intervene on one at least of the arguments (input token and top-of-stack symbol) of the *PDA* transition function, which is undefined in the current configuration. Notice that parsers of different types may have as third argument the internal state of the *PDA*, which is irrelevant in the case we are considering. It is easy to modify such arguments to make the next move possible, but it is more difficult to ensure that the following moves do not sink the parser into a new error configuration, or worse into a series of errors and meaningless diagnostics.

A most elementary recovery technique, named *panic mode*, keeps skipping the next tokens $b_1, b_2, \ldots, \dashv$, until for some token b_j the transition function becomes defined for arguments b_j and A_n, as this technique does not modify the stack. The intervening tokens from b_1 to b_{j-1} are ignored, thus causing the compiler to skip the analysis of potentially large text sections.

Example 4.109 (*panic mode*) Consider a toy language containing two statement types: assignments such as $i = i + i + i$, where symbol i denotes a variable or procedure identifier; and procedure calls with actual parameters, such as *call* i $(\ i + i,\ i,\ i + i + i\ )$. We illustrate the panic mode error recovery for the following (erroneous) text:

$$\underbrace{\textit{call}\ i}_{\text{scanned text}}\ \underbrace{=\ i\ +\ i\ \ldots\ +\ i\ \dashv}_{\text{panic—skipped text}} \qquad \text{error} \tag{4.27}$$

The error is detected after correctly scanning two tokens, *call* and i. The parser reacts to the error event in this way: first it writes the diagnostic "I expected a left parenthesis and found an equal sign ="; then it panics and skips over a substring of unbounded length (including the error token), possibly generating useless diagnostics, as far as the very end of the text. So it can eventually recover a procedure call without parameters, *call* i, which may be the author's intended meaning or not. However, skipping has the annoying effect that long text portions, which may contain more errors, are not analyzed; discovering such errors would require repeated compilation.

Notice that the text admits legal interpretations at small editing distance: deleting token *call* yields the assignment $i = i + i \cdots + i$; and substituting token '=' with '(' and then inserting token ')' at the end before '⊣' yields the invocation *call* i $(i + i \cdots + i)$ of a procedure with one parameter. □

To suggest possible improvements over the panic technique, we observe that to exit the error configuration shown in (4.27), it suffices to let the parser *insert* tokens, according to its grammatical expectations. In fact, starting again from text (4.27), the parser is processing a statement belonging to the syntactic class of procedure calls, since it has already correctly scanned the *call* and *i* tokens when it hits the erroneous equal sign '='. Inserting before the error token a left parenthesis token '(', which is what the parser expects at this point instead of the equal sign '=', yields:

$$call\ i \overset{\text{inserted}}{(}\ = i + i \cdots + i \dashv$$

and thus allows one more parsing step, which scans the inserted token. Then the parser hits again the equal sign '=', which is the error token:

$$call\ i\ (\overset{\text{error}}{=}\ i + i \cdots + i \dashv$$

Now the parser, having already inserted and not wanting to alter the original text too much, resorts to the panic technique, skips over the equal sign '=' (or simply deletes it) and correctly parses the following piece of text $i + \cdots + i$ since this piece fits as actual parameter for the procedure, without any further errors as far as the end-of-text '⊣', which becomes the new error token:

$$call\ i\ (\overset{\text{skipped (deleted)}}{=}\ i + i \cdots + i\ \overset{\text{error}}{\dashv}$$

The last action is to insert a right parenthesis ')' before the end-of-text '⊣', as expected (since now the syntactic class is that of a parameter list):

$$call\ i\ (i + i \cdots + i\ \overset{\text{inserted}}{)}\ \dashv$$

Then the analysis concludes correctly and recovers a procedure call that has an expression as its actual parameter: *call* i $(i + i \cdots + i)$. In total the parser performs two insertions and one deletion, and the loss of text is limited. We leave to the reader to imagine the suitable diagnostic messages to be produced each time the parser intervenes on the error configuration.

We have seen that a careful combination of token insertions and skips (or deletions), and possibly also of token substitutions, can reduce the substrings that are skipped and produce more accurate diagnostics. An interesting method for error recovery suitable to *LL* and *LR* parsing is in [45].

4.12.2 Incremental Parsing

In some situations another capability is requested from a parser (more generally from a compiler): to be able to incrementally process the grammar or the input string. As a matter of fact, the concept of incremental compilation takes two different meanings.

incrementality with respect to the grammar This situation is not common and occurs when the source language is subjected to change, implying the grammar is not fixed. When changes are maybe minor but frequent, it is rather annoying or even impossible to create a new parser after each change. Such is the case of the so-called extensible languages, where the language user may modify the syntax of some constructs or introduce new instructions. It is then mandatory to be able to automatically construct the new parser, or better to incrementally modify the existing one after each syntax change.[35]

incrementality with respect to the source string A more common requirement is to quickly reconstruct the syntax tree after some change or correction of the input string has taken place.

A good program construction environment should interact with the user and allow him to edit the source text and to quickly recompile it, so minimizing time and effort. In order to reduce the recompilation time after a change, incremental compilation methods have been developed that are based on special algorithms for syntax and semantic analysis. Focusing on the former, suppose the parser has analyzed a text, identified some kind of error and produced an error message. Then the author has made some corrections, typically in a few points of the text.

A parser qualifies as incremental if the time it takes for analyzing the corrected text is much shorter than the time for parsing it the first time. To this end, the algorithm has to save the result of the previous analysis in such a form that updates can be just made in the few spots affected by changes. In practice, the algorithm saves the configurations traversed by the pushdown automaton in recognizing the previous text and rolls back to the most recent configuration that has not been affected by the changes to the text. From there the algorithm resumes parsing.[36] We notice that the local parsability property of operator-precedence grammars (exploited in Sect. 4.8.1 for parallel parsing) is also valuable for obtaining incremental parsers.

References

1. Harrison M (1978) Introduction to formal language theory. Addison Wesley, Reading
2. Hopcroft J, Ullman J (1969) Formal languages and their relation to automata. Addison-Wesley, Reading

[35] Some valuable methods for the incremental generation of parsers are in [46].
[36] The main ideas and algorithms for incremental parsing can be found in [47,48].

3. Hopcroft J, Ullman J (1979) Introduction to automata theory, languages, and computation. Addison-Wesley, Reading
4. Salomaa A (1973) Formal languages. Academic, New York
5. Salomaa A (1996) A direct complement construction for LR(1) grammars. Acta Inf 33(8):781–797
6. Senizergues G (2002) L(A) = L(B)? A simplified decidability proof. Theor Comput Sci 281:555–608
7. Floyd RW, Beigel R (1994) The language of machines: an introduction to computability and formal languages. Computer Science Press, New York
8. Simovici D, Tenney R (1999) Theory of formal languages with applications. World Scientific, Singapore
9. Alur R, Madhusudan P (2004) Visibly pushdown languages, STOC '04. ACM, New York. pp 202–211
10. Okhotin A, Salomaa K (2014) Complexity of input-driven pushdown automata. SIGACT News 45(2):47–67
11. Okhotin A, Salomaa K (2009) Adding nesting structure to words. J ACM 56(3):16:1–16:43
12. Baier C, Katoen JP (2008) Principles of model checking. MIT Press, Cambridge
13. Clarke EM Jr, Grumberg O, Peled DA (1999) Model checking. MIT Press, Cambridge
14. Berstel J, Boasson L (2002) Formal properties of XML grammars and languages. Acta Inf 38:115–125
15. Murata M, Lee D, Mani M, Kawaguchi K (2005) Taxonomy of XML schema languages using formal language theory. ACM TIT 5(4):660–704
16. Knuth DE (1965) On the translation of languages from left to right. Inf Control 8:607–639
17. Aho A, Lam M, Sethi R, Ullman J (2006) Compilers: principles, techniques and tools. Prentice-Hall, Englewoof Cliffs
18. Grune D, Jacobs C (2009) Parsing techniques: a practical guide, 2nd edn. Springer, London
19. Borsotti A, Breveglieri L, Crespi Reghizzi S, Morzenti A (2017) Fast deterministic parsers for transition networks. Acta Inf 1–28
20. Heilbrunner S (1979) On the definition of ELR(k) and ELL(k) grammars. Acta Inf 11:169–176
21. Chapman NP (1987) LR parsing: theory and practice. Cambridge University Press, Cambridge
22. Rosenkrantz DJ, Stearns RE (1970) Properties of deterministic top-down parsing. ic 17(3): 226–256
23. Rosenkrantz DJ, Stearns RE (1971) Top-down syntax analysis. Acta Inf 1:79–110
24. Wirth N (1975) Algorithms + data structures = programs. Prentice-Hall, Englewood Cliffs
25. Lewi J, De Vlaminck K, Huens J, Huybrechts M (1979) A programming methodology in compiler construction, I and II. North-Holland, Amsterdam
26. Breveglieri L, Crespi Reghizzi S, Morzenti A (2013) Parsing methods streamline, pp 1–64. CoRR abs/ arXiv:1309.7584
27. Beatty JC (1982) On the relationship between the LL(1) and LR(1) grammars. J ACM **29**(4):1007–1022
28. Quong R, Parr T (1995) ANTLR: A predicated-LL(k) parser generator. Softw Pract Exp 25:789–810
29. Floyd RW (1963) Syntactic analysis and operator precedence. J ACM 10(3):316–333
30. Crespi Reghizzi S, Mandrioli D, Martin DF (1978) Algebraic properties of operator precedence languages. Inf Control 37:(2):115–133
31. Barenghi A, Crespi Reghizzi S, Mandrioli D, Panella F, Pradella M (2015) Parallel parsing made practical. Sci Comput Prog 112:195–226
32. Crespi Reghizzi S, Mandrioli D (2012) Operator precedence and the visibly pushdown property. J Comput Syst Sci 78(6):1837–1867
33. Sippu S, Soisalon-Soininen E (1988) Parsing theory, volume 1: languages and parsing. Springer, Berlin

34. Sippu S, Soisalon-Soininen E (1990) Parsing theory, volume 2: LR(k) and LL(k). Springer, Berlin
35. Aho A, Ullman J (1972) The theory of parsing, translation, and compiling, volume 1: parsing. Prentice-Hall, Englewoof Cliffs
36. Fischer MJ (1969) Some properties of precedence languages. In: Fischer PC, Ginsburg S, Harrison MA (eds) Proceedings of the 1st ACM symposium on theory of computing, pp 181–190
37. Beatty JC (1980) Two iteration theorems for the LL(k) languages. Theor Comput Sci 12:193–228
38. Tomita M (1986) Efficient parsing for natural language: a fast algorithm for practical systems. Kluwer, Boston
39. Earley J (1970) An efficient context-free parsing algorithm. Commun ACM 13(2):94–102
40. Scott E (2008) SPPF-style parsing from Earley recognizers. Electr Notes Theor Comput Sci 203(2):53–67
41. Aycock J, Borsotti A (2009) Early action in an Earley parser. Acta Inf 46(8):549–559
42. Aycock J, Horspool R (2002) Practical Earley parsing. Comput J 45(6):620–630
43. Révész G (1991) Introduction to formal languages. Dover, New York
44. Dain JA (1994) A practical minimum distance method for syntax error handling. Comput Lang 20(4):239–252
45. Burke MG, Fisher GA (1987) A practical method for LR and LL syntactic error diagnosis. TOPLAS 9(2):164–197
46. Heering J, Klint P, Rekers J (1990) Incremental generation of parsers. IEEE TSE 16(12):1344–1351
47. Ghezzi C, Mandrioli D (1979) Incremental parsing. ACM TOPLAS 1(1):58–70
48. Larchevêque J (1995) Optimal incremental parsing. ACM TOPLAS 17(1):1–15



5 Translation Semantics and Static Analysis

5.1 Introduction

In addition to recognition and parsing, most language processing tasks perform some kind of transformation of the original sentence. For instance, a compiler translates a program from a high-level programming language, e.g., Java, to the machine code of some microprocessor. This chapter presents a progression of translation models and methods.

A translation is a function or more generally a mapping from the strings of the source language to the strings of the target language. As for string recognition, two approaches are possible. The generative point of view relies on two coupled grammars, termed a syntactic translation scheme, to generate a pair of strings that correspond to each other in the translation. The other approach uses a transducer, which is similar to a recognizer automaton but differs from it by its capability to emit a target string.

Such methods may be called purely syntactic translations. They extend and complete the language definition and the parsing methods of previous chapters, but we hasten to say they are not adequate for implementing the rather involved translations required for compilers. What is missing in the purely syntactic methods is the concern for the meaning or semantics of the language. For that, we shall present the attribute grammar model, which is a valuable software engineering approach for designing well-structured translators by taking advantage of the syntactic modularity of grammar rules.

We attempt to clarify the distinction between the *syntax* and *semantics* of a language. The etymology of the two terms says rather vaguely that the former has to do with the structure and the latter with the meaning of the message or phrase to be communicated. In linguistics, the two terms have often been taken as emblems pointing to forms and contents, respectively; yet this reference to the studies in the human sciences does not make the distinction any more precise or formal.

© Springer Nature Switzerland AG 2019

S. Crespi Reghizzi et al., *Formal Languages and Compilation*,
Texts in Computer Science, https://doi.org/10.1007/978-3-030-04879-2_5

In the case of computer languages, there is a sort of consensus on a demarcation line between syntactic and semantic methods. The first difference comes from the domains of the entities and operations that are permitted by syntax versus semantics. Syntax uses the concepts and operations of formal language theory and represents algorithms as automata. The entities are alphabets, strings and syntax trees; the operations are concatenation, morphisms on characters and strings, and tree construction primitives. On the negative side, the concepts of number and arithmetic operation (sum, product, etc.) are extraneous to syntax. On the other hand, in semantics the entities are not limited as in syntax: also numbers, and any type of data structures available to programmers (such as tables or linked lists), may be defined and used as needed by semantic algorithms. These can take advantage of the syntactic structure as a skeleton for orderly processing the data associated with language components.

The second difference is the higher computational complexity of the semantic algorithms with respect to the syntactic ones. We recall that the formal languages of concern for compilers belong almost exclusively to the regular and deterministic context-free families. String recognition, parsing and syntactic translation are typical syntactic tasks that can be performed in a linear time, i.e., in a time proportional to the length of the source text. But such very efficient algorithms fall short of all the controls required to check program correctness with respect to the language reference manual. For instance, with a parser it is impossible to check that an object used in a Java expression has been consistently defined in a declaration. As a matter of fact, such a control cannot be done in linear time. This and similar operations are performed by a compiler subsystem usually referred to as a semantic analyzer.

Continuing the comparison, the distinction between syntactic and semantic models is imposed by pragmatic considerations. In fact, it is well known from computation theory that any computable function, such as deciding whether a source string is a valid Java program, in principle can be realized by a Turing machine. This is for sure a syntactic formalism, since it operates just on strings and uses the basic operations of formal language theory. But, in practice, a Turing machine is too complicated to be programmed for any realistic problem, let alone compiler design.

Years of attempts at inventing grammars or automata that would allow some of the basic semantic controls to be performed by the parser have shown that the legibility and convenience of syntactic methods rapidly decay, as soon as the model goes beyond the context-free languages and enters the domain of the context-dependent languages (p. 105). In the state of the art, the practical syntactic methods are limited to the context-free domain, as we have argued on p. 110.

Chapter Outline

The word *translation* signifies a correspondence between two texts that have the same meaning, but are written in two different languages. Many translation cases occur with artificial languages: the compilation of a programming language into machine code; the transformation of an *HTML* document into the *PDF* format used for portable documents; etc. The given text and language are qualified as the *source*, and the other text and language are the *target*. We may also include under translation

the case of two texts in the same language: an example is the program transformation from, say, Java code to Java code, performed by certain software engineering tools.

In our presentation, the first and most abstract definition of translation to be considered will be a mapping between two formal languages. The second kind of translation is that obtained by applying local transformations to the source text, such as replacing a character with a string in accordance with a transliteration table. Then, two further, purely syntactic methods of defining a translation, will be presented: the *translation grammar* and the *translation regular expression*. Such translation models are also characterized by the abstract machines that compute them, respectively: the pushdown transducer, which can be implemented on top of a parsing algorithm; and the finite transducer, which is a finite automaton enriched with an output function.

We repeat that the purely syntactic translation models fall short of the requirements of compilation, since they cannot express various typical text transformations which are necessary. Nevertheless, they are important as a conceptual foundation and a scaffolding for the actual methods used in compilation. Moreover, they have another use as methods for abstracting from the concrete syntax in order to expose similarities between languages.

It is enlightening to show a structural analogy between the theories of the previous chapters and of the present one. At the level of set-theoretical definition the set of the sentences of the source language becomes the set of the matching pairs (source string and target string) of the translation relation. At the level of generative definition the language grammar becomes a translation grammar that generates pairs of source/target strings. Finally, at the level of operational definition the finite or pushdown automaton or parser that recognizes a language becomes a translator that computes the transformation. Such conceptual correspondences will clearly surface in this chapter.

The fifth and last conceptual model to be presented is the *syntax-directed semantic translation*, a semiformal approach based on the previous models. This makes a convenient engineering method for designing well-structured modular translators. Its presentation relies on attribute grammars, which consist of a combination of syntax rules and semantic functions.

Several typical examples will be presented. A lexical analyzer or scanner is specified by the addition of simple semantic attributes and functions to a finite transducer. Other important examples are: type checking in assignments and expressions; translation of conditional instructions into jumps; and semantics-directed parsing.

The last part of the chapter presents another central method used to compile programming languages, namely *static program analysis*. This analysis applies to executable programs rather than to generic technical languages. The flowchart or control-flow graph of the program to be analyzed is viewed as a finite automaton. Static analysis detects on this automaton various properties of the program, which are related to its correctness, or needed to perform program optimizations. This final topic completes the well-balanced exposition of the elementary compilation methods.

5.2 Translation Relation and Function

We introduce some notions from the mathematical theory of translations,[1] which suffice for the scope of the book. Let the *source* and *target* alphabets be denoted by Σ and Δ, respectively. A translation is a correspondence between source and target strings, which we formalize as a binary relation between the source and the target universal languages Σ^* and Δ^*; that is, a translation relation is a subset of the Cartesian product $\Sigma^* \times \Delta^*$.

A *translation relation* ρ is a set of pairs of strings (x, y), with $x \in \Sigma^*$ and $y \in \Delta^*$, in formula:

$$\rho = \{ (x, y), \ldots \} \subseteq \Sigma^* \times \Delta^*$$

We say that the target string y is the *image* or *translation* (or sometimes *destination*) of the source string x and that the two strings *correspond* to each other in the translation. Given a translation relation ρ, the *source language* L_1 and *target language* L_2 are, respectively, defined as the projections of the relation on the first and second component, as follows:

$$L_1 = \{ x \in \Sigma^* \mid \text{ there exist a string } y \text{ such that } (x, y) \in \rho \}$$
$$L_2 = \{ y \in \Delta^* \mid \text{ there exist a string } x \text{ such that } (x, y) \in \rho \}$$

Alternatively, a translation can be formalized by taking the set of all the images of a source string. Then a translation is modeled as a *function* τ:

$$\tau \colon \Sigma^* \to \wp(\Delta^*) \qquad \tau(x) = \{ y \in \Delta^* \mid (x, y) \in \rho \}$$

where symbol ρ is a translation relation.[2] Function τ maps each source string on the set of the corresponding images, that is, on a language.

Notice that the application of the translation function to every string of the source language L_1 produces a set of languages, and their union yields the target language L_2, as follows:

$$L_2 = \bigcup_{x \in L_1} \tau(x)$$

In general, a translation function is partially defined: for some strings over the source alphabet the function may be undefined. A simple expedient for making the function total is to posit that, where the application of function τ to string x is undefined, it is assigned the special value *error*.

A particular but practically important case occurs when every source string has no more than one image, and in this case the translation function is $\tau \colon \Sigma^* \to \Delta^*$.

[1] A rigorous presentation can be found in Berstel [1] and in Sakarovich [2].

[2] Remember that given a set X, symbol $\wp(X)$ denotes the power set of X, i.e., the set of all the subsets of X.

The *inverse translation* τ^{-1} is a function that maps a target string on the set of the corresponding source strings[3]:

$$\tau^{-1} : \Delta^* \to \wp\left(\Sigma^*\right) \qquad \tau^{-1}(y) = \left\{ x \in \Sigma^* \mid \ y \in \tau(x) \right\}$$

Depending on the mathematical properties of the function, the following cases arise for a translation:

total	every source string has one or more images
partial	one or more source strings do not have any image
single-valued	no string has two distinct images
multi-valued	one or more source strings have more than one image
injective	distinct source strings have distinct images, or differently stated, any target string corresponds to at most one source string; only in this case the inverse translation is single-valued
surjective	the image of the translation coincides with the range; i.e., every string over the target alphabet is the image of at least one source string; only in this case the inverse translation is total
bijective (or simply *one to one*)	the correspondence between the source and the target strings is both injective and surjective; i.e., it is one to one; only in this case the inverse translation is bijective as well

To illustrate, consider a high-level source program, say in Java, and its image in the code of a certain machine. Clearly the translation is total because any valid program can be compiled into machine code and any incorrect program has the value *error*, i.e., a diagnostic, for image. Such translation is multi-valued because usually the same Java statement admits several different machine code realizations. The translation is not injective because two source programs may have the same machine code image: just think of two `while` and `for` loops translated to the same code that uses conditional and unconditional jumps. The translation is not surjective since some machine programs that operate on special hardware registers cannot be expressed by Java programs.

On the other hand, if we consider a particular compiler from Java into machine code, the translation is totally defined, as before, and in addition it is single-valued,

[3]Given a translation relation ρ, the inverse translation relation ρ^{-1} is obtained by swapping the corresponding source and target strings, i.e., $\rho^{-1} = \{ (y, x) \mid \ (x, y) \in \rho \}$; it is equivalent to the inverse translation function τ^{-1}.

because the compiler chooses exactly one out of the many possible machine implementations of the source program. The translation is not necessarily injective (for the same reasons as above) and certainly it is not surjective, because a typical compiler does not use all the instructions of a machine.

A decompiler reconstructs a source program from a given machine program. Notice that this translation is not the reverse translation of the compilation, because compiler and decompiler are algorithms independently designed, and they are unlikely to make the same design decisions for their mappings. A trivial example: given a machine program $\tau(x)$ produced by the compiler τ, the decompiler δ will output a Java program $\delta(\tau(x))$ that almost certainly differs from program x with respect to the presence of blank spaces!

Since compilation is a mapping between two languages that are not finite, it cannot be specified by the exhaustive enumeration of the corresponding pairs of source/target strings. The chapter continues with a gradual presentation of the methods to specify and implement such infinite translations.

5.3 Transliteration

A naif way to transform a text is to apply a local mapping in each position of the source string. The simplest transformation is the transliteration or alphabetic homomorphism, introduced in Chap. 2 on p. 97. Each source character is transliterated to a target character or more generally to a string.

Let us read Example 2.89 on p. 97 anew. The translation defined by an alphabetic homomorphism is clearly single-valued, whereas the inverse translation may or may not be single-valued. In that example the little square $\square$ is the image of any Greek letter; hence the inverse translation is multi-valued:

$$h^{-1}(\square) = \{\, \alpha,\ \ldots,\ \omega \,\}$$

If the homomorphism erases a letter, i.e., maps the letter to the empty string, as it happens with characters start-text and end-text, the inverse translation is multi-valued because any string made of erasable characters can be inserted in any text position.

If the inverse function too is single-valued, the source/target mapping is a one-to-one or bijective function, and it is possible to reconstruct the source string from a given target string. This situation occurs when encryption is applied to a text. A historical example defined by transliteration is Julius Caesar's encryption method, which replaces each letter of the Latin alphabet, having position i, $1 \leq i \leq 26$, in the lexicographic ordering, by the letter in the position $(i + k) \mod 26$ (i.e., at alphabetic distance k), where constant k, $1 \leq k \leq 25$, is the ciphering secret key.

To finish, we stress that transliteration transforms a letter into another one without any regard to the occurrence context. It goes without saying that such a process falls short of the needs of compilation.

5.4 Purely Syntactic Translation

When the source language is defined by a grammar, a typical situation for programming languages, it is natural to consider a translation model where each syntactic component, i.e., a subtree, is individually mapped on a target subtree. The latter are then assembled into the target syntax tree that represents the translation. Such structural translation is now formalized as a mapping scheme, by relating the source and target grammar rules.

Definition 5.1 (*translation grammar*) A *translation grammar* $G_\tau = (V, \Sigma, \Delta, P, S)$ is a context-free grammar that has as terminal alphabet a set $C \subseteq \Sigma^* \times \Delta^*$ of pairs (u, v) of source/target strings, also written as a fraction $\frac{u}{v}$.

The translation relation ρ_G defined by grammar G_τ is the following:

$$\rho_{G_\tau} = \{ (x, y) \mid \exists z \in L(G_\tau) \wedge x = h_\Sigma(z) \wedge y = h_\Delta(z) \}$$

where $h_\Sigma : C \to \Sigma$ and $h_\Delta : C \to \Delta$ are the projections from the grammar terminal alphabet to the source and target alphabets, respectively. □

Intuitively a pair of corresponding source/target strings is obtained by taking a sentence z generated by G_τ and by projecting it on the two alphabets. Such a *translation* is termed *context-free* or algebraic.[4]

The *syntactic translation scheme* associated with the translation grammar is the set of pairs of source and target syntax rules, obtained by, respectively, canceling from the rules of G_τ the characters of the target alphabet or of the source alphabet. The set of source/target rules comprises the *source grammar* G_1 and the *target grammar* G_2 of the translation scheme. A translation grammar and a translation scheme are just notational variations of the same conceptual model. Example 5.2 shows a translation grammar and the equivalent translation scheme.

Example 5.2 (*translation grammar for string reversal*) Given string $a\,a\,b$, its translation is the mirror string $b\,a\,a$. The translation grammar G_τ for string reversal is as follows:

$$G_\tau : S \to \frac{a}{\varepsilon}\, S\, \frac{\varepsilon}{a} \mid \frac{b}{\varepsilon}\, S\, \frac{\varepsilon}{b} \mid \frac{\varepsilon}{\varepsilon}$$

[4] Another historical name for such translations is *simple syntax-directed translations*.

Equivalently, the translation grammar can be replaced by the following translation scheme (G_1, G_2):

source grammar G_1	*target grammar* G_2
$S \to a\ S$	$S \to S\ a$
$S \to b\ S$	$S \to S\ b$
$S \to \varepsilon$	$S \to \varepsilon$

The two columns list the source and target grammars, and each row contains two *corresponding rules*. For instance, the second row is obtained by the source/target projections h_Σ and h_Δ (see Definition 5.1), as follows:

$$h_\Sigma\left(S \to \frac{b}{\varepsilon}\ S\ \frac{\varepsilon}{b}\right) = S \to b\ S \qquad h_\Delta\left(S \to \frac{b}{\varepsilon}\ S\ \frac{\varepsilon}{b}\right) = S \to S\ b$$

To obtain a pair of strings that correspond in the translation relation, we construct a derivation, like:

$$S \Rightarrow \frac{a}{\varepsilon}\ S\ \frac{\varepsilon}{a} \Rightarrow \frac{a}{\varepsilon}\ \frac{a}{\varepsilon}\ S\ \frac{\varepsilon}{a}\ \frac{\varepsilon}{a} \Rightarrow \frac{a}{\varepsilon}\ \frac{a}{\varepsilon}\ \frac{b}{\varepsilon}\ S\ \frac{\varepsilon}{b}\ \frac{\varepsilon}{a}\ \frac{\varepsilon}{a} \Rightarrow \frac{a}{\varepsilon}\ \frac{a}{\varepsilon}\ \frac{b}{\varepsilon}\ \frac{\varepsilon}{\varepsilon}\ \frac{\varepsilon}{b}\ \frac{\varepsilon}{a}\ \frac{\varepsilon}{a} = z$$

and then we project the sentence z of $L(G_\tau)$ on the two alphabets, obtaining:

$$h_\Sigma(z) = a\,a\,b \qquad h_\Delta(z) = b\,a\,a$$

Differently, using the translation scheme, we generate a source string by a derivation of G_1 and its image by a derivation of G_2, where we pay attention to use two corresponding rules at each step. □

The reader may have noticed that the preceding translation grammar G_τ of Example 5.2 is almost identical to the familiar grammar of even-length palindromes. By marking with a prime the characters in the second half of a string, the palindrome grammar becomes the following grammar:

$$G_p\colon S \to a\,S\,a' \mid b\,S\,b' \mid \varepsilon \tag{5.1}$$

Recoding strings $\frac{a}{\varepsilon}$ as a, $\frac{b}{\varepsilon}$ as b, $\frac{\varepsilon}{a}$ as a' and $\frac{\varepsilon}{b}$ as b', the two grammars G_τ and G_p coincide. This remark leads to the next property.

Property 5.3 (context-free language and translation) *The following conditions are equivalent:*

1. *The translation relation* $\rho_{G_\tau} \subseteq \Sigma^* \times \Delta^*$ *is defined by a translation grammar* G_τ.

2. *There exists an alphabet Ω, a context-free language L over alphabet Ω and two alphabetic homomorphisms (transliterations) $h_1 : \Omega \to \Sigma \cup \{ \varepsilon \}$ and $h_2 : \Omega \to \Delta \cup \{ \varepsilon \}$, such that:*

$$\rho_{G_\tau} = \{ (h_1 (z),\ h_2 (z)) \mid z \in L \}$$ □

Notice that the two homomorphisms may also erase a character of alphabet Ω. We reuse the translation from a string to its reversal, to illustrate the second statement of Property 5.3.

Example 5.4 (*string reversal, continued from Example* 5.2) From part 2 of Property 5.3, to express the translation ρ_{G_τ} of Example 5.2 we use the alphabet $\Omega = \{ a,\ b,\ a',\ b' \}$ and the following context-free language L

$$L = \left\{ u\,(u^R)' \mid u \in (a \mid b)^* \right\} = \{ \varepsilon,\ a\,a',\ \ldots,\ a\,b\,b'\,b'\,a',\ \ldots \}$$

where string $(v)'$ is the primed copy of string v. Notice that language L is generated by the grammar at line (5.1) above. The homomorphisms h_1 and h_2 are the following:

Σ	h_1	h_2
a	a	ε
b	b	ε
a'	ε	a
b'	ε	b

Then, the string $a\,b\,b'\,a' \in L$ is transliterated to the two strings below:

$$\left(h_1 \left(a\,b\,b'\,a' \right),\ h_2 \left(a\,b\,b'\,a' \right) \right) = (\,a\,b,\ b\,a\,)$$

that belong to the translation relation ρ_{G_τ}. □

5.4.1 Infix and Polish Notation

A relevant application of context-free translation is to convert back and forth between various representations of arithmetic (or logical) expressions, which differ by the relative positions of their operands and signs, and by the use of parentheses or other delimiters.

We define the *degree of an operator* as the number of arguments or operands it may have. The degree can be fixed or variable, and in the latter case the operator is called *variadic*. Operators of degree two are very common and are called *binary* operators. For instance, the comparisons, such as equality "=" and difference "≠", are binary operators. Arithmetic addition has a degree ≥ 2, but in a typical machine

language the add instruction is just binary, since it adds two registers. Moreover, since addition is usually assumed to satisfy the associative property, a many-operand addition can be decomposed into a series of binary additions to be performed, say, from left to right.

Arithmetic subtraction provides an example of non-commutative binary operation, whereas a change of sign (as in $-x$) is an example of *unary* operation. If the same symbol "−" is used to denote both operations, the operator becomes variadic with degree one or two.

Examining now the relative positions of operators and operands, we have the following cases. An *operator* is *prefix* if it precedes its arguments, and it is *postfix* if it follows them. A binary operator is *infix* if it is placed between its arguments.

One may also generalize the property of being infix, to operators with higher degree. An operator of degree $n \geq 2$ is *mixfix* if its representation can be segmented into $n + 1$ parts, as follows:

$$o_0\ arg_1\ o_1\ arg_2\ \ldots\ o_{n-1}\ arg_n\ o_n$$

that is, if the argument list starts with an opening mark o_0, then it is followed by $n - 1$ (possibly different) separators o_i and terminates with a closing mark o_n. Sometimes the opening and closing marks are missing.

For instance, the conditional operator if of many programming languages is mixfix with degree two or three if the else clause is present:

$$\mathsf{if}\ arg_1\ \mathsf{then}\ arg_2\ \left[\ \mathsf{else}\ arg_3\ \right]$$

Because of the varying degree, this representation is ambiguous if the second argument can be in turn a conditional operator (as seen on p. 60). To remove ambiguity (see also Sect. 2.5.11.6 on p. 60), in certain languages the conditional construct is terminated by a closing mark end_if, e.g., in *ADA*.

In machine language the binary conditional operator is usually represented in prefix form by an instruction such as:

$$\mathsf{jump_if_false}\ arg_1,\ arg_2$$

Actually, in machine language operations, every machine instruction is of the prefix type as it begins with an operation code. The degree is fixed for each code, and it typically ranges from zero, e.g., in a nop instruction, to three, e.g., in a three-register addition instruction add r1, r2, r3.

A representation is called *polish*[5] if it does not use parentheses and if all the operators are either prefix or postfix (not mixed). The elementary grammar of polish expressions is printed on p. 56.

[5] From the nationality of the logician Jan Lukasiewicz, who proposed its use for compacting and normalizing logical formulas.

We illustrate a frequent transformation performed by compilers to eliminate parentheses, by converting an arithmetic expression from infix to polish notation. For simplicity, we prefer to use disjoint source/target alphabets, in order to avoid the need of fractions in the rules of the translation grammar.

Example 5.5 (*from infix to prefix operators*) The source language comprises arithmetic expressions with (infix) addition and multiplication, parentheses and the terminal i that denotes a variable identifier. The translation is to polish prefix: operators are moved to the prefix position, parentheses disappear, and identifiers are transcribed to i'.
Source and target alphabets:

$$\Sigma = \{ +,\ \times,\ \text{`(',}\ \text{`)',}\ i \} \qquad\qquad \Delta = \{ \mathsf{add},\ \mathsf{mult},\ i' \}$$

Translation grammar (axiom E):

$$G_\tau \begin{cases} E \ \to\ \mathsf{add}\ T\ +\ E \ \mid\ T \\ T \ \to\ \mathsf{mult}\ F\ \times\ T \ \mid\ F \\ F \ \to\ \text{`(' } E \text{ `)'} \ \mid\ i\, i' \end{cases}$$

Notice that rule $E \to \mathsf{add}\ T\ +\ E$ abridges rule $E \to \frac{\varepsilon}{\mathsf{add}}\ T\ \frac{+}{\varepsilon}\ E$, at no danger of confusion because the source/target alphabets are disjoint.

The equivalent translation scheme follows:

source grammar G_1	*target grammar* G_2
$E \to T\ +\ E$	$E \to \mathsf{add}\ T\ E$
$E \to T$	$E \to T$
$T \to F\ \times\ T$	$T \to \mathsf{mult}\ F\ T$
$T \to F$	$T \to F$
$F \to \text{`(' } E \text{ `)'}$	$F \to E$
$F \to i$	$F \to i'$

(5.2)

An example of translation is shown in the combined source/target syntax tree in Fig. 5.1. Imagine to erase the dashed part of the tree: then the source syntax tree of expression $(i + i) \times i$ shows up, as the parser would construct it by using the source grammar G_1. Conversely, erasing from the tree the leaves of the source alphabet and their edges, we would see the tree generated by the target grammar for the image string $\mathsf{mult}\ \mathsf{add}\ i'\ i'\ i'$. □

Construction of Target Syntax Tree

We observe that in a translation scheme, such as (5.2) above, derived from a translation grammar according to Definition 5.1, the following conditions are always met:

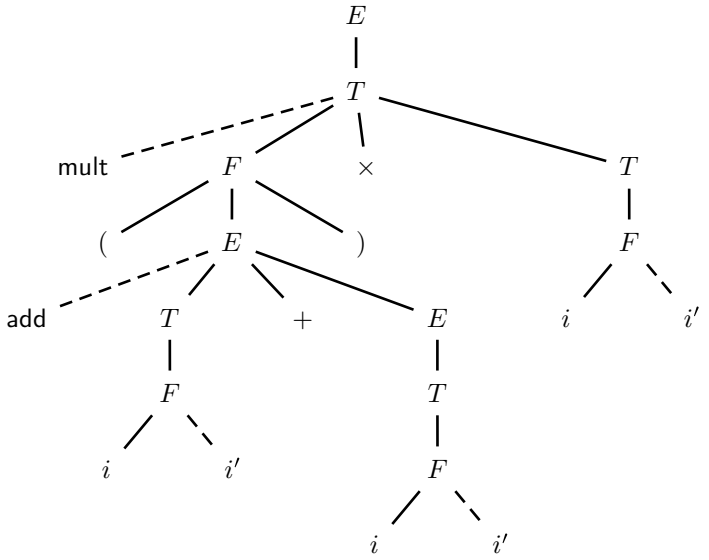


Fig. 5.1 Source/target tree generated by the translation grammar of Example 5.5

1. The rules of the two grammars are in one-to-one correspondence.
2. Any two corresponding rules have identical left parts, and their nonterminal symbols occur in the same order in their right parts.

Given a translation grammar, we explain how to compute the image of a source sentence x. First, we parse the string using source grammar G_1 and we construct the source syntax tree t_x of x, which is unique if the sentence is unambiguous. Then, we visit tree t_x, say, in left-to-right pre-order.

At each visited node, if the current node of tree t_x is the left part symbol of a certain rule of grammar G_1, then we apply the corresponding rule of target grammar G_2 and thus we append some children nodes to the target tree. At the end of visit, the target tree is complete.

Abstract Syntax Tree

Syntactic translations are a convenient method for trimming and transforming source syntax trees, in order to remove the elements that are irrelevant to the later stages of compilation, and to reformat the tree as needed. This transformation is an instance of language abstraction (p. 26). A case has already been considered: the elimination of parentheses from arithmetic expressions. One can easily imagine other cases, such as the elimination or recoding of separators between the elements of a list, or of the mixfix keywords of the conditional instructions like if …then …else …end_if. The result of such transformations is often called an *abstract syntax tree*.

5.4.2 Ambiguity of Source Grammar and Translation

We have already observed that most applications are concerned with single-valued translations. However, if the source grammar is ambiguous, a sentence admits two different syntax trees, each one corresponding to a target syntax tree. Therefore the sentence will have two images, generally different.

The next example (Example 5.6) shows the case of a translation that is multi-valued because of the ambiguity of its source component.

Example 5.6 (*redundant parentheses*) A case of multi-valued translation is the conversion from prefix polish to infix notation, the latter with parentheses. It is immediate to write the translation scheme, as it is the inverse translation of Example 5.5 on p. 393. Therefore it suffices to interchange source and target grammars, to obtain:

source grammar G_1	*target grammar* G_2
$E \to \mathsf{add}\ T\ E$	$E \to T\ +\ E$
$E \to T$	$E \to T$
$T \to \mathsf{mult}\ F\ T$	$T \to F\ \times\ T$
$T \to F$	$T \to F$
$F \to E$	$F \to$ ‘(’ E ‘)’
$F \to i'$	$F \to i$

Here the source grammar G_1 has an ambiguity of unbounded degree, which comes from the following circular derivation (made of copy rules):

$$E \Rightarrow T \Rightarrow F \Rightarrow E$$

For instance, the following derivations of source string i' are possible:

$$E \Rightarrow T \Rightarrow F \Rightarrow i' \qquad E \Rightarrow T \Rightarrow F \Rightarrow E \Rightarrow T \Rightarrow F \Rightarrow i' \qquad \ldots$$

and each of them has a distinct image:

$$E \Rightarrow T \Rightarrow F \Rightarrow i \qquad E \Rightarrow T \Rightarrow F \Rightarrow (\ E\ ) \Rightarrow (\ T\ ) \Rightarrow (\ F\ ) \Rightarrow (\ i\ ) \qquad \ldots$$

The fact is scarcely surprising since in the conversion from prefix to infix, one can insert as many pairs of parentheses as he wishes, but the translation scheme does not prescribe their number and allows the insertion of redundant parentheses. □

On the other hand, suppose now that the source grammar is unambiguous, so that each sentence has only one syntax tree. Yet, it may happen that the translation is

multi-valued, if in the translation scheme different target rules correspond to the same source rule. An example is the next translation grammar:

$$S \to \frac{a}{b}\ S \qquad S \to \frac{a}{c}\ S \qquad S \to \frac{a}{d}$$

where the source grammar $S \to a\ S \mid a$ is unambiguous, yet the translation $\tau\,(a\,a) = \{\, b\,d,\ c\,d \,\}$ is not single-valued, because the first two rules of the translation grammar correspond to the same source rule $S \to a\ S$.

The next statement (Property 5.7) gives sufficient conditions for avoiding ambiguity in a translation grammar.

Property 5.7 (unambiguity conditions for translation) *Let* $G_\tau = (\,G_1,\ G_2\,)$ *be a translation grammar such that:*

1. *The source grammar* G_1 *is unambiguous.*
2. *No two rules of the target grammar* G_2 *correspond to the same rule of* G_1.

Then the translation specified by grammar G_τ *is single-valued and defines a translation function.* □

In the previous discussion on translation ambiguity, we have considered the ambiguity of the source and target grammars separately, but not the ambiguity of the translation grammar G_τ itself, because it is not relevant to ensure the single-valued translation property. The next example (Example 5.8) shows that a translation grammar, though unambiguous, may have an ambiguous source grammar which causes the translation to be multi-valued.

Example 5.8 (*end-mark in conditional instruction*) The translation mandatorily places an end-mark end_if after the conditional instruction if … then …[else]. The translation grammar below:

$$S \to \frac{\mathsf{if}\ c\ \mathsf{then}}{\mathsf{if}\ c\ \mathsf{then}}\ S\ \frac{\varepsilon}{\mathsf{end_if}} \mid \frac{\mathsf{if}\ c\ \mathsf{then}}{\mathsf{if}\ c\ \mathsf{then}}\ S\ \frac{\mathsf{else}}{\mathsf{else}}\ S\ \frac{\varepsilon}{\mathsf{end_if}} \mid a$$

is unambiguous. Yet, the underlying source grammar, which defines the conditional instruction without end-mark, is a typical case of ambiguity (p. 60). The translation produces two images of the same source string, as follows:

$$\mathsf{if}\ c\ \mathsf{then}\ \mathsf{if}\ c\ \mathsf{then}\ a \overset{\mathsf{end_if}}{\downarrow}\ \mathsf{else}\ a \overset{\mathsf{end_if}}{\downarrow} \underset{\mathsf{end_if}\ \mathsf{end_if}}{\uparrow}$$

which are obtained by inserting the end-marks in the positions indicated by the vertical darts either over or under the line. □

Compiler designers ought to pay attention to avoid translation grammars that make the translation multi-valued. Parser construction tools help because they routinely check the source grammar for determinism, which excludes ambiguity.

5.4.3 Translation Grammar and Pushdown Transducer

We shift focus from translation grammars to the abstract machines, called transducers or input–output automata, that compute such translations. Much as the recognizer of a context-free language, the transducer implementing a translation grammar needs a pushdown or *LIFO* store.

A *pushdown transducer* also known as *IO-automaton* is like a pushdown automaton (defined on p. 205), enriched with the capability to output zero or more characters at each move. More precisely, eight items have to be specified to define such a machine:

Q	set of states
Σ	source alphabet
Γ	pushdown stack alphabet
Δ	target alphabet
δ	state transition and output function
$q_0 \in Q$	initial state
$Z_0 \in \Gamma$	initial symbol on the stack
$F \subseteq Q$	set of final states

The function δ is defined in the domain $Q \times (\Sigma \cup \{ \varepsilon \}) \times \Gamma$ and has the set $Q \times \Gamma^* \times \Delta^*$ as range.[6] The meaning of the function is the following: if the current state, input character and stack top, respectively, are q', a and Z, and if it holds $\delta (q', a, Z) = \big(q'', \gamma, y \big)$, then the machine reads character a from the input and symbol Z from the stack top, enters the next state q'' and writes sequence γ on top of the stack and character y to the output. If the automaton recognizes the source string by empty stack, then the set of final states coincides with Q. The meaning of function δ is schematized in Fig. 5.2. The *subjacent automaton* of the translator is obtained by erasing the target alphabet symbols and the output string actions from the definitions.

The formalization of the translation computed by a pushdown translator follows the same lines as the definition of acceptance by the subjacent pushdown automaton,

[6] Alternatively, it is possible to specify the output by a separate output function, as we did for the *BSP* finite-state transducer in Definition 3.54 on p. 178. We skip the formalization of the case of nondeterministic transducer, just saying that the range of function δ would become the powerset $\wp$ of the preceding Cartesian product.

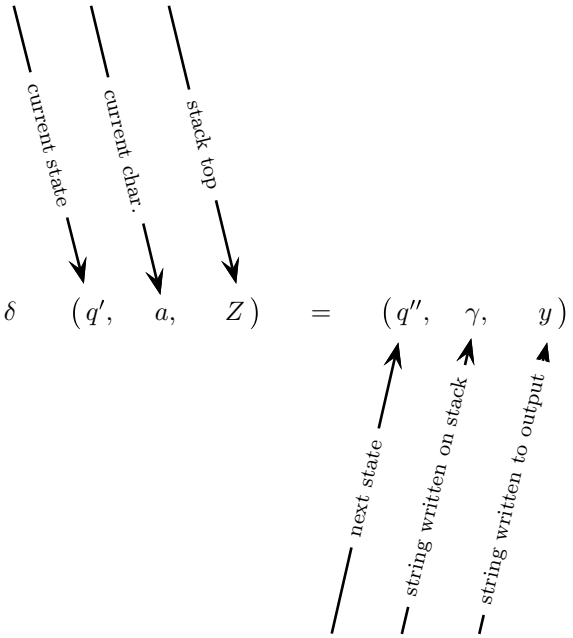


Fig. 5.2 Scheme of the move of a pushdown transducer

as stated in Sect. 4.2.2 on p. 205. The instantaneous configuration of the pushdown transducer is defined as a 4-tuple $(q,\ y,\ \eta,\ z) \in (Q \times \Sigma^* \times \Gamma^* \times \Delta^*)$, including:

q	current state
y	remaining portion (suffix) of the source string x to be read
η	stack contents
z	string written to the output tape, up to the current configuration

When a move is executed, a transition from a configuration to the next one occurs, denoted as $(q,\ x,\ \eta,\ w) \mapsto (p,\ y,\ \lambda,\ z)$. A computation is a chain of zero or more transitions, denoted by $\overset{*}{\mapsto}$. Depending on the kind of the performed move (reading or spontaneous), the following two kinds of transition are possible.

current conf.	*next conf.*	*applied move*
$(q,\ a\,x,\ \eta\,Z,\ z)$	$(p,\ x,\ \eta\,\gamma,\ z\,y)$	reading move $\delta(q,\ a,\ Z) = (p,\ \gamma,\ y)$
$(q,\ a\,x,\ \eta\,Z,\ z)$	$(p,\ a\,x,\ \eta\,\gamma,\ z\,y)$	spontaneous move $\delta(q,\ \varepsilon,\ Z) = (p,\ \gamma,\ y)$

Initial configuration and string acceptance are defined as in Sect. 4.2.2, and the computed translation τ is defined as follows (notice that acceptance by final state is assumed):

$$\tau(x) = z \quad \Longleftrightarrow \quad (q_0, x, Z_0, \varepsilon) \overset{*}{\mapsto} (q, \varepsilon, \lambda, z) \text{ with } q \in F \text{ and } \lambda \in \Gamma^*$$

5.4.3.1 From Translation Grammar to Pushdown Transducer

Translation schemes and pushdown transducers are two ways of representing language transformations: the former is a generative model suitable for specification, and the latter is procedural and helps in compilation. Their equivalence is stated in the next property.

Property 5.9 (equivalence of translation grammar and pushdown transducer) *A translation relation is defined by a translation grammar or scheme if, and only if, it is computed by a (nondeterministic) pushdown transducer.* □

For brevity, we only describe the conversion from a grammar to a transducer, because the other direction is less pertinent to compilation.

Consider a translation grammar G_τ. A first way of deriving the equivalent pushdown translator T is to apply essentially the same algorithm that we used to construct the pushdown recognizer of language $L(G_\tau)$ (Algorithm 4.1 on p. 202). Then, this machine is transformed into a transducer through a small change of the operations on the target characters. After pushing a target symbol onto the stack, when that symbol surfaces anew on the stack top, it is written to the output; but a target symbol is not matched against the current input character, unlike source characters.

Normalization of Translation Rules

To simplify the construction, it helps to reorganize without loss of generality the source and target strings that occur in a grammar rule, in such a way that the initial character is a source one, where possible. More precisely, we make the following hypotheses in the form of the source/target pairs $\frac{u}{v}$ that occur in the rules,[7] where $u \in \Sigma^*$ and $v \in \Delta^*$:

1. For any pair $\frac{u}{v}$ it holds $|u| \leq 1$; i.e., the source u is a single character $a \in \Sigma$ or the empty string. Clearly this is not a limitation because if the pair $\frac{a_1 a_2}{v}$ occurs in a rule, it can be replaced by the pairs $\frac{a_1}{v}\,\frac{a_2}{\varepsilon}$ without affecting the translation.
2. No rule may contain the following substrings:

$$\frac{\varepsilon}{v_1}\,\frac{a}{v_2} \quad \text{or} \quad \frac{\varepsilon}{v_1}\,\frac{\varepsilon}{v_2} \qquad \text{where } v_1, v_2 \in \Delta^*$$

[7] For completeness, we mention that in some publications on formal languages, the source/target fractions $\frac{u}{v}$ in the grammar rules are denoted as $u \{ v \}$, e.g., $X \to \alpha u \{ v \} \beta$ instead of $X \to \alpha \frac{u}{v} \beta$; but such a notation is not used in this book.

Should such combinations be present in a rule, they can be respectively replaced by the equivalent pairs $\frac{a}{v_1\, v_2}$ or $\frac{\varepsilon}{v_1\, v_2}$.

We are ready to describe the correspondence between the rules of grammar $G_\tau = (V,\ \Sigma,\ \Delta,\ P,\ S)$ and the moves of the so-called *predictive* transducers.

Algorithm 5.10 (*construction of the (nondeterministic) predictive pushdown transducer*) Let C be the set of the pairs of type $\frac{\varepsilon}{v}$ with $v \in \Delta^+$, and of type $\frac{b}{w}$ with $b \in \Sigma$ and $w \in \Delta^*$, occurring in some grammar rule. The rules for constructing the transducer moves are in Table 5.1. Rows 1, 2, 3, 4 and 5 apply when the stack top is a nonterminal symbol. In case 2 the right part begins with a source terminal and the move is conditioned by its presence in the input. Rows 1, 3, 4 and 5 give rise to spontaneous moves, which do not shift the reading head. Rows 6 and 7 apply when a pair surfaces on top of stack. If the pair contains a source character (row 7), it must coincide with the current input character; if it contains a target string (rows 6 and 7), the latter is output. Initially, the stack contains the axiom S, and the reading head is positioned on the first character of the source string. At each step, the automaton (nondeterministically) chooses an applicable rule and executes the corresponding move. Finally, row 8 accepts the string if the stack is empty and the current character marks the end of text. □

Notice the automaton does not make use of states; i.e., the stack is the only memory used. As we did for recognizers, we will later enrich the machine with states in order to obtain a more efficient deterministic algorithm.

The next example (Example 5.11) shows a simple translation computed by a nondeterministic pushdown transducer.

Example 5.11 (*nondeterministic pushdown transducer*) Consider the source language L below:

$$L = \left\{\, a^* a^m b^m \mid \ m > 0 \,\right\}$$

and define the following translation τ on L:

$$\tau\,(a^k\, a^m\, b^m) = d^m\, c^k \qquad \text{where } k \geq 0 \text{ and } m > 0$$

The translation τ first changes letter b to letter d, and then it transcribes to letter c any letter a that exceeds the number of b's. The moves of the transducer are listed in Table 5.2 next to the corresponding rules of the translation grammar. The choice of moves 1 or 2 in Table 5.2 is not deterministic, and so is the choice of moves 3 or 4. Move 5 outputs a target character that had been pushed by move 1. Move 6 just scans an input character, and move 7 is for acceptance. The subjacent pushdown automaton is not deterministic. □

The following example (Example 5.12) shows that not all context-free translations can be computed by a pushdown transducer of the deterministic kind. A similar property holding for regular translations and finite transducers will be illustrated in Sect. 5.5.2.

Table 5.1 Correspondence between translation grammar rules and pushdown translator moves (if $n = 0$, then the sequence $A_1 \ldots A_n$ is absent)

#	Rule	Move	Comment
1	$A \to \frac{\varepsilon}{v}\, B\, A_1 \ldots A_n$ $n \geq 0$ $v \in \Delta^+ \quad B \in V$ $A_i \in (C \cup V)$	If $top = A$ then write (v); pop; push $(A_n \ldots A_1\, B)$	Emit the target string v and push on stack the prediction string $B\, A_1 \ldots A_n$
2	$A \to \frac{b}{w}\, A_1 \ldots A_n$ $n \geq 0$ $b \in \Sigma \quad w \in \Delta^*$ $A_i \in (C \cup V)$	If $cc = b \wedge top = A$ then write (w); pop; push $(A_n \ldots A_1)$; advance the reading head	Char b was the next expected and has been read; emit the target string w; push the prediction string $A_1 \ldots A_n$
3	$A \to B\, A_1 \ldots A_n$ $n \geq 0 \quad B \in V$ $A_i \in (C \cup V)$	If $top = A$ then pop; push $(A_n \ldots A_1\, B)$	Push the prediction string $B\, A_1 \ldots A_n$
4	$A \to \frac{\varepsilon}{v} \quad v \in \Delta^+$	If $top = A$ then write (v); pop	Emit the target string v
5	$A \to \varepsilon$	If $top = A$ then pop	
6	For every pair $\frac{\varepsilon}{v} \in C$	If $top = \frac{\varepsilon}{v}$ then write (v); pop	The past prediction $\frac{\varepsilon}{v}$ is now completed by writing v
7	For every pair $\frac{b}{w} \in C$	If $cc = b \wedge top = \frac{b}{w}$ then write (w); pop; advance the reading head	The past prediction $\frac{b}{w}$ is now completed by reading b and writing w
8	- - - - - - - -	If $cc = \dashv \wedge$ stack is empty then accept; halt	The source string has been entirely scanned and no goal is present in the stack

Example 5.12 (*context-free nondeterministic translation*) The translation function τ below:

$$\tau(u) = u^R\, u \qquad \text{where } u \in \{\, a,\ b \,\}^*$$

maps every string into a reversed copy followed by the same string. Function τ is easily specified by the following scheme:

translation gram. G_τ	*source gram.* G_1	*target gram.* G_2
$S \to \frac{\varepsilon}{a}\, S\, \frac{a}{a}$	$S \to S\, a$	$S \to a\, S\, a$
$S \to \frac{\varepsilon}{b}\, S\, \frac{b}{b}$	$S \to S\, b$	$S \to b\, S\, b$
$S \to \frac{\varepsilon}{\varepsilon}$	$S \to \varepsilon$	$S \to \varepsilon$

Table 5.2 Moves of a nondeterministic pushdown transducer

#	Rule	Move
1	$S \to \frac{a}{\varepsilon}\ S\ \frac{\varepsilon}{c}$	If $cc = a\ \wedge\ top = S$, then pop; push $(\frac{\varepsilon}{c}\ S)$; advance the reading head
2	$S \to A$	If $top = S$, then pop; push (A)
3	$A \to \frac{a}{d}\ A\ \frac{b}{\varepsilon}$	If $cc = a\ \wedge\ top = A$, then pop; write (d); push $(\frac{b}{\varepsilon}\ A)$; advance the reading head
4	$A \to \frac{a}{d}\ \frac{b}{\varepsilon}$	If $cc = a\ \wedge\ top = A$, then pop; write (d); push $(\frac{b}{\varepsilon})$; advance the reading head
5	- - -	If $top = \frac{\varepsilon}{c}$, then pop; write (c)
6	- - -	If $cc = b\ \wedge\ top = \frac{b}{\varepsilon}$, then pop; advance the reading head
7	- - -	If $cc = \dashv\ \wedge$ stack is empty, then accept; halt

The translation cannot be deterministically computed by a pushdown machine. The reason is that such a machine should output the mirror copy of the input before the direct copy of the input.[8] The only way to reverse a string by means of a pushdown device is to store the string in the stack and then to pop and output each character. But popping destroys the stored string and makes it unavailable for later copying to the output. □

Nondeterministic algorithms, such as in Example 5.11, are used infrequently in compilation. In the next section we develop translator construction methods suitable for use in combination with the widespread deterministic parsers.

5.4.4 Syntax Analysis with Online Translation

Given a context-free translation scheme, the construction of Algorithm 5.10 produces a pushdown transducer that is often nondeterministic and generally unsuitable for use in a compiler. To construct an efficient and well-engineered translator, it is convenient to resume from the point reached in Chap. 4 with the construction of deterministic parsers, and to enrich them with output actions. Given a context-free translation grammar or scheme, we make the assumption that the source grammar is suitable for deterministic parsing. In this case, to compute the image of the source string, the parser emits the translation piece by piece, as it completes the parsing of a syntactic subtree.

[8] For a formal proof see [3,4].

We know that bottom-up and top-down parsers differ in the construction order of the syntax tree. A question to be clarified is how the order interferes with the possibility of correctly producing the translation. We anticipate the main result: the top-down parsing order is fully compatible with the translations defined by translation grammars, whereas the bottom-up order places some restrictions on the translation.

We recall there are two techniques for building efficient top-down deterministic parsers: as pushdown automata and as recursive descent procedures. Both techniques are simple to extend toward the construction of translators. Our initial example is the translation that, for a given input string, outputs its leftmost derivation represented as a list of source grammar rules. The following approach works for any $LL(k)$ grammar, and we illustrate it with the grammar of a Dyck language.

Example 5.13 (*leftmost derivation of Dyck language*) The source grammar G_1 (below left) has the following rules, numbered for reference:

$$G_1 \begin{cases} 1: \ S \to a\, S\, a'\, S \\ 2: \ S \to b\, S\, b'\, S \\ 3: \ S \to \varepsilon \end{cases} \qquad G_t \begin{cases} S \to \frac{a}{1}\, S\, a'\, S \\ S \to \frac{b}{2}\, S\, b'\, S \\ S \to \frac{\varepsilon}{3} \end{cases}$$

The target alphabet is $\{ 1,\ 2,\ 3 \}$, and each rule of the translation grammar G_t (above right) just plugs the rule label in the first position. The translation thus computed, for a string, say, $b\, a\, a'\, b' \in L(G_1)$, is 2 1 3 3 3, which is the sequence of rules in the order they are applied in the leftmost derivation $S \overset{+}{\Rightarrow} b\, a\, a'\, b'$. Clearly, the *DPDA* that implements the top-down parser can be enriched with a write instruction that emits the rule label, as soon as the current rule has been selected. Compare also with the similar example in Chap. 4 on p. 311.

5.4.5 Top-Down Deterministic Translation by Recursive Procedures

We next focus on the approach based on the recursive descent translator. In order to streamline the design of syntactic translation algorithms for grammars extended with regular expressions, it is convenient to represent the source grammar G_1 by means of a recursive network of finite machines, as we did in Chap. 4. Assuming the source grammar is $ELL(k)$ with $k = 1$ or larger, we recall the organization of the recursive descent parser (p. 313). For each nonterminal symbol, a procedure has the task of recognizing the (sub)strings derived from it. The procedure body blueprint is identical to the state-transition graph of the corresponding machine in the network that represents the grammar. For computing the translation, we simply insert a write transition in the machine and a corresponding write instruction in the procedure body.

An example (Example 5.14) should be enough to explain such a straightforward modification of a recursive descent parser.

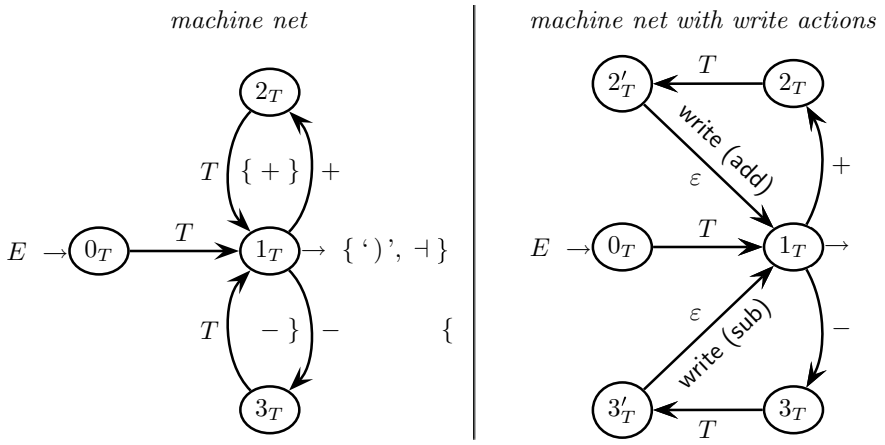


Fig. 5.3 Machine M_E that represents the *EBNF* axiomatic rule $E \to T\,(\,+\,T\;\mid\;-\,T\,)^*$ of the source grammar in Example 5.14 (left), and the same machine M_E augmented with write actions and states where necessary (right)

Example 5.14 (*recursive descent translator from infix to postfix*) The source language consists of arithmetic (infix) expressions with two levels of operators and with parentheses, and the translation converts such expressions to the postfix polish notation as exemplified below:

$$v \times (v + v) \qquad \Rightarrow \qquad v\ v\ v\ \textsf{add}\ \textsf{mult}$$

The source language is defined by the extended *BNF* grammar G_1 below (axiom E):

$$G_1 \begin{cases} E \to T\,(\,+\,T\;\mid\;-\,T\,)^* \\ T \to F\,(\,\times\,F\;\mid\;\div\,F\,)^* \\ F \to v\;\mid\;\text{‘(’}\;E\;\text{‘)’} \end{cases}$$

Figure 5.3 (left) represents the machine for nonterminal E: the machine meets the *ELL* (1) condition (p. 306), and the relevant guide sets are written in braces.

The recursive descent procedure for nonterminal E is shown in Fig. 5.4 (left); we recall that function *next* returns the current input token; for simplicity, the check of error cases is omitted. Notice that the procedure could be refined so that at loop exit it checks if the current character is in the guide set { ‘) ’, ⊣ }, and thus anticipates error detection. Similar procedures for nonterminals T and F can be derived.

In Fig. 5.3 (right) we add to the machine for nonterminal E, a few suitable states and transitions that encode the write actions to output the operators in postfix position. From this augmented machine, the next recursive procedure including write instructions is easily derived and is also shown in Fig. 5.4 (right). For brevity we omit the similar procedures for nonterminals T and F.

partial rec. des. parser

```
procedure E
    call T
    while cc ∈ { '+', '−' } do
        case cc of
        '+' :begin
                cc := next
                call T
            end
        '−' :begin
                cc := next
                call T
            end
        end case
    end while
end procedure
```

partial parser with write instructions

```
procedure E
    call T
    while cc ∈ { '+', '−' } do
        case cc of
        '+' :begin
                cc := next
                call T
                write ("add")
            end
        '−' :begin
                cc := next
                call T
                write ("sub")
            end
        end case
    end while
end procedure
```

Fig. 5.4 Recursive descent syntactic procedures for nonterminal E, without (left) and with (right) write instructions, which are framed

The machines and procedures that compute the complete translation correspond to the following extended *BNF* translation grammar G_τ (axiom E):

$$G_\tau \begin{cases} E \rightarrow T \left(\frac{+}{\varepsilon}\, T\, \frac{\varepsilon}{\mathsf{add}} \mid \frac{-}{\varepsilon}\, T\, \frac{\varepsilon}{\mathsf{sub}} \right)^* \\ T \rightarrow F \left(\frac{\times}{\varepsilon}\, F\, \frac{\varepsilon}{\mathsf{mult}} \mid \frac{\div}{\varepsilon}\, F\, \frac{\varepsilon}{\mathsf{div}} \right)^* \\ F \rightarrow \frac{v}{v} \mid \frac{\text{`(`}}{\varepsilon}\, E\, \frac{\text{`)`}}{\varepsilon} \end{cases}$$

In the grammar rules, the elements of the target string, i.e., the characters of alphabet Δ, appear in the fraction denominators. The source and target alphabets Σ and Δ are as follows:

$$\Sigma = \{\ +,\ -,\ \times,\ \div,\ v,\ \text{`(`},\ \text{`)`}\ \}$$

$$\Delta = \{\ \mathsf{add},\ \mathsf{sub},\ \mathsf{mult},\ \mathsf{div},\ v\ \}$$

and the round parentheses disappear from the target alphabet Δ, since they are unnecessary in the postfix polish form. □

Notice that we have not formally defined the concept of extended *BNF* translation grammar, but we rely upon the reader's intuition for it.

5.4.6 Bottom-Up Deterministic Translation

Now, consider a context-free translation scheme and assume the source grammar is suitable for bottom-up deterministic parsing, by the *ELR* (1) condition on p. 254 (actually by the *LR* (1) condition since this time the grammar is not extended). Unlike the top-down case, it is not always possible to extend the parser with the write actions that compute the translation, without jeopardizing determinism. Intuitively, the impediment is simple to understand. We know the parser works by shifts and reductions: a shift pushes on stack a macro-state of the pilot finite automaton, i.e., a set of states of some machines. Of course, when a shift is performed, the parser does not know which rule will be used in the future for a reduction, and conservatively keeps all the candidates open. Imagine that two distinct machine states occur as candidates in the current macro-state, and make the likely hypothesis that different output actions are associated with them. Then, when the parser shifts, the translator should perform two different and contradictory write actions, which is impossible for a deterministic transducer. Thus, the reasoning above explains the impossibility to emit the output during a shift move of the translator.

On the other hand, when a reduction takes place, exactly one source grammar rule has been recognized, which is identified by the final state of the corresponding machine. Since the mapping between source and target rules in the translation scheme is a total function, a reduction identifies exactly one target rule and can safely output the associated target string.

We present a case (Example 5.15) where the translation grammar has rules that contain output symbols exclusively localized at the rule end, so that the pilot automaton can be enriched with write actions in the reduction m-states.

Example 5.15 (*translation of expressions to postfix notation*) The *BNF* translation grammar below, which specifies certain formulas that use two infix signs[9] to be translated to postfix operators, is represented as a machine network (axiom E) in Fig. 5.5:

$$E \to E \;\underset{\varepsilon}{+}\; T \;\underset{\mathsf{add}}{\varepsilon} \;\mid\; T \qquad\qquad T \to T \;\underset{\varepsilon}{\times\, a} \;\underset{a\ \mathsf{mult}}{\varepsilon} \;\mid\; \underset{\varepsilon}{a}\;\underset{a}{\varepsilon}$$

Notice that care has been taken to position all the target characters as suffix at the rule end; incidentally, rule $E \to T$ does not need target characters. The *ELR* (1) pilot graph can be easily upgraded for translation by inserting the output actions in the reductions, as shown in Fig. 5.6. Of course, the parser will execute a write action when it performs the associated reduction. □

[9] More precisely, two-level infix arithmetic expressions of type sum of product, with variables a but without parentheses.

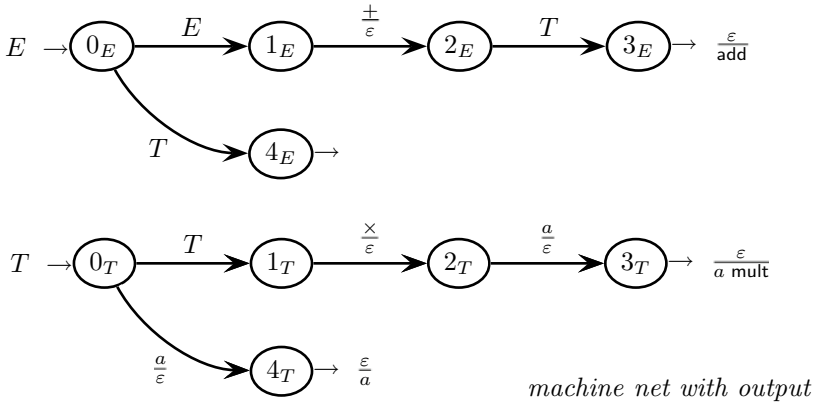


Fig. 5.5 Machine net of the *BNF* translation grammar of Example 5.15

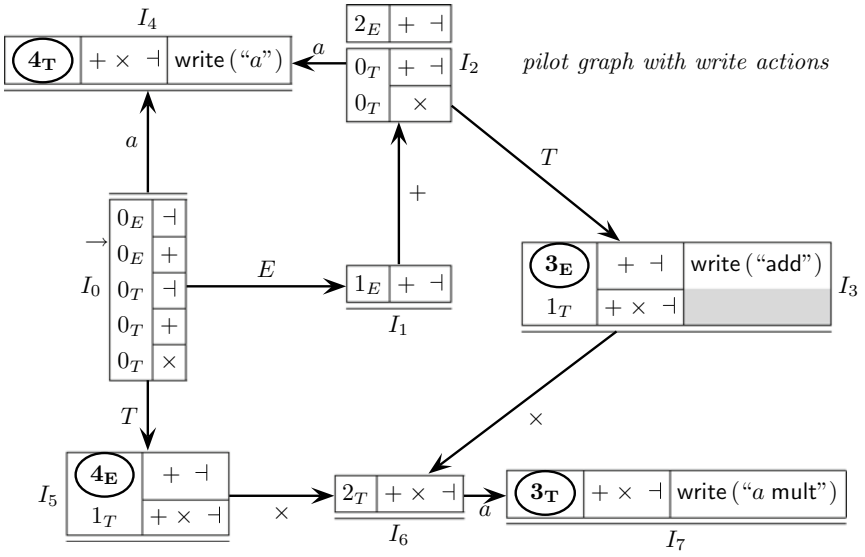


Fig. 5.6 Pilot of the translator of Example 5.15 with write actions in the reduction m-states. (The closure of m-states I_0 and I_2 is shown step by step.)

On the contrary, in the next case (Example 5.16) the presence of write actions on shift moves makes the translation incorrect.

Example 5.16 (*translation of parenthesized expressions*) The *BNF* grammar G_τ below specifies a translation of a language similar to the Dyck one, where every pair of matching "parentheses" a, c is translated to the pair b, e, but with one exception: no translation is performed for the innermost pair of symbols a, c, which is just erased.

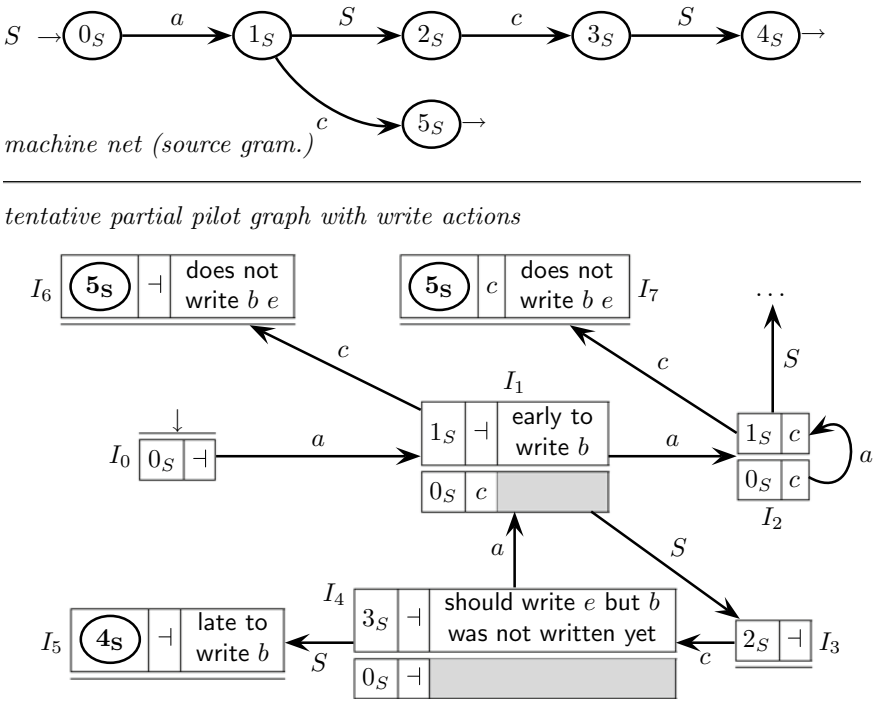


Fig. 5.7 Machine net and tentative partial pilot with write actions (Example 5.16)

$$G_\tau\colon S \to \frac{a}{b}\ S\ \frac{c}{e}\ S \mid \frac{a}{\varepsilon}\ \frac{c}{\varepsilon} \qquad \begin{aligned} \tau(a\ a\ c\ c\ a\ c) &= b\ e \\ \tau(a\ c) &= \varepsilon \end{aligned}$$

Figure 5.7 shows the machine net of the source grammar G_1 of G_τ. The final states are separate in order to identify which alternative rule has been analyzed, as each rule has a distinct output.

In Fig. 5.7 an attempt is made to obtain an *ELR* (1) pilot with write actions; for brevity the construction is partial. The attempt fails almost immediately: either the write actions are premature, as in the m-state I_1 where it is unknown whether the letter a that has just been shifted belongs to an innermost pair or not, or they are too late, as in the m-state I_5. □

5.4.6.1 Postfix Normal Form

In practice, when specifying a context-free translation intended for bottom-up parsing, it is preferable to put the translation grammar into a form (Definition 5.17) that confines write actions within reduction moves.

Definition 5.17 (*postfix translation grammar*) A translation grammar or scheme is in the *postfix normal form* if every target grammar rule has the form $A \to \gamma\, w$, where $\gamma \in V^*$ and $w \in \Delta^*$. □

Differently stated, no target string may occur as an inner substring of a rule, but only as a suffix. Example 5.2 (p. 389) is in the postfix form, while Examples 5.12 (p. 401) and 5.16 (p. 407) are not.

One may wonder what loss or worsening, if any, is caused by the postfix condition. From the standpoint of the family of translation relations that can be specified, it is easy to show that postfix grammars have the same expressivity as the general context-free translation grammars do, sometimes at the cost of some obscurity. In Algorithm 5.18 we explain the transformation of a generic translation grammar into postfix normal form.

Algorithm 5.18 (*converting a translation grammar to postfix form*) Consider in turn each rule $A \to \alpha$ of the given translation grammar. If the rule violates the postfix condition, find the maximal (longest) target string $v \in \Delta^+$ that occurs in the rightmost position in α, and transcribe the rule as:

$$A \to \gamma \, \frac{\varepsilon}{v} \, \eta$$

where γ is any string and η is a non-empty string devoid of target characters. Replace this rule with the next ones:

$$A \to \gamma\ Y\ \eta \qquad\qquad Y \to \frac{\varepsilon}{v}$$

where symbol Y is a new nonterminal. The second rule complies with the postfix condition; if the first rule does not (yet), find anew the maximal rightmost target string within string γ, and repeat the transformation above. Eventually, all the target elements that occur in the middle of a rule will have been moved to suffix positions, and so the resulting grammar will be in the postfix normal form. □

The next example (Example 5.19) illustrates the transformation to postfix, and it should convince the reader that the original and transformed grammars define the same translation.

Example 5.19 (*grammar transformation to postfix normal form*) Examining the translation grammar G_τ: $S \to \frac{a}{b}\, S\, \frac{c}{e}\, S \mid a\, c$ of the previous example (Example 5.16), we notice that in the target grammar G_2 two letters (pointed) violate the postfix form:

$$\begin{array}{ccccccc} & & & \downarrow & & \downarrow & \\ S & \to & b & S & e & S & \end{array}$$

To normalize the grammar, we apply Algorithm 5.18, thus obtaining the following grammar, which contains two new nonterminals:

$$\begin{array}{rllll} G_\tau: & S \to A\ S\ C\ S \mid a\ c & A \to \frac{a}{b} & C \to \frac{c}{e} \\ G_{2\ \text{postfix}}: & S \to A\ S\ C\ S \mid \varepsilon & A \to b & C \to e \end{array}$$

Notice that A stands for the left parenthesis b and B for the right one e. It is easy to check that the original and the postfix target grammars G_2 and $G_{2\ \text{postfix}}$ define the same translation. The machine net in Fig. 5.8 represents the translation grammar normalized to postfix form. Since the grammar is *BNF* and the two alternative rules of the axiom S do not have any output, the two recognizing paths of machine M_S end in the same final state, 4_S. The pilot of the postfix grammar, partially shown in Fig. 5.8, writes the output only at reduction time. Now, the write actions in the pilot automaton are, respectively, associated with the reduction states 1_A and 1_C of nonterminals A and C, which correspond to parenthesis pairs that are not innermost. Also, notice that there is not any write action in the m-state I_5, which includes the final state 4_S and hence specifies a reduction to nonterminal S, because the symbol c just shifted to reach m-state I_5 certainly belongs to an innermost pair. □

We summarize the previous discussion on bottom-up translators with a sufficient condition (Property 5.20) for being deterministic.

Property 5.20 (determinism of bottom-up transducers) *A translation defined by a* BNF *(not extended) translation grammar in the postfix normal form, such that the source grammar satisfies condition* LR (k) *with* $k \geq 1$*, can be computed by a deterministic bottom-up parser, which writes on output at reduction moves exclusively.* □

In essence, the postfix form allows the parser to defer its write actions until it reaches a state where the action is uniquely identified.

However, this normalization method has some drawbacks. The introduction of new nonterminal symbols, such as A and B in Example 5.19, makes the new grammar less readable. Another nuisance may come from rule normalization, when empty rules such as $Y \to \varepsilon$ are added to the source grammar. Empty rules tend to increase the length k of the look-ahead needed for parsing; in some cases the *LR* (k) property may be lost, as the next example (Example 5.21) shows.

Example 5.21 (*loss of LR* (1) *property caused by normalization*) The translation of an infix expression[10] to prefix form is specified by the grammar G_τ below. By looking

[10] The expression is simply a summation $a + a + \cdots + a$, of one or more terms.

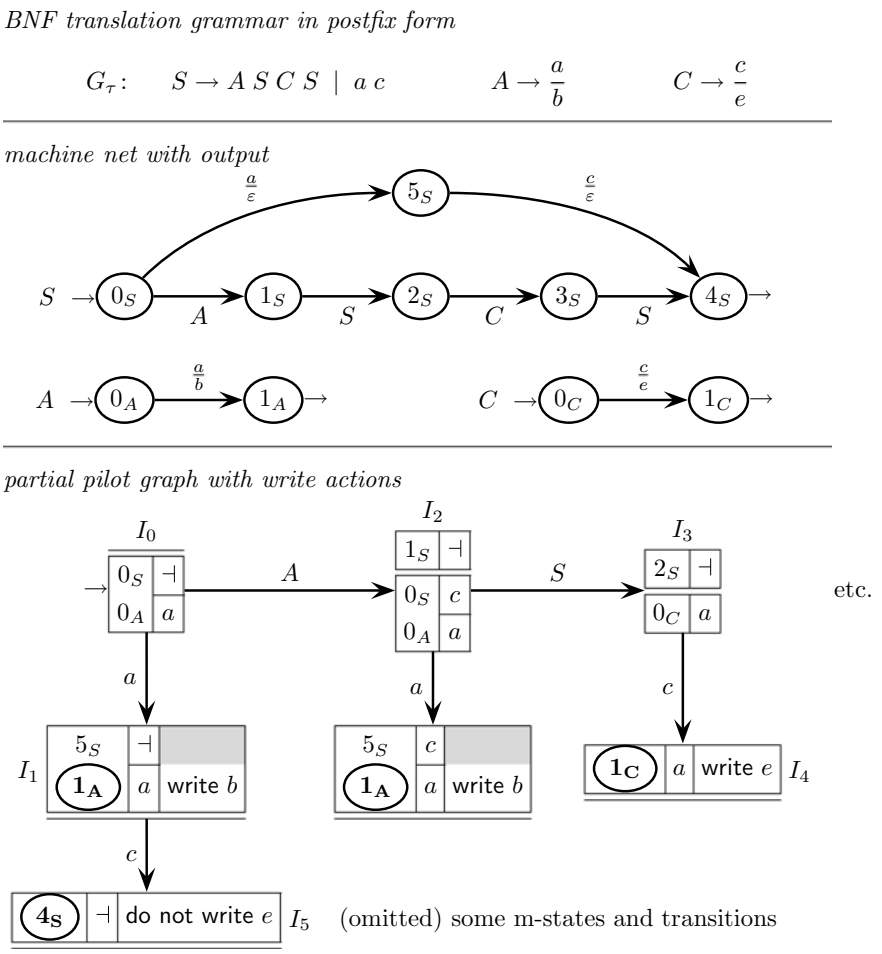


Fig. 5.8 Translation grammar G_τ in the postfix form, machine net with output and partial pilot with write actions (Example 5.19)

at the target grammar G_2, we ascertain that the first rule of G_τ is not in the postfix form.

G_τ *original*	G_1	G_2
$E \to \frac{\varepsilon}{\mathsf{add}}\ E\ \frac{+a}{a}$	$E \to E\ +\ a$	$E \to \mathsf{add}\ E\ a$
$E \to \frac{a}{a}$	$E \to a$	$E \to a$

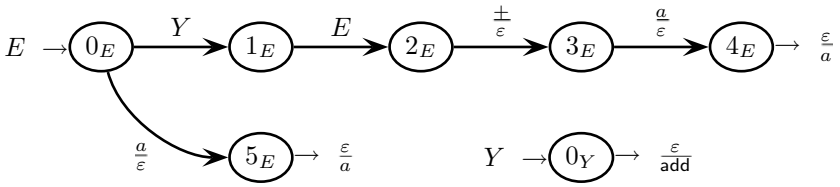


Fig. 5.9 Machine net with output of a *BNF* translation grammar normalized in the postfix form (Example 5.21)

Here is the postfix form G'_τ of grammar G_τ, obtained by normalization (Algorithm 5.18):

G'_τ *postfix*	G'_1	G'_2
$E \to Y\ E\ \frac{+\ a}{\varepsilon}\ \frac{\varepsilon}{a}$	$E \to Y\ E\ +\ a$	$E \to Y\ E\ a$
$E \to \frac{a}{\varepsilon}\ \frac{\varepsilon}{a}$	$E \to a$	$E \to a$
$Y \to \frac{\varepsilon}{\mathsf{add}}$	$Y \to \varepsilon$	$Y \to \mathsf{add}$

The machine net of grammar G'_τ is in Fig. 5.9. It is easy to check that grammar G'_τ and the original G_τ define the same translation. Unfortunately, while the original source grammar G_1 satisfies the *LR* (1) condition, the new rule $Y \to \varepsilon$ of the postfix source grammar G'_1 makes state 0_Y both initial and final, and causes an *LR* (1) shift-reduce conflict in the macro-states I_0, I_1 and I_6 of the pilot shown in Fig. 5.10. □

Fortunately, in many practical situations grammar normalization to postfix form does not hinder deterministic parsing.

5.4.6.2 Syntax Tree as Translation

A common application of syntactic translation (both top-down and bottom-up) is to construct the syntax tree of the source text. Programs usually represent a tree as a linked data structure, but to construct such a representation we need semantic actions, to be discussed in later sections. Here, instead of producing a linked list, which would be impossible to do with a purely syntactic translation, we are content with outputting the sequence of labels of the source rules, and in the order they occur in the series of reduction steps executed by the shift-reduce parser. This is similar to Example 5.13 above and only differs in the label order, which corresponds to a reversed right derivation instead of a left one; see also the tree visit orders in Fig. 4.10 on p. 232.

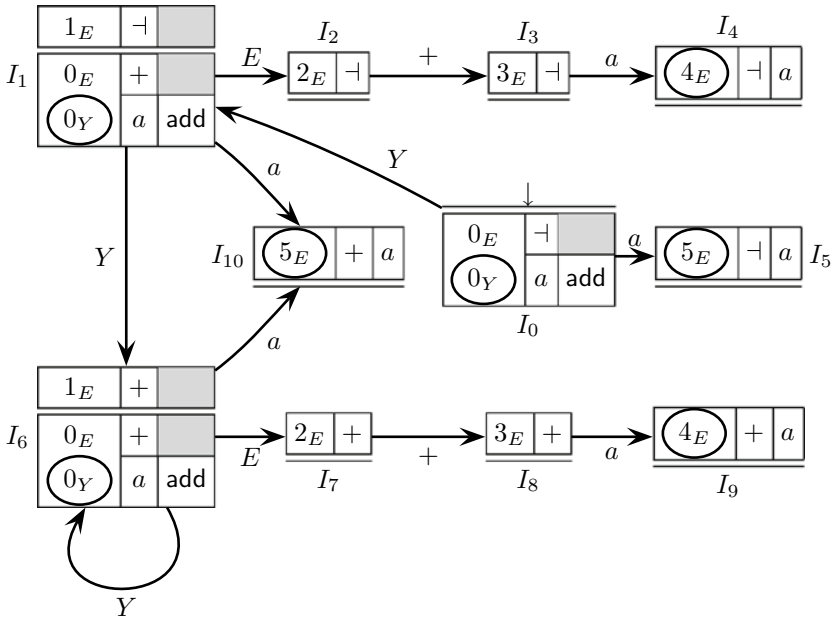


Fig. 5.10 Pilot *LR* (1) of the translation grammar of Fig. 5.9. Correctly, write actions are only in reduction m-states, but there are shift-reduce conflicts in m-states I_0, I_1 and I_6

Given a source grammar with rules labeled for reference, the next postfix translation scheme produces the label sequence in the order of execution of the reduction steps:

label	*source grammar rule*	*translation grammar rule*
r_i	$A \to \alpha$	$A \to \alpha \ \frac{\varepsilon}{r_i}$

It is not difficult to see that the image produced by the above translation is the reversal of the sequence of rule labels occurring in the rightmost derivation of the input string. Since by hypothesis the source grammar is *LR* (1) and the scheme is postfix, it suffices to enrich the parser with write actions to compute this translation.

5.4.6.3 Comparison of Top-Down and Bottom-Up Approaches

We recapitulate the main considerations about upgrading parsers to translators. The main argument in favor of the top-down methods is that they do not suffer any limitation as they allow the implementation of any syntactic translation scheme, provided of course that the source grammar meets the suitable condition for parsing. Moreover, a recursive descent translator can be easily constructed by hand, and the resulting program is easy to understand and maintain.

On the other hand, for the bottom-up methods the limitation imposed by the postfix normal form of the translation grammar may be compensated by the superiority of the $LR\,(k)$ grammars over the $LL\,(k)$ ones, for the definition of the source language. In conclusion neither method, top-down versus bottom-up, is entirely superior to the other one.

5.5 Regular Translation

Just like context-free grammars have as a special case the right-linear grammars, which in turn have regular expressions and finite automata as their natural counterpart, in a similar way translation grammars include as a special case the right-linear translation grammars, which define translations that can be characterized in terms of regular translation expressions and finite transducers.

Consider for instance the translation τ below which converts a letter a to b or c, depending on the number of letters being even or odd:

$$\tau \begin{cases} a^{2n} \overset{\tau}{\mapsto} b^{2n} & n \geq 0 \\ a^{2n+1} \overset{\tau}{\mapsto} c^{2n+1} & n \geq 0 \end{cases} \tag{5.3}$$

Translation τ is specified by the following right-linear translation grammar G_τ (axiom A_0):

$$G_\tau \begin{cases} A_0 \to \frac{a}{c}\, A_1 \mid \frac{a}{c} \mid \frac{a}{b}\, A_3 \mid \varepsilon \\ A_1 \to \frac{a}{c}\, A_2 \mid \varepsilon \\ A_2 \to \frac{a}{c}\, A_1 \\ A_3 \to \frac{a}{b}\, A_4 \\ A_4 \to \frac{a}{b}\, A_3 \mid \varepsilon \end{cases}$$

The well-known equivalence between right-linear grammars and finite automata is now extended by defining regular translation expressions.

First, regular expressions can be modified in order to specify a translation relation: the arguments of the expression are pairs of source/target strings, instead of being characters as customary. Therefore, a sentence generated by such an r.e. model is a sequence of pairs. By separating the source component of each pair from the target one, we obtain two strings, which can be interpreted as a pair belonging to the translation relation. Thus, such an r.e. model defines a translation relation to be called *regular* or *rational*.

By this approach, the translation relation τ in (5.3) is defined by the following regular translation expression e_τ:

$$e_\tau = \left(\frac{a^2}{b^2}\right)^* \cup \frac{a}{c}\left(\frac{a^2}{c^2}\right)^*$$

The string of fractions below:

$$\frac{a}{c} \cdot \left(\frac{a^2}{c^2} \right)^2 = \frac{a}{c} \cdot \frac{a^2}{c^2} \cdot \frac{a^2}{c^2} = \frac{a^5}{c^5} \in L\,(e_\tau)$$

then corresponds to the pair $\left(a^5,\, c^5\right)$ in the translation relation ρ_τ defined by the expression e_τ.

Example 5.22 (*consistent transliteration of an operator*) The source text is a list of numbers separated by a division sign "/". The translation may replace the sign by either one of the signs ":" or "÷", but it must consistently choose the same sign throughout. For simplicity we assume the numbers to be unary. The source/target alphabets are as follows:

$$\Sigma = \{\, 1,\ `/'\,\} \qquad \Delta = \{\, 1,\ `\div',\ `:'\,\}$$

The source strings have the form $c\ (\,/\,c\,)^*$, where c stands for any unary number denoted by 1^+. Two valid translations are the following:

$$(\,3\,/\,5\,/\,2,\ 3\ :\ 5\ :\ 2\,) \qquad (\,3\,/\,5\,/\,2,\ 3\ \div\ 5\ \div\ 2\,)$$

On the contrary, transliteration $(\,3\,/\,5\,/\,2,\ 3\ :\ 5\ \div\ 2\,)$ is wrong, because the division signs are differently transliterated.

Notice this translation cannot be expressed by a homomorphism, as the image of the division sign is not single-valued; on the other hand, the inverse translation is an alphabetic homomorphism. The translation is defined by the r.e. below:

$$(1,\ 1)^+\ \left((`/',\ `:')\ (1,\ 1)^+\right)^* \cup\ (1,\ 1)^+\ \left((`/',\ `\div')\ (1,\ 1)^+\right)^*$$

or better by the more readable fractional notation below:

$$\left(\frac{1}{1}\right)^+ \left(\frac{/}{:}\left(\frac{1}{1}\right)^+\right)^* \cup \left(\frac{1}{1}\right)^+ \left(\frac{/}{\div}\left(\frac{1}{1}\right)^+\right)^*$$

The terms produced by applying a derivation are strings of fractions, i.e., string pairs. Consider the following derived string:

$$\left(\frac{1}{1}\right)^1 \left(\frac{/}{\div}\left(\frac{1}{1}\right)^2\right)^1 = \frac{1}{1}\ \frac{/}{\div}\ \frac{1}{1}\ \frac{1}{1}$$

project it on the top and bottom components, and thus obtain the pair of source/target strings $(\,1\,/\,1\,1,\ 1\ \div\ 1\,1\,)$. □

We summarize the previous discussion and examples (Example 5.22) about regular translations, in the next definition.

Definition 5.23 (*regular (or rational) translation*) A *regular* or *rational translation expression*, shortly r.t.e., is a regular expression with union, concatenation and star (and cross) operators that has as arguments some string pairs (u, v), also written as $\frac{u}{v}$, where terms u and v are possibly empty strings, respectively, on the source and on the target alphabet.

Let $C \subset \Sigma^* \times \Delta^*$ be the set of the pairs (u, v) occurring in the expression. The *regular* or *rational translation relation* defined by the r.t.e. e_τ consists of the pairs (x, y) of source/target strings such that:

- There exists a string $z \in C^*$ in the regular set defined by the r.t.e. e_τ.
- Strings x and y are the projections of string z on the first and second components, respectively. □

It is straightforward to see that the set of source strings defined by an r.t.e. (as well as the set of target strings) is a regular language. Yet notice that not every translation relation that has two regular sets as its source and target languages can be defined with an r.t.e.: an example to be discussed later is the relation that maps each string to its mirror string.

5.5.1 Two-Input Automaton

Since the set C of pairs occurring in an r.t.e. can be viewed as a new terminal alphabet, the regular language over C can be recognized by a finite automaton, as illustrated in the coming example.

Example 5.24 (*consistent transliteration of an operator (Example 5.22) continued*) The recognizer of the regular translation relation is shown in Fig. 5.11.

This automaton can be viewed as a machine with *two* read-only input tapes, in short a $2I$-machine, each one with an independent reading head, respectively, containing the source string x and the target string y. Initially the heads are positioned on the first characters and the machine is in the start state. The machine performs as specified by the state-transition graph: e.g., in state q_1, on reading a slash "/" from the source tape and a sign "÷" from the target tape, the automaton moves to state q_4 and shifts both heads by one position. If the machine reaches a final state and both tapes have been entirely scanned, the pair (x, y) belongs to the translation relation.[11]

This automaton can check that two strings, such as:

$$(1\,1\,\text{`}/\text{'}\,1,\ 1\,\text{`}\div\text{'}\,1) \equiv \frac{1\,1\,/\,1}{1\,\div\,1}$$

[11] The model is known as a Rabin and Scott machine. For greater generality such a machine may be equipped with three or more tapes, in order to define a relation between more than two languages; see Sakarovitch [2] and Berstel [1] for this and similar models.

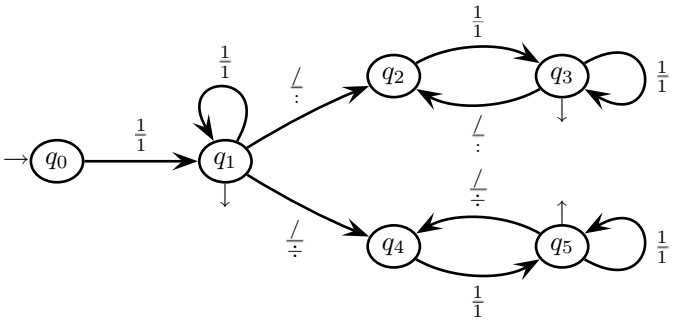


Fig. 5.11 *2I*-automaton of the r.t.e. of Examples 5.22 and 5.24

do not correspond in the translation, because the following computation:

$$q_0 \xrightarrow{\frac{1}{1}} q_1$$

does not admit any continuation with the next pair, i.e., fraction $\frac{1}{\div}$. □

It is sometimes convenient to assume that each tape is delimited on the right by a reserved character marking the tape end.

At a first glance, regular translation expressions and two-input machines may seem to be wrong idealizations for modeling a compiler, because in compilation the target string is not given and must be computed by the translator. Yet this conceptualization is valuable for specifying some simple translations and also as a rigorous method for studying translation functions.

When designing a two-input recognizer, we can assume without loss of generality that each move reads one character or nothing from both the source and the target tapes; the following definition (Definition 5.25) formalizes the concept.

Definition 5.25 (*two-input automaton or* 2*I-automaton*) A *finite automaton with two inputs* or 2*I-automaton* is defined by a set of states Q, the initial state $q_0 \in Q$ and a set $F \subseteq Q$ of final states. The transition function δ is defined as follows:

$$\delta\colon Q \times (\Sigma \cup \{\varepsilon\}) \times (\Delta \cup \{\varepsilon\}) \to \wp(Q)$$

An automaton move includes the following actions:

- The automaton enters state q'.
- If $q' \in \delta(q, a, b)$, the automaton reads characters a from the source tape and b from the target tape; if $q' \in \delta(q, \varepsilon, b)$, the automaton does not read the source tape and reads character b from the target tape; similarly for the case $q' \in \delta(q, a, \varepsilon)$; if $q' \in \delta(q, \varepsilon, \varepsilon)$, the automaton reads neither the source nor the target tape.

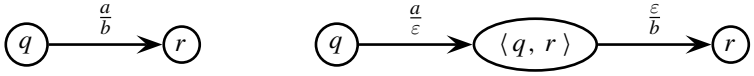
The automaton recognizes the source and target strings if the computation ends in a final state after both tapes have been entirely scanned. □

Imagine now to project the arc labels of a $2I$-automaton onto the first component. The resulting automaton has just one input tape with symbols from the source alphabet Σ and termed the input automaton *subjacent* to the original machine; it recognizes the source language.

Sometimes another *normal* form of a $2I$-automaton is used, characterized by the fact that each move reads exactly one character either from the source tape or from the target tape, but not from both. More precisely the arc labels are of the following two types:

$$\text{label } \frac{a}{\varepsilon} \text{ with } a \in \Sigma, \text{ i.e., read one character from source}$$
$$\text{label } \frac{\varepsilon}{b} \text{ with } b \in \Delta, \text{ i.e., read one character from target}$$

This means that a machine in such a normal form shifts only one head per move and one position at a time. It is to be expected that such a normalization will often increase the number of states because a non-normalized move, like the following one (left), is replaced by the cascade of normalized moves (right):



where the state $\langle q, r \rangle$ is new.

On the other hand, in order to make the model more expressive and concise, it is convenient to allow regular translation expressions as arc labels. As for finite automata, this generalization does not change the computational power, but it helps in hiding the details of complicated examples.

Finally, a short notation: in an arc label we usually drop a component that is the empty string. Thus we may write:

$$\frac{a^+\ b}{d} \mid \frac{a^+\ c}{e}$$

in place of the following:

$$\frac{a^+}{\varepsilon}\ \frac{b}{d} \mid \frac{a^+}{\varepsilon}\ \frac{c}{e}$$

This r.t.e. says that a sequence of letters a, if followed by a letter b, is translated to a letter d; if it is followed by c, it is translated to e.

5.5.1.1 Equivalence of Models

In accordance with the well-known equivalence of regular expressions and finite automata, we state a similar property for translations.

Property 5.26 (equivalence of translation models) *The families of translations defined by regular translation expressions and by finite (nondeterministic)* 2I-*automata coincide.* □

We recall that an r.t.e. defines a regular language R over an alphabet that consists of a finite set of pairs of strings $(u, v) \equiv \frac{u}{v}$, where $u \in \Sigma^*$ and $v \in \Delta^*$. By separately extracting from each string in R the source and target elements, we obtain a crisp formulation of the relation between source and target languages.

Property 5.27 (Nivat theorem) *The following four conditions are equivalent:*

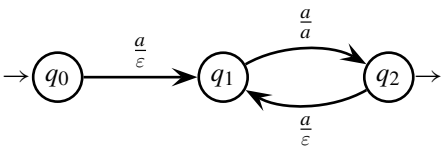
1. *The translation relation ρ_τ is defined by a right-linear (or left-linear) translation grammar G_τ.*
2. *The translation relation ρ_τ is defined by a* 2I-*automaton.*
3. *The translation relation $\rho_\tau \subseteq \Sigma^* \times \Delta^*$ is regular.*
4. *There exist an alphabet Ω, a regular language R over Ω, and two alphabetic homomorphisms $h_1 : \Omega \to \Sigma \cup \{\varepsilon\}$ and $h_2 : \Omega \to \Delta \cup \{\varepsilon\}$, such that:*

$$\rho_\tau = \{ (h_1(z), h_2(z)) \mid z \in R \}$$ □

Example 5.28 (*division by two*) The image string is the halved source string. The translation relation $\left\{ \left(a^{2n}, a^n\right) \mid n \geq 1 \right\}$ is defined by the following r.t.e.:

$$\left(\frac{a\ a}{a}\right)^+$$

An equivalent 2*I*-automaton A is shown below:



To apply the Nivat theorem (clause 4 of Property 5.27), we derive from automaton A the following r.t.e.:

$$\left(\frac{a}{\varepsilon}\frac{a}{a}\right)^+$$

and we rename for clarity the pairs below:

$$\frac{a}{\varepsilon} = c \qquad \frac{a}{a} = d$$

Next, consider the alphabet $\Omega = \{ c, d \}$. The r.t.e. defines the regular language $R = (c\, d)^+$, obtained replacing each fraction with a character of the new alphabet. The following alphabetic homomorphisms h_1 and h_2:

Ω	h_1	h_2
c	a	ε
d	a	a

produce the intended translation relation. Thus for $z = c\, d\, c\, d \in R$ we have $h_1 (z) = a\, a\, a\, a$ and $h_2 (z) = a\, a$.

To illustrate Nivat theorem (clause 1 of Property 5.27), we apply the well-known equivalence of finite automata and right-linear grammars (p. 129), to obtain the following equivalent right-linear translation grammar:

$$S \to \frac{a}{\varepsilon}\, Q_1 \qquad Q_1 \to \frac{a}{a}\, Q_2 \qquad Q_2 \to \frac{a}{\varepsilon}\, Q_1 \mid \varepsilon$$

Each grammar rule corresponds to a move of the 2I-automaton. Rule $Q_2 \to \varepsilon$ is a short notation for $Q_2 \to \frac{\varepsilon}{\varepsilon}$. □

We recall from Sect. 5.4 that for the syntactic translations that have a context-free grammar as their support, the notation based on a translation grammar has to be used, because such translations require a pushdown stack and cannot be described by a finite memory device, such as a 2I-automaton or an r.t.e.

Several, but not all, properties of regular languages have a counterpart for regular translation relations.[12] Thus, the union of regular translation relations still yields a regular translation relation, but it is not always so for their intersection and set difference. For some translation relations, it is possible to formulate a pumping lemma similar to the one for the regular languages on p. 89.

Non-regular Translation of Regular Languages

As already noticed, not every translation relation with two regular languages as source and target is necessarily regular; i.e., it can be defined through a 2I-finite automaton or an r.t.e. Consider for instance the following languages L_1, L_2 and translation τ:

$$L_1 = (a \mid b)^* \qquad L_2 = (a \mid b)^* \qquad \tau (x) = x^R \text{ with } x \in L_1$$

This translation τ cannot be defined by a 2I-automaton, as a finite set of states does not suffice to check that the string on the second tape is the reversal of the one on the first tape.

[12] See the already cited books [1,2] for a comprehensive presentation.

5.5.2 Translation Function and Finite Transducer

We leave the static perspective of a translation as a relation between two strings, and we focus instead on the translation process, where an automaton is viewed as an algorithmic implementation of a translation function. We introduce the *finite transducer* or *IO-automaton*. This machine reads the source string from the input tape and writes the image on the output tape. We shall mostly study single-valued translations and especially those that are computed by deterministic machines, but we introduce the model by means of a nondeterministic example, for the same translation relation τ on p. 414.

Example 5.29 (*nondeterministic translation*) It is required to translate a string a^n to the image b^n if n is even, or to the image c^n if n is odd. The translation relation ρ_τ below:

$$\rho_\tau = \left\{ \left(a^{2n},\, b^{2n}\right) \mid \ n \geq 0 \right\} \cup \left\{ \left(a^{2n+1},\, c^{2n+1}\right) \mid \ n \geq 0 \right\}$$

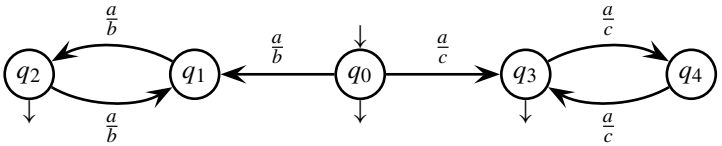
defines the following translation function τ:

$$\tau\left(a^n\right) = \begin{cases} b^n & \text{for } n \geq 0 \text{ even } (n = 0 \text{ is even}) \\ c^n & \text{for } n \geq 1 \text{ odd} \end{cases}$$

The r.t.e. e_τ is the following:

$$e_\tau = \left(\frac{a^2}{b^2}\right)^* \cup \frac{a}{c}\left(\frac{a^2}{c^2}\right)^*$$

A simple deterministic two-input automaton recognizes this relation ρ_τ:



To check for determinism, observe that only the initial state q_0 has two outgoing arcs, but that the source labels of such arcs are different.[13] Therefore the 2I-machine can deterministically decide if two strings memorized on the two tapes, such as $\frac{a\,a\,a\,a}{b\,b\,b\,b}$, correspond to each other in the translation relation.

The transducer or *IO*-automaton has the same state-transition graph as the 2$\boldsymbol{I}$-automaton above, but the meaning of an arc, e.g., $q_0 \xrightarrow{\frac{a}{b}} q_1$, is entirely different:

[13] If the 2I-automaton has some moves that do not read from either tape, the determinism condition has to be formulated more carefully; e.g., see [2].

in the state q_0 it reads character a from the input, writes character b to the output and enters state q_1. Yet the other arc $q_0 \xrightarrow{\frac{a}{c}} q_3$ instructs the machine to perform a different action, while reading the same character a in the same state q_0. In other words, the choice between the two moves is not deterministic. As a consequence, two computations are possible for the input string $a\,a$:

$$q_0 \to q_1 \to q_2 \qquad q_0 \to q_3 \to q_4$$

but only the former succeeds in reaching a final state, namely q_2, and thus only its output is considered: $\tau\,(a\,a) = b\,b$. Notice that the presence of nondeterministic moves shows up also in the input automaton subjacent to the transducer, which is clearly nondeterministic.

Moreover, it should be intuitively clear that the requested translation τ cannot be computed by any deterministic finite transducer, because the choice of the character to emit can only be made when the input tape has been entirely scanned; but then it is too late to decide how many characters have to be output. □

To sum up the findings of this example (Example 5.29), there are single-valued regular translations that cannot be computed by a deterministic finite *IO*-automaton. This is a striking difference with respect to the well-known equivalence of the deterministic and nondeterministic models of finite automata (Property 3.24 on p. 147).

5.5.2.1 Sequential Transducer

In some applications it is necessary to efficiently compute the translation in real time, because the translator must produce the output while scanning the input. At last, when the input is finished, the automaton may append to the output a finite piece of text that depends on the final state reached. This is the behavior of a type of deterministic machine called a sequential transducer.[14] The next definition formalizes the concept (Definition 5.30).

Definition 5.30 (*sequential transducer*) A *sequential transducer* or *IO-automaton* T is a deterministic machine defined by a set Q of states, a source alphabet Σ and a target alphabet Δ, an initial state q_0 and a set $F \subseteq Q$ of final states. Furthermore there are three single-valued functions:

1. The *state-transition* function δ computes the next state.
2. The *output function* η computes the string to be emitted by a move.
3. The *final function* φ computes the last suffix to be appended to the target string at termination.

[14] This is the terminology of [2]; but [1] calls subsequential the same model. In electrical engineering a sequential machine is a quite similar logical device that computes a binary output, which is a function of the sequence of input bits.

The domains and images of these three functions are the following (symbol $\dashv$ is the end-marker of the input):

$$\delta\colon Q \times \Sigma \to Q \qquad \eta\colon Q \times \Sigma \to \Delta^* \qquad \varphi\colon F \times \{\dashv\} \to \Delta^* \qquad \square$$

In the graphical presentation as a state-transition graph, the two functions $\delta(q, a) = r$ and $\eta(q, a) = u$ are represented by the arc $q \xrightarrow{\frac{a}{u}} r$, which means: in the state q, when reading character a, emit string u and move to the next state r. The final function $\varphi(r, \dashv) = v$ means: when the source string has been entirely scanned, if the final state is r, then write string v.

For a source string x, the translation $\tau(x)$ computed by the sequential transducer T is the concatenation of two strings, produced by the output function and by the final one:

$$\tau(x) = \left\{ y\,z \in \Delta^* \;\middle|\; \begin{array}{l} \exists \text{ a labeled computation } \frac{x}{y} \text{ that ends in the} \\ \text{state } r \in F \text{ and it holds } z = \varphi(r, \dashv) \end{array} \right\}$$

The machine is deterministic because the input automaton $\langle Q, \Sigma, \delta, q_0, F \rangle$ subjacent to T is deterministic, and the output and final functions η and φ are single-valued.

However, the condition that the subjacent input automaton is deterministic does not ensure by itself that the translation is single-valued, because between two states of the sequential transducer T there may be two arcs labeled $\frac{a}{b}$ and $\frac{a}{c}$, which cause the output not to be unique.

We call *sequential* a translation function that is computable by a sequential transducer, as of Definition 5.30. The next example (Example 5.31) shows a simple use of a sequential translation, inspired to arithmetic.

Example 5.31 (*meaningless zeroes*) The source text is a list of binary integers, i.e., a list of bit sequences, separated by a blank represented by symbol "␣". The single-valued translation deletes all the meaningless zeros, i.e., the leading ones; an integer of all zeroes is translated to only one zero. This translation is defined by the following r.t.e.:

$$\left(\left(\frac{0^+}{0} \;\middle|\; \left(\frac{0}{\varepsilon}\right)^* \; \frac{1}{1} \; \left(\frac{0}{0} \,\middle|\, \frac{1}{1} \right)^* \right) \; \frac{\sqcup}{\sqcup} \right)^* \left(\frac{0^+}{0} \;\middle|\; \left(\frac{0}{\varepsilon}\right)^* \; \frac{1}{1} \; \left(\frac{0}{0} \,\middle|\, \frac{1}{1} \right)^* \right) \; \frac{\dashv}{\varepsilon}$$

The equivalent sequential transducer is shown in Fig. 5.12. The final function φ does not write anything in the final state q_1, whereas it writes a character 0 in the final state q_2. For instance, when translating the source string 0 0 ␣ 0 1, the machine traverses the state sequence $q_0\,q_2\,q_2\,q_0\,q_2\,q_1$ and writes the target string $\varepsilon \cdot \varepsilon \cdot 0 \cdot \sqcup \cdot \varepsilon \cdot 1 \cdot \varepsilon = 0 \sqcup 1$. For the source string 0 0, the machine traverses the state sequence $q_0\,q_2\,q_2$, writes the target string $\varepsilon \cdot \varepsilon$ and at last writes the target character 0. $\square$

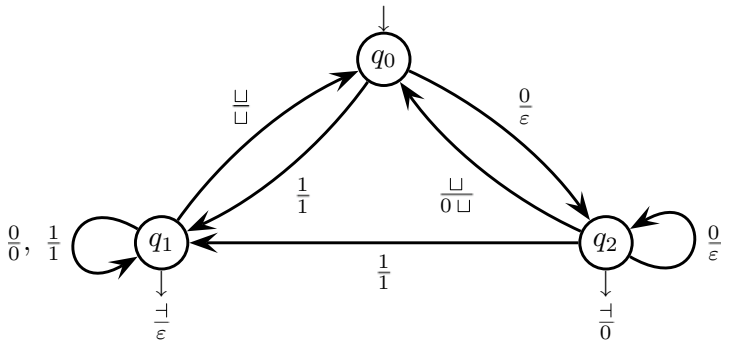


Fig. 5.12 Sequential transducer (*IO*-machine) of Example 5.31. Notice the graphical representation of the final function φ by means of the fractions $\frac{\dashv}{\varepsilon}$ and $\frac{\dashv}{0}$ that label the final darts on states q_1 and q_2, respectively

We show a second example of sequential transducer that makes an essential use of the final function φ for computing a translation τ specified as follows:

$$\tau\,(a^n) = \begin{cases} e & \text{for } n \geq 0 \text{ even} \\ o & \text{for } n \geq 1 \text{ odd} \end{cases}$$

The sequential transducer has only two states, both final, that correspond to the parity classes, even and odd, of the source string. The machine does not write anything, while it switches from one state to the other one; at the end, depending on the final state, it writes character either e or o for either even or odd, respectively.

To conclude, we mention a practically relevant property: the composition of two sequential functions is still a sequential function. This means that the cascade of two sequential transducers can be replaced by just one (typically larger) transducer of the same kind.

5.5.2.2 Two Opposite Passes

Given a single-valued translation specified by a regular translation expression or by a 2*I*-automaton, we have seen that it is not always possible to implement the translation by means of a sequential transducer, i.e., a deterministic *IO*-automaton. On the other hand, even in such cases the translation can always be implemented by two cascaded (deterministic) sequential passes, each one being one way but scanning the string in opposite directions. In the first pass a sequential transducer scans from left to right and converts the source string into an intermediate string. Then in the second pass another sequential transducer scans the intermediate string from right to left and produces the specified target string. We show a simple case.

Example 5.32 (*regular translation by two one-way passes in opposite directions*) We recall from Example 5.29 on p. 421 that the translation of string a^n into string b^n if n is even or into c^n if n is odd (with $n \geq 0$) cannot be deterministically computed by an *IO*-automaton. The reason is that by a one-way scan, the parity class of the string would be known just at the end; but then it is too late for writing the output, because number n is unbounded and exceeds the finite memory capacity of the machine. We show an implementation by two cascaded sequential transducers that scan their respective input in opposite directions.

The first sequential machine scans the input from left to right and computes the intermediate translation τ_1, represented by the r.t.e. e_{τ_1} below:

$$e_{\tau_1} = \left(\frac{a}{a'} \; \frac{a}{a''} \right)^* \left[\frac{a}{a'} \right]$$

which maps to a' (resp. to a'') a character a occurring at odd (resp. at even) position in the input string. The last term $\frac{a}{a'}$ may be missing.

The second transducer scans the intermediate text from right to left and computes the final translation τ_2, represented by the r.t.e. e_{τ_2} below:

$$e_{\tau_2} = \left(\frac{a''}{b} \; \frac{a'}{b} \right)^* \cup \left(\frac{a'}{c} \; \frac{a''}{c} \right)^* \frac{a'}{c}$$

where the choice between the two sides of the union is controlled by the first character being a' or a'' in the reversed intermediate string produced by τ_1. Thus, for instance, for the source string $a\,a$ we have the following:

$$\tau_2 \left((\, \tau_1 \, (a\, a) \,)^R \right) = \tau_2 \left((a'\, a'')^R \right) = \tau_2 \, (a''\, a') = b\, b$$

In fact, a cascade of two one-way sequential transducers that scan their input in opposite directions is equivalent to a one-way transducer equipped with a pushdown stack, which is a more powerful automaton model. □

In several applications the computing capacity of the sequential translator model is adequate to the intended job. In practice, the sequential transducer is often enriched with the capability to look-ahead on the input string, in order to anticipate the choice of the output to be emitted. This enhancement is similar to what has been extensively discussed for parsing in Chap. 4. The sequential translator model with look-ahead is implemented by compiler construction tools of widespread application, such as *Lex* and *Flex*.[15]

[15] For a formalization of look-ahead sequential transducers, we refer to Yang [5].

5.5.3 Closure Properties of Translation

To finish with the purely syntactic translations, we focus now on the formal language families that are induced by such transformations. Given a language L belonging to a certain family, imagine to apply a translator that computes a translation function τ. Then consider the image language generated by τ, below:

$$\tau(L) = \{\ y \in \Delta^* \mid \ \ y = \tau(x) \ \wedge \ x \in L\ \}$$

The question is: what is the language family of the image language $\tau(L)$? For instance, if the language L is context-free and the translator is a pushdown *IO*-automaton, is the target language context-free as well?

One should not confuse the language L and the source language L_1 of the transducer, though both ones have the same alphabet Σ. The source language L_1 includes all and only the strings recognized by the automaton subjacent to the translator, i.e., the sentences of the source grammar G_1 of the translation scheme G_τ. The transducer converts a string of language L to a target string, provided the string is also a sentence of the source language L_1 of the transducer; otherwise an error occurs and the transducer does not produce anything.

The essential closure properties of translations are in the next table:

#	*language*	*finite transducer*	*pushdown transducer*
1	$L \in REG$	$\tau(L) \in REG$	
2	$L \in REG$		$\tau(L) \in CF$
3	$L \in CF$	$\tau(L) \in CF$	
4	$L \in CF$		$\tau(L)$ not always $\in CF$

Cases 1 and 3 descend from the Nivat theorem (Property 5.27 on p. 419) and from the fact that both language families *REG* and *CF* are closed under intersection with regular languages (p. 213). In more detail, the recognizer of language L can be combined with the finite transducer, thus obtaining a new transducer that has the intersection $L \cap L_1$ as source language. The new transducer model is the same as that of the recognizer of L: a pushdown machine if language L is context-free; a finite machine if it is regular. To complete the proof of cases 1 and 3, it suffices to convert the new transducer into a recognizer of the target language $\tau(L)$, by deleting from the moves all the source characters while preserving all the target ones. Clearly, the machine thus obtained is of the same model as the recognizer of L.

For case 2 essentially the same reasoning applies, but now the recognizer of the target language needs in general a pushdown memory. An example of case 2 is the translation (of course by means of a pushdown transducer) of a string $u \in L = \{\ a,\ b\ \}^*$ to the palindromic image $u\,u^R$, which clearly belongs to a context-free language.

Case 4 is different because, as we know from Table 2.9 on p. 95, the intersection of two context-free languages L and L_1 is not always in the family *CF*. Therefore,

it is not certain that a pushdown automaton will be able to recognize the image of L computed by a pushdown translator. The next example (Example 5.33) shows a situation of this kind.

Example 5.33 (*pushdown translation of a context-free language*) To illustrate case 4 of the translation closure table above, consider the translation τ of the following context-free language L:

$$L = \{\, a^n\, b^n\, c^* \mid \; n \geq 0 \,\}$$

to the language with three powers shown below (Example 2.86 on p. 93):

$$\tau\,(L) = \{\, a^n\, b^n\, c^n \mid \; n \geq 0 \,\}$$

which is not context-free, as we know. The image of translation τ is defined by the following translation grammar G_τ, for brevity presented in *EBNF* but easily convertible to *BNF*:

$$G_\tau\colon \quad S \to \left(\frac{a}{a}\right)^*\, X \qquad X \to \frac{b}{b}\, X\, \frac{c}{c} \;\mid\; \varepsilon$$

which constrains the numbers of characters b and c in a target string to be equal, whereas the equality of the number of a's and b's is externally imposed by the fact that any source string has to be in the language L. □

5.6 Semantic Translation

None of the previous purely syntactic translation models is able to compute but the simplest transformations, because they rely on too elementary devices: finite and pushdown *IO*-automata. On the other hand, most compilation tasks need more involved translation functions.

A first elementary example is the conversion of a binary number to decimal. Another typical case is the compilation of data structures to addresses: for example, a `record` declaration as

```
BOOK : record
  AUT: char(8); TIT: char(20); PRICE: real; QUANT: int;
end
```

is converted to a table describing each symbol: type, dimensions in bytes, offset of each field relative to a base address. Assuming the base address of the record is fixed, say, at 3401, the translation is:

symbol	*type*	*dimension*	*address*
BOOK	record	34	3401
AUT	string	8	3401
TIT	string	20	3409
PRICE	real	4	3429
QUANT	int	2	3433

In both examples, to compute the translation we need some arithmetic functions which are beyond the capacity of pushdown transducers. Certainly it would not be viable to use more powerful automata, such as Turing machines or context-sensitive translation grammars, as we have argued (Chap. 2 on p. 105) that such models are already too intricate for language definition, not to mention for specifying translation functions.

A pragmatic solution is to encode the translation function in some programming language or in a more relaxed pseudo-code, as used in software engineering. To avoid confusion, such language is called *compiler language* or *semantic metalanguage*.

The translator is then a program implementing the desired translation function. The implementation of a complex translation function would produce an intricate program, unless care is taken to modularize it, in accordance with the syntax structure of the language to be translated. This approach to compiler design has been used for years with varying degrees of formalization under the title of *syntax-directed translation*. Notice the term "directed" marks the difference from the purely syntactic methods of previous sections.

The leap from syntactic to semantic methods occurs when the compiler includes tree-walking procedures, which move along the syntax tree and compute some variables, called *semantic attributes*. The attribute values, computed for a given source text, compose the translation or, as it is customary to say, the *meaning* or *semantics*.

A syntax-directed translator is not a formal model, because attribute computing procedures are not formalized. It is better classified as a software design method, based on syntactic concepts and specialized for designing input–output functions, such as the translation function of a compiler.

We mention that formalized semantic methods exist, which can accurately represent the meaning of programming languages, using logical and mathematical functions. Their study is beyond the scope of this book.[16]

A syntax-directed compiler performs two cascaded phases:

1. Parsing or syntax analysis
2. Semantic evaluation or analysis

[16]Formal semantic methods are needed if one has to prove that a compiler is correct; i.e., for any source text the corresponding image expresses the intended meaning. For an introduction to formal semantics, see for instance [6,7].

Phase 1 is well known: it computes a syntax tree, usually condensed into a so-called abstract syntax tree, containing just the essential information for the next phase. In particular, most source language delimiters are deleted from the tree.

The semantic phase consists of the application of certain semantic functions, on each node of the syntax tree until all attributes have been evaluated. The set of evaluated attribute values is the meaning or translation.

A benefit of decoupling syntax and semantic phases is that the designer has greater freedom in writing the concrete and abstract syntaxes. The former must comply with the official language reference manual. On the other hand, the abstract syntax should be as simple as possible, provided it preserves the essential information for computing meaning. It may even be ambiguous: ambiguity does not jeopardize the single-value property of translation, because in any case the parser passes just one abstract syntax tree per sentence to the semantic evaluator.

The above organization, termed *two-pass compilation*, is most common, but simpler compilers may unite the two phases. In that case there is just one syntax, the one defining the official language.

5.6.1 Attribute Grammars

We need to explain more precisely how the meaning is superimposed on a context-free language. The meaning of a sentence is a set of attribute values, computed by the so-called semantic functions and assigned to the nodes of the syntax tree. The syntax-directed translator contains the definition of the semantic functions, which are associated with the grammar rules. The set of grammar rules and associated semantic functions is called an *attribute grammar*.

To avoid confusion, in this part of the book a context-free grammar will be called *syntax*, reserving the term *grammar* to attribute grammars. For the same reason the syntactic rules will be called *productions*. In agreement with standard pratice, the context-free productions are in pure BNF.

5.6.1.1 Introductory Example

Attribute grammar concepts are now introduced on a running example.

Example 5.34 (*converting a fractional binary number to base* 10 (*Knuth*[17])) The source language L, defined by the following regular expression:

$$L = \{\ 0,\ 1\ \}^{+}\ \bullet\ \{\ 0,\ 1\ \}^{+}$$

is interpreted as the set of fractional base 2 numbers, with the point separating the integer and fractional parts. Thus the meaning of string 1 1 0 1 • 0 1 is the number $2^3 + 2^2 + 2^0 + 2^{-2} = 8 + 4 + 1 + \frac{1}{4} = 13.25$ in base ten. The attribute grammar

[17] This historical example by D. E. Knuth introduced [8] attribute grammars as a systematization of compiler design techniques used by practitioners.

Table 5.3 Attribute grammar of Example 5.34

#	Syntax	Semantic functions		Comment
1	$N \to D \bullet D$	$v_0 := v_1 + v_2 \times 2^{-l_2}$		Add integer value to fractional value divided by weight 2^{l_2}
2	$D \to D\ B$	$v_0 := 2 \times v_1 + v_2$	$l_0 := l_1 + 1$	Compute value and length
3	$D \to B$	$v_0 := v_1$	$l_0 := 1$	
4	$B \to 0$	$v_0 := 0$		Value initialization
5	$B \to 1$	$v_0 := 1$		

is in Table 5.3. The syntax is listed in column two. The axiom is N, nonterminal D stands for a binary string (integer or fractional part), and nonterminal B stands for a bit. In the third column we see the semantic functions or rules, which compute the following attributes:

attribute	*meaning*	*domain*	*nonterminals that possess the attribute*
v	value	decimal number	N, D, B
l	length	integer	D

A semantic function needs the *support* of a production, and several functions may be supported by the same production. Productions 1, 4 and 5 support one function, while productions 2 and 3 support two functions. Observe now the subscript of the attribute instances, such as v_0, v_1, v_2 and l_2, on the first row of the grammar. A subscript cross-references the grammar symbol possessing that attribute, in accordance with the following numbering convention[18]:

$$\underbrace{N}_{0} \to \underbrace{D}_{1} \bullet \underbrace{D}_{2}$$

stating that instance v_0 is associated with the left part N of the production rule, instance v_1 with the first nonterminal of the right part, etc. However, if in a production a nonterminal symbol occurs exactly once, as N in the first production, the more expressive notation v_N can be used instead of v_0, without confusion.

[18] Alternatively, a more verbose style is used in other texts, e.g., for the first function: v *of* N instead of v_0.

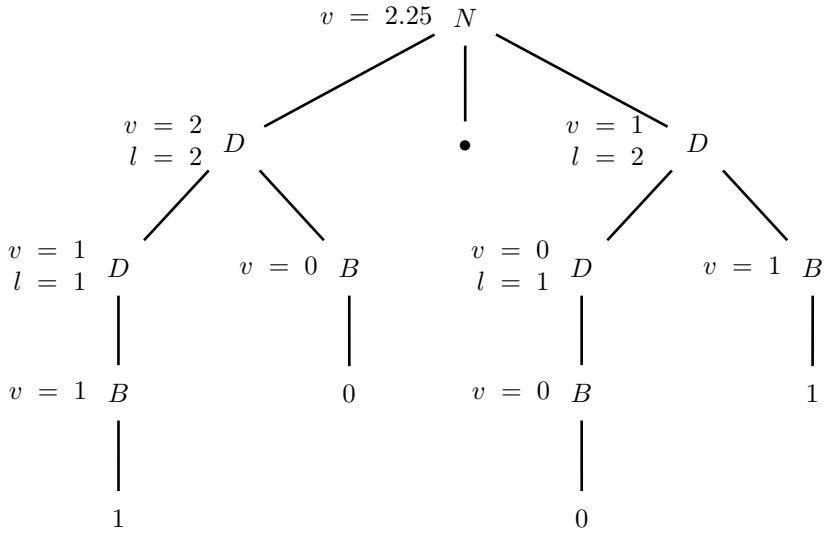


Fig. 5.13 Decorated syntax tree of Example 5.34

The first semantic rule assigns to attribute v_0 a value computed by the expressions containing the attributes v_1, v_2, l_2, which are the function arguments. We can write in functional form:

$$v_0 := f\,(\,v_1,\ v_2,\ l_2\,)$$

We explain how to compute the meaning of a given source string. First we construct its syntax tree; then for each node we apply each function supported by the corresponding production. The semantic functions are first applied to the nodes where the arguments of the functions are available. The computation terminates when all the attributes have been evaluated.

The tree is then said to be *decorated* with attribute values. The decorated tree, which represents the translation or semantics of the source text 1 0 • 0 1, is shown in Fig. 5.13. There are several possible orders or schedules for attribute evaluation: for a schedule to be valid, it must satisfy the condition that no function f is applied before the functions that return the arguments of f.

In this example the final result of the semantic analysis is the attribute of the root, i.e., $v = 2.25$. The other attributes act as intermediate results. The root attribute is then the *meaning* of the source text 1 0 • 0 1. □

5.6.2 Left and Right Attributes

In the grammar of Example 5.34, attribute computation essentially flows from bottom to top because an attribute of the left part (father) of a production rule is defined by a function having as arguments some attributes of the right part (children).

However in general, considering the relative positions of the symbols in the supporting production, the result and arguments of a semantic function may occur in various positions, to be discussed.

Consider a function supported by a production and assigning a value to an attribute (result). We name *left* (or *synthesized*) the attribute if it is associated with the left part of the production. Otherwise, if the result is associated with a symbol in the right part of the production, we say it is a *right* (or *inherited*[19]) attribute. By the previous definition, also the arguments of a function can be classified as left or right, with respect to the supporting production.

To illustrate the classification, first we return to the grammar in Table 5.3: both attributes v and l are left. Then, we show a grammar featuring both left and right attributes.

Example 5.35 (*breaking a text into lines—Reps* [9]) A text has to be segmented into lines. The syntax generates a series of words separated by a space (written $\perp$). The text has to be displayed in a window having a width of $W \geq 1$ characters and an unbounded height, in such a way that each line contains a maximum number of left-aligned words and no word is split across lines. By hypothesis no word has length greater than W. Assume the columns are numbered from 1 to W.

The grammar computes the attribute *last*, which identifies the column number of the last character of each word. For instance, the text "no doubt he calls me an outlaw to catch" with window width $W = 13$ is displayed as follows:

1	2	3	4	5	6	7	8	9	10	11	12	13
n	o		d	o	u	b	t		h	e		
c	a	l	l	s		m	e		a	n		
o	u	t	l	a	w		t	o				
c	a	t	c	h								

Variable *last* takes value 2 for word no, 8 for doubt, 11 for he, …, and 5 for catch.

The syntax generates lists of words separated by a blank space. The terminal symbol c represents any character. To compute the text layout, we use the following attributes:

length	the length of a word (left attribute)
prec	the column of the last character of the preceding word (right attribute)
last	the column of the last character of the current word (left attribute)

To compute attribute *last* for a word, we must first know the column of the last character of the preceding word, denoted by attribute *prec*. For the first word of the text, the value of *prec* is set to -1.

[19] The word *inherited* is used by object-oriented languages in a totally unrelated sense.

Table 5.4 Attribute grammar of Example 5.35

#	Syntax	Right attributes	Left attributes
1	$S_0 \to T_1$	$prec_1 := -1$	
2	$T_0 \to T_1 \perp T_2$	$prec_1 := prec_0$ $prec_2 := last_1$	$last_0 := last_2$
3	$T_0 \to V_1$		$last_0 :=$ **if** $(prec_0 + 1 + length_1) \leq W$ **then** $(prec_0 + 1 + length_1)$ **else** $length_1$ **end if**
4	$V_0 \to c\ V_1$		$length_0 := length_1 + 1$
5	$V_0 \to c$		$length_0 := 1$

Attribute computation is expressed by the rules of the attribute grammar in Table 5.4. Two remarks on the syntax are: first, the subscripts added to nonterminal symbols act as reference to the semantic functions, but they do not differentiate the syntactic classes; i.e., the productions with and without subscripts are equivalent. Second, the syntax has an ambiguity caused by production rule $T \to T \perp T$, which is bilaterally recursive (see Chap. 2 on p. 55). But the drawbacks an ambiguous syntax has for parsing do not concern us here, because the semantic evaluator receives exactly one parse tree to work on. The ambiguous syntax is more concise, and this reduces also the number of semantic rules. The length of a word V is assigned to the left attribute *length* in the rules associated with the last two productions. Attribute *prec* is a right one, because the value is assigned to a symbol in the right part of the first two productions. Attribute *last* is a left one; its value decorates the nodes with label T of a syntax tree and provides the final result in the root of the tree.

To choose a feasible attribute evaluation schedule for a given syntax tree, we have to examine the dependencies between the assignment statements. In Fig. 5.14 the left and right attributes are, respectively, placed to the left and the right of a node; the nodes are numbered for reference. To simplify drawing, the subtrees of nonterminal V are omitted, but attribute *length*, which is the relevant information, is present with its value.

The attributes of a decorated tree can be viewed as the nodes of another directed graph, the (data) *dependence graph*. For instance, observe arc *last* (2) $\to$ *prec* (4) from $last$ of T_2 to $prec$ of T_4: it represents a dependence of the latter attribute from the former, induced by function $prec_2 := last_1$, which is supported by production 2. A function result has as many dependence arcs as it has arguments. Notice the arcs interconnect only attributes pertaining to the same production.

To compute the attributes, the assignments must be executed in any order satisfying the precedences expressed by the dependence graph. At the end, the tree is completely decorated with all the attribute values. □

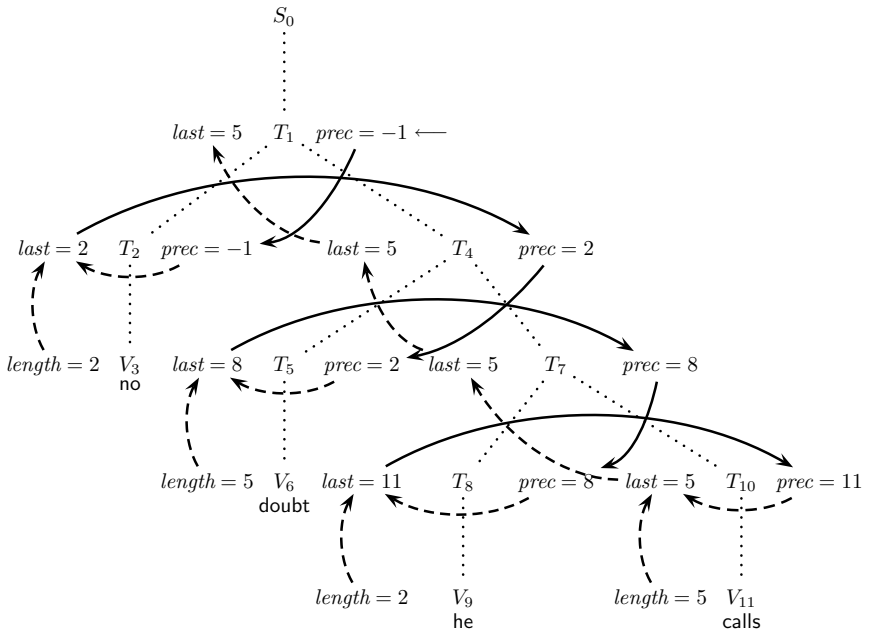


Fig. 5.14 Decorated tree with dependence graph for Example 5.35

An important quality of the attribute evaluation process is that the result is independent of the application order of functions. This property holds for grammars complying with certain conditions, to be considered soon.

Usefulness of Right Attributes

The grammar in Table 5.4 uses both left and right attributes, so the questions arise: can we define the same semantics without using right attributes (as we did in the introductory Example 5.3)? And then, is it convenient to do so?

The first answer is yes in general, and we show it for Example 5.35. The position of the last letter of a word can be computed by a different approach. Initially compute the left attribute *length*, and then construct a new left attribute *list*, having as domain a list of integers representing word lengthes. For the tree in Fig. 5.14, node T_7, which covers the text he calls, would have the attribute $list = \langle\, 2, 5 \,\rangle$. After processing all the nodes, the value *list* in the subroot T_1 of the tree is available, $list = \langle\, 2,\ 5,\ 2,\ 5 \,\rangle$. It is then straightforward, knowing the page width W, to compute the position of the last character of each word.

But this solution is fundamentally bad, because the computation by the semantic function at the root of the tree has essentially the same complexity as the original text breaking problem: nothing has been gained by the syntax-directed approach, since the original problem has not been decomposed into simpler subproblems.

Another drawback is that information is now concentrated in the root, rather than distributed on all the nodes by means of the right attribute *last*, as it was in the grammar in Table 5.4.

Finally, to counterbalance the suppression of right attributes, it is often necessary to introduce non-scalar attributes, such as lists or sets, or other complex data structures.

In conclusion, when designing an attribute grammar the most elegant and effective design is often obtained relying on both left and right attributes.

5.6.3 Definition of Attribute Grammar

It is time to formalize the concepts introduced by previous examples. This is done in the next definition.

Definition 5.36 (*attribute grammar*) An *attribute grammar* is defined as follows.

1. A context-free syntax $G = (V, \Sigma, P, S)$, where V and Σ are the terminal and nonterminal sets, P the production rule set and S the axiom. It is convenient to avoid the presence of the axiom in the right parts of productions.
2. A set of symbols, the (semantic) *attributes*, associated with nonterminal and terminal syntax symbols. The set of the attributes associated with symbol, e.g., D, is denoted $attr(D)$.
   The attribute set of a grammar is partitioned into two disjoint subsets, the *left attributes* and the *right attributes*.
3. Each attribute σ has a *domain*, the set of values it may take.
4. A set of *semantic functions* (or rules). Each function is associated with a production rule:

$$p: \; D_0 \rightarrow D_1 \; D_2 \; \ldots \; D_r \qquad r \geq 0$$

   where D_0 is a nonterminal and the other symbols can be terminal or nonterminal. The production p is the syntactic *support* of the function. In general, several functions may have the same support.
   Notation: the attribute σ associated with a symbol D_k is denoted by σ_k or also by σ_D if the syntactic symbol occurs exactly once in production p.
   A semantic function has the form:

$$\sigma_k := f \; (attr \; (\{ \; D_0, \; D_1, \; \ldots, \; D_r \; \}) \setminus \{ \; \sigma_k \; \})$$

   where $0 \leq k \leq r$; function f assigns to the attribute σ of symbol D_k, the value computed by the function body; the arguments of f can be any attributes of the same production p, excluding the result of the function.
   Usually, the semantic functions are total functions in their domains. They are written in a suitable notation, termed *semantic metalanguage*, such as a programming language or a higher-level specification language, which can be formal or informal as a pseudo-code.
   A function $\sigma_0 := f \; (\ldots)$ defines an attribute, qualified as *left*, of the nonterminal D_0, which is the left part (or father or parent) of the production.

A function $\sigma_k := f\ (\ldots)$ with $k \geq 1$ defines an attribute, qualified as *right*, of a symbol (sibling or child) D_k occurring in the right part.
It is forbidden (as stated in 2.) for the same attribute to be left in a function and right in another one.
Notice that since terminal characters never occur in the left part, their attributes cannot be of the left type.[20]

5. Consider the set $fun\,(p)$ of all the functions supported by production p. They must satisfy the following conditions:

   a. For each left attribute σ_0 of D_0, there exists in $fun\,(p)$ exactly one function defining the attribute.
   b. For each right attribute δ_0 of D_0, no function exists in $fun\,(p)$ defining the attribute.
   c. For each left attribute σ_i, where $i \geq 1$, no function exists in $fun\,(p)$ defining the attribute.
   d. For each attribute δ_i, where $i \geq 1$, there exists in $fun\,(p)$ exactly one function defining the attribute.

   The left attributes σ_0 and the right ones δ_i with $i \geq 1$ are termed *internal* for production p, because they are defined by functions supported by p.
   The right attributes δ_0 and left attributes σ_i with $i \geq 1$, are termed *external* for production p, because they are defined by functions supported by other productions.
6. Some attributes can be initialized with constant values or with values computed by external functions. This is often the case for the so-called lexical attributes, those associated with terminal symbols. For such attributes the grammar does not specify a computation rule. □

Example 5.37 (*Example* 5.35 *on p.* 432 *continued*) We refer again to the grammar in Table 5.4 on p. 433, where the attributes are classified as follows:

left attributes	*length* and *last*
right attributes	*prec*
internal/external	for production 2 the internal attributes are $prec_1$, $prec_2$ and $last_0$; the external ones are $prec_0$ and $last_2$ (attribute *length* is not pertinent to production 2)

Then we have:

$$attr\,(T) = \{\ prec,\ last\ \} \qquad attr\,(V) = \{\ length\ \} \qquad attr\,(S) = \emptyset$$

[20] In practice, the attributes of terminal symbols are often not defined by the semantic functions of the grammar, but are initialized with values computed during lexical analysis, which is the scanning process that precedes parsing and semantic analysis.

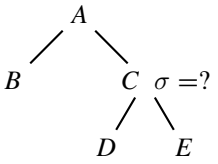
We warn the reader against misuse of not local attributes. Item 4 of Definition 5.36 expresses a sort of *principle of locality* of semantic functions: it is an error to designate as argument or result of a semantic function supported by p, an attribute which is not pertinent to production p. An instance of such error occurs in the modified rule 2 below:

#	*syntax*	*semantic functions*
1	$S_0 \to T_1$	...
2	$T_0 \to T_1 \perp T_2$	$prec_1 := prec_0 + \underbrace{length_0}_{\text{non-pertinent attribute}}$
3	...	...

Here the principle of locality is violated because $length \notin attr(T)$: any attribute of a node, other than the father or a sibling, is out of scope. □

The rationale of the condition that left and right attributes are disjoint sets is discussed next. Each attribute of a node of the syntax tree must be defined by exactly one assignment; otherwise it would take two or more different values depending on the order of evaluation, and the meaning of the tree would not be unique. To prevent this, the same attribute may not be left and right, because in that case there would be two assignments, as shown in the fragment:

#	*support*	*semantic functions*
1	$A \to B\ C$	$\sigma_C := f_1\,(attr(A,\ B))$
2	$C \to D\ E$	$\sigma_C := f_2\,(attr(D,\ E))$



Clearly variable σ_C, internal for both productions, is a right attribute in the former, a left one in the latter. Therefore the final value it takes will depend on the order of function applications. Then the semantics loses the most desirable property of being independent of the implementation of the evaluator.

5.6.4 Dependence Graph and Attribute Evaluation

An advantage of a grammar as a specification of a translation is that it abstracts from the details of tree-traversing procedures. In fact, the attribute evaluation program can be automatically constructed from the functional dependencies between attributes, of course supposing that the bodies of the semantic functions are given.

To prepare for such construction, we formalize the functional dependencies, within three progressively more comprehensive scopes, by means of the following directed graphs:

dependence graph of a semantic function
: the nodes of this graph are the arguments and result of the function considered, and there is an arc from each argument to the result

dependence graph of a production p
: this graph, denoted by dep_p, collects the dependence graphs for all the functions supported by the production considered

dependence graph of a decorated syntax tree
: this graph already introduced (see Fig. 5.14 on p. 434) is obtained by pasting together the graphs of the individual productions that are used in the tree nodes.

The next example (Example 5.38) shows a case.

Example 5.38 (*dependence graphs*) We reproduce from p. 433 into Fig. 5.15 the grammar of Example 5.35. For clarity we lay each graph over the supporting production (dotted edges), to evidence the association between attributes and syntactic components. Production 2 is the most complex, with three semantic functions, each one with just one argument, hence one arc (visually differentiated by the style) of the graph. Notice, in the dependence graph of each production, that a node with (respectively without) incoming arcs is an attribute of type internal (respectively external).

The dependence graph of a (decorated) syntax tree was already shown in Fig. 5.14 on p. 434. □

5.6.4.1 Attribute Values as Solution of Equations

We expect each sentence of a technical language to have exactly one meaning, i.e., a unique set of values assigned to the semantic attributes; otherwise we would be faced with an undesirable case of semantic ambiguity.

We know that the values are computed by assignments and that there is exactly one assignment per attribute instance in the tree. We may view the set of assignments as a system of simultaneous equations, where the unknowns are the attribute values. From this perspective, the solution of the system is the meaning of the sentence.

For a sentence, consider now the attribute dependence graph of the tree, and suppose it contains a directed path:

$$\sigma_1 \to \sigma_2 \to \ldots \to \sigma_{j-1} \to \sigma_j \qquad \text{with } j > 1$$

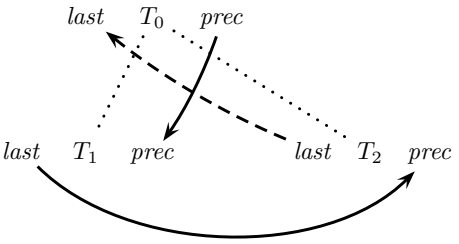
where each σ_k stands for some attribute instance. The names of the attribute instances can be the same or different. The corresponding equations are as follows:

$$\begin{aligned} \sigma_j &= f_j\ (\ldots,\ \sigma_{j-1},\ \ldots) \\ \sigma_{j-1} &= f_{j-1}\ (\ldots,\ \sigma_{j-2},\ \ldots) \\ &\ldots \\ \sigma_2 &= f_2\ (\ldots,\ \sigma_1,\ \ldots) \end{aligned}$$

syntax support and semantic functions

#	*syntax*	*right attributes*	*left attributes*
1	$S_0 \to T_1$	$prec_1 := -1$	
2	$T_0 \to T_1 \perp T_2$	$prec_1 := prec_0$ $prec_2 := last_1$	$last_0 := last_2$
3	$T_0 \to V_1$		$last_0 := \textbf{if } (prec_0 + 1 + length_1) \leq W$ $\textbf{then } (prec_0 + 1 + length_1)$ $\textbf{else } length_1$ $\textbf{end if}$
4	$V_0 \to c\, V_1$		$length_0 := length_1 + 1$
5	$V_0 \to c$		$length_0 := 1$

dependence graph dep_2 of production 2



dependence graphs of the remaining productions

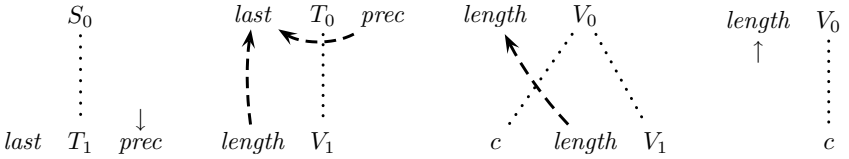


Fig. 5.15 Grammar of Example 5.35 and dependence graphs of productions

since the result of a function is an argument of the next one.

For instance, in Fig. 5.14 on p. 434 one such path is the following:

$$prec\,(T_1) \to prec\,(T_2) \to last\,(T_2) \to prec\,(T_4) \to prec\,(T_5) \to \ldots$$

Revisiting the previous examples of decorated trees, it would be easy to verify that, for any sentence and syntax tree, no path of the dependence graph ever makes a circuit, to be formalized next.

A *grammar* is *acyclic* if, for each sentence, the dependence graph of the tree[21] is acyclic. The next property (Property 5.39) states when a grammar can be considered correct, of course disregarding any errors in the function bodies.

Property 5.39 (correct attribute grammar) *Given an attribute grammar satisfying the conditions of Definition 5.36, consider any syntax tree. If the attribute dependence graph of the tree is acyclic, the system of equations corresponding to the semantic functions has exactly one solution.* □

To prove the property, we show that, under the acyclicity condition, the equations can be ordered in such a way that each semantic function is applied after the functions that compute its arguments. This produces a value for the solution, since the functions are total. The solution is clearly unique, as in a classical system of simultaneous linear equations.

Let $G = (V, E)$ be an acyclic directed graph, and identify the nodes by numbers $V = \{1, 2, \ldots, |V|\}$. The next algorithm (Algorithm 5.40) computes a total order of nodes, called *topological*. The result $ord\,[1 \ldots |V|]$ is the vector of sorted nodes: it gives the identifier of the node that has been assigned to the ith position ($1 \leq i \leq |V|$) in the ordering.

Algorithm 5.40 (*topological sorting*)

```
begin
    i := 0                          - - initialize node counter i
    while V ≠ ∅ do
        n := any node of set V without incoming arcs
        - - notice: such a node exists as graph G is acyclic
        i + +                       - - increment node counter i
        ord[i] := n                 - - put node n into vect. ord at pos. i
        V := V \ { n }              - - remove node n from set V
        - - remove the dangling arcs from set E
        E := E \ { arcs going out from node n }
    end while
end  □
```

[21] We assume as usual that the parser returns exactly one syntax tree per sentence.

In general, many different topological orders are possible, because the dependence graph typically does not enforce a total order relation.

Example 5.41 (*topological sorting*) Applying the algorithm to the graph of Fig. 5.14 on p. 434, we obtain a topological order:

$$length_3,\ length_6,\ length_9,\ length_{11},\ prec_1,\ prec_2,\ last_2,\ prec_4,$$

$$prec_5,\ last_5,\ prec_7,\ prec_8,\ last_8,\ prec_{10},\ last_{10},\ last_7,\ last_4,\ last_1 \qquad \square$$

Next, we apply the semantic functions in topological order. Pick the first node; its equation is necessarily constant; i.e., it initializes the result attribute. Then proceed by applying the next equations in the order, which guarantees availability of all arguments. Since all the functions are total, a result is always computed. The tree is thus progressively decorated with a unique set of values. Therefore for an acyclic grammar the meaning of a sentence is a single-valued function.

Actually the above evaluation algorithm is not very efficient, because on one hand it requires computing the topological sort, and on the other hand it may require multiple visits of the same node of the syntax tree. We are going to consider more efficient algorithms, although less general, which operate under the assumption of a fixed order of visit (scheduling) of the tree nodes.

Now consider what happens if the dependence graph of a tree contains a path that makes a circuit, implying that in the chain:

$$\sigma_1 \to \sigma_2 \to \ldots \to \sigma_{j-1} \to \sigma_j \qquad \text{with } j > 1$$

two elements i and k with $1 \leq i < k \leq j$ are identical, i.e., $\sigma_i = \sigma_k$. Then the system of equations may have more than one solution, and the grammar may be semantically ambiguous.

A remaining problem is how to check whether a given grammar is acyclic: how can we be sure that no decorated syntax tree will ever present a closed dependence path? Since the source language is usually infinite, the acyclicity test cannot be performed by the exhaustive enumeration of the trees. An algorithm to decide if an attribute grammar is acyclic exists but is complex[22] and not used in practice. It is more convenient to test certain sufficient conditions, which not only guarantee the acyclicity of a given grammar, but also permit constructing the attribute evaluation schedule, to be used by the semantic analyzer. Some simple yet practical conditions are described next.

[22] See Knuth [8,10]. The asymptotic time complexity is *NP*-complete with respect to the size of the attribute grammar.

5.6.5 One-Sweep Semantic Evaluation

A fast evaluator should compute the attributes of each tree node with a single visit or at worst with a small number of visits of the nodes. A well-known visit order of a tree is the *depth-first* traversal, which in many cases permits the evaluation of the attributes with just one sweep over the tree.

Let N be a node of a tree and $N_1, \ldots, N_r$ its child nodes; denote by t_i the subtree rooted in node N_i.

A depth-first algorithm first visits the tree root. Then, in order to visit the generic subtree t_N rooted in a node N, it recursively proceeds as follows. It performs a depth-first visit of the subtrees $t_1, \ldots, t_r$, in an order corresponding to some permutation of $1, 2, \ldots, r$, which not necessarily coincides with the natural order $1, 2, \ldots, r$.

This semantic evaluation algorithm, termed *one-sweep*, computes the attributes according to the following principles:

- Before entering and evaluating a subtree t_N, it computes the right attributes of node N (the root of the subtree).
- At the end of visit of subtree t_N, it computes the left attributes of N.

We hasten to say that not all the grammars are compatible with this algorithm, because more intricate functional dependencies may require several visits of the same node. The appeal of this method is that it is very fast, and that practical, sufficient conditions for one-sweep evaluation are simple to state and to check on the dependence graph dep_p of each production p.

Experience with grammar design indicates it is often possible to satisfy the one-sweep conditions, sometimes with minor changes to the original semantic functions.

5.6.5.1 One-Sweep Grammar

For each production $p\colon D_0 \to D_1\ D_2\ \ldots\ D_r$, with $r \geq 0$, we need to define a binary relation between the syntactic symbols of the right part, to be represented in a directed graph, called the *sibling graph*, denoted $sibl_p$. The idea is to summarize the dependencies between the attributes of the semantic functions supported by the production. The nodes of the sibling graph are the symbols $\{\ D_1,\ D_2,\ \ldots,\ D_r\ \}$ of the production. The sibling graph has an arc:

$$D_i \to D_j \qquad \text{with } i \neq j \text{ and } i,\ j \geq 1$$

if in the dependence graph dep_p there is an arc $\sigma_i \to \delta_j$ from an attribute of symbol D_i to an attribute of symbol D_j. We stress that the nodes of the sibling graph are not the same as those of the dependence graph of the production: the former are syntactical symbols, and the latter are attributes. Clearly, all the nodes (attributes) of dep_p having the same subscript j are coalesced into one node D_j of $sibl_p$: in mathematical terms, the sibling graph is related to the dependence graph by a node homomorphism.

The next definition states the conditions that make a grammar suitable for one-sweep semantic evaluation.

Definition 5.42 (*one-sweep grammar*) A grammar satisfies the one-sweep condition if, for each production $p\colon D_0 \to D_1\, D_2\, \ldots\, D_r$ (with $r \geq 0$) that has a dependence graph dep_p, the following clauses hold:

1. Graph dep_p contains no circuit.
2. Graph dep_p does not contain a path:

$$\lambda_i \to \ldots \to \rho_i \qquad \text{with } i \geq 1$$

   that goes from a left attribute λ_i to a right attribute ρ_i of the same symbol D_i, where D_i is a sibling.
3. Graph dep_p contains no arc $\lambda_0 \to \rho_i$, with $i \geq 1$, from a left attribute of the father node D_0 to a right attribute of a sibling node D_i.
4. The sibling graph $sibl_p$ contains no circuit. □

We orderly explain each item:

1. This condition is necessary for the grammar to be acyclic (a requirement for ensuring existence and uniqueness of meaning).
2. If we had a path $\lambda_i \to \ldots \to \rho_i$, with $i \geq 1$, it would be impossible to compute the right attribute ρ_i before visiting subtree t_i, because the value of the left attribute λ_i is available only after the visit of the subtree. This contravenes the depth-first visit order we have opted for.
3. As in the preceding item, the value of attribute ρ_i would not be available when we start visiting the subtree t_i.
4. This condition permits to topologically sort the child nodes, i.e., the subtrees $t_1, \ldots, t_r$, and to schedule their visit in an order consistent with the precedences expressed by graph dep_p. If the sibling graph had a circuit, there would be conflicting precedence requirements on the order of visiting the sibling subtrees. In that case it would be impossible to find a schedule valid for all the attributes of the right part of production p.

Algorithm 5.43 (*construction of one-sweep evaluator*) We write a semantic procedure for each nonterminal symbol, having as arguments the subtree to be decorated and the right attributes of its root. The procedure visits the subtrees, and computes and returns the left attributes of the root (of the subtree).

For each production $p\colon\ D_0 \to D_1\, D_2\, \ldots\, D_r$ with $r \geq 0$:

1. Choose a topological order, denoted *TOS*, of the nonterminals $D_1, D_2, \ldots, D_r$ with respect to the sibling graph $sibl_p$.
2. For each symbol D_i, with $1 \leq i \leq r$, choose a topological order, denoted *TOR*, of the right attributes of symbol D_i with respect to the dependence graph dep_p.

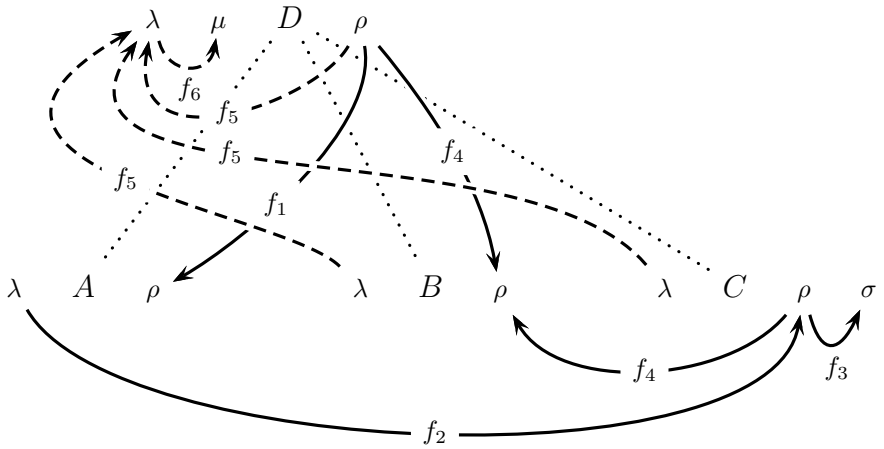


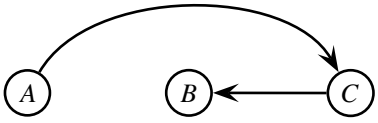
Fig. 5.16 Dependence graph *dep* of rule $D \to A\ B\ C$ for Example 5.44. The semantic function names are placed on dependence arcs

3. Choose a topological order, denoted *TOL*, of the left attributes of symbol D_0, with respect to the dependence graph dep_p.

The three orders *TOS*, *TOR* and *TOL* together prescribe how to arrange the instructions in the body of the semantic procedure, to be illustrated in the coming example (Example 5.44). □

Example 5.44 (*one-sweep semantic procedure*) For brevity we consider a grammar fragment containing just one production and we leave the bodies of the semantic functions unspecified. Production $D \to A\ B\ C$ has the dependence graph *dep* shown in Fig. 5.16. It is straightforward to check that the graph satisfies conditions 1, 2 and 3 of Definition 5.42, because:

1. there are neither circuits
2. nor any path from a left attribute λ_A, λ_B or λ_C to a right attribute, such as ρ_B, of the same node
3. nor any arc from a left attribute λ_D or μ_D to a right one of A, B or C
4. the sibling graph *sibl*, below, is acyclic:



We explain where its arcs come from: $A \to C$ comes from dependence $\lambda_A \to \rho_C$ and $C \to B$ from dependence $\rho_C \to \rho_B$. Next, we compute the topological orders:

- Sibling graph: $TOS = A,\ C,\ B$.
- Right attributes of each sibling: since A and B have only one right attribute, the topological sorting is trivial; for C we have $TOR = \rho,\ \sigma$.
- Left attributes of D: the topological order is $TOL = \lambda,\ \mu$.

To complete the design, it remains to list the instructions of the semantic procedure of this production, in an order compatible with the chosen topological orders. More precisely, the order of left attribute assignments is *TOL*, the order of procedure invocations (to evaluate subtrees) is *TOS*, and the order of right attribute assignments is *TOR*.

The final semantic procedure follows:

procedure D (**in** $t,\ \rho_D$; **out** $\lambda_D,\ \mu_D$)
- - t root of subtree to be decorated
$\rho_A := f_1\,(\,\rho_D\,)$
- - abstract functions are denoted $f_1,\ f_2$, etc.
$A\,(\,t_A,\ \rho_A;\ \lambda_A\,)$
- - invocation of A to decorate subtree t_A
$\rho_C := f_2\,(\,\lambda_A\,)$
$\sigma_C := f_3\,(\,\rho_C\,)$
$C\,(t_C,\ \rho_C,\ \sigma_C;\ \lambda_C\,)$
- - invocation of C to decorate subtree t_C
$\rho_B := f_4\,(\,\rho_D,\ \rho_C\,)$
$B\,(\,t_B,\ \rho_B;\ \lambda_B\,)$
- - invocation of B to decorate subtree t_C
$\lambda_D := f_5\,(\,\rho_D,\ \lambda_B,\ \lambda_C\,)$
$\mu_D := f_6\,(\,\lambda_D\,)$
end procedure □

To conclude, this method is very useful for designing an efficient recursive semantic evaluator, provided the grammar satisfies the one-sweep condition.

5.6.6 Other Evaluation Methods

One-sweep evaluation is practical, but some grammars have complicated dependencies that prevent its use. More general classes of evaluators and corresponding grammar conditions are available, which we do not discuss.[23] To expand the scope of the one-sweep evaluation methods, we develop a rather intuitive idea. The evaluation process is decomposed into a cascade of two or more phases, each one of the one-sweep type, which operate on the same syntax tree, but on distinct sets of attributes.

We describe the method focusing on two phases, but generalization is straightforward. The attribute set *Attr* of the grammar is partitioned by the designer into two disjoint sets $Attr_1 \cup Attr_2 = Attr$, to be, respectively, evaluated in phase one and two. Each attribute set, together with the corresponding semantic functions, can be viewed as an attribute *subgrammar*.

Next, we have to check that the first subgrammar satisfies the general conditions of Definition 5.36 (p. 435) as well as the one-sweep condition of Definition 5.42 (p. 443). In particular, the general condition imposes that every attribute is defined by some semantic function; as a consequence no attribute of set $Attr_1$ may depend on any attribute of set $Attr_2$; otherwise it would be impossible to evaluate the former attribute in phase one.

Then, we construct the one-sweep semantic procedures for phase one, exactly as we would have done for a one-sweep grammar (as in Example 5.44 above). After the execution of phase one of the semantic analyzer, all the attributes in set $Attr_1$ have a value, and it remains to evaluate the attributes of the second set.

For phase two, we have again to check whether the second subgrammar meets the same conditions. Notice, however, that for the second evaluator, the attributes of set $Attr_1$ are considered as initialized constants. This means that the dependencies between two elements of $Attr_1$, and between an element of $Attr_1$ and an element of $Attr_2$, are disregarded when checking the conditions. In other words, only the dependencies inside set $Attr_2$ need to be considered. The phase two evaluator operates on a tree decorated with the attributes of the first set and computes the remaining attributes in one sweep.

The crucial point for multi-sweep evaluation to work is to find a good partition of the attributes into two (or more) sets. Then, the construction works exactly as in one sweep. Notice that not all the attribute values computed in phase one have to be stored in the decorated tree produced by phase one, but only those used as arguments by semantic functions are applied in the second sweep.

As a matter of fact, designing a semantic evaluator for a rich technical language is a complex task, and it is often desirable to modularize the project in order to master the difficulty; some forms of modularization are discussed in [13]. Partitioning the

[23] Among them we mention the evaluators based on multiple visits, on the ordered attribute grammar (*OAG*) condition and on the absolute acyclicity condition. A survey of evaluation methods and corresponding grammar conditions is in Engelfriet [11,12].

global attribute set into subsets associated with evaluation phases offers a precious help for modularization. In practice, in many compilers the semantic analyzer is subdivided into phases of smaller complexity. For instance, the first stage analyzes the declarations of the various program entities (variables, types, classes, etc.) and the second stage processes the executable instructions of the programming language to be compiled.

5.6.7 Combined Syntax and Semantic Analysis

For faster processing, it is sometimes possible and convenient to combine syntax tree construction and attribute computation, trusting the parser with the duty to invoke the semantic functions.

In the following discussion we use pure *BNF* syntax for the support, because the use of *EBNF* productions makes it more complicate to specify the correspondence between syntax symbols and attributes.

There are three typical situations for consideration, depending on the nature of the source language:

- The source language is regular: lexical analysis with lexical attributes.
- The source syntax is *LL* (*k*): recursive descent parser with attributes.
- The source syntax is *LR* (*k*): shift-reduce parser with attributes.

Next we discuss the enabling conditions for such combined syntax–semantic processors.

5.6.7.1 Lexical Analysis with Attribute Evaluation

The task of a *lexical analyzer* (or *scanner*) is to segment the source text into the lexical elements, called *lexemes* or tokens, such as identifiers, integer or real constants and comments. Lexemes are the smallest substrings that can be invested with some semantic property. For instance, in many languages the keyword *begin* has the property of opening a compound statement, whereas its substring *egin* has no meaning.

Each technical language uses a finite collection of *lexical classes*, as the ones just mentioned. A lexical class is a regular formal language: a typical example is the class of identifiers, defined by the regular expression of Example 2.27 on p. 29. A lexeme of an identifier class is a sentence belonging to the corresponding regular language.

In language reference manuals, we find two levels of syntactic specifications: from lower to higher, the *lexical* and *syntactic* levels. The former defines the form of the lexemes. The latter assumes the lexemes are given in the text, and considers them to be the characters of the terminal alphabet of its syntax. Moreover, some lexemes may carry a meaning, i.e., a semantic attribute, which is computed by the scanner.

Lexical Classes

Focusing on typical lexicons, we notice that some lexical classes, viewed as formal languages, have a finite cardinality. Thus, the reserved keywords of a programming language make a finite or *closed* class. An example is the class of keywords, including:

$$\{\ \mathsf{begin},\ \mathsf{end},\ \mathsf{if},\ \mathsf{then},\ \mathsf{else},\ \mathsf{do},\ \ldots,\ \mathsf{while}\ \}$$

Similarly, the number of arithmetic, boolean and relational operation signs is finite. On the contrary, identifiers, integer constants and comments are cases of *open* lexical classes, having unbounded cardinality.

A scanner is essentially a finite transducer or *IO*-automaton (see Sect. 5.5.2 on p. 421). Its task is to divide the source text into lexemes and to assign an encoding to each one. In the source text the lexemes are separated, depending on their classes, by blank spaces or delimiters such as `new-line`. The transducer returns the encoding of each lexeme and removes the delimiters.

More precisely, the scanner transcribes each lexeme into the target string as a pair of elements: the name, i.e., the encoding of the lexeme, and another semantic attribute, termed *lexical*.

Lexical attributes change from a class to another and are altogether missing from certain classes.

Some typical cases are:

integer const	the attribute is the value of the constant in base ten
identifier	the attribute is a key, to be used by the compiler for quickly locating the identifier in a symbol table
comment	a comment has no attribute, if the compilation framework does not manage program documentation, and throws away source program comments; if the compiler keeps and classifies comments, their lexical attribute is instrumental to retrieve them
keyword	has no semantic attribute, just a code for identification

Unique Segmentation

In a well-designed technical language, lexical definitions should ensure that, for any source text, the segmentation into lexemes is unique. A word of caution is necessary, for the concatenation of two or more lexical classes may introduce ambiguity. For instance, string *beta*237 can be divided into many ways into valid lexemes: a lexeme *beta* of class *identifier* followed by 237 of class *integer*, or an identifier *beta*2 followed by the integer 37, and so on.

In practice, this sort of concatenation ambiguity (p. 58) is often cured by imposing to the scanner the *longest prefix rule*. The rule tells the scanner to segment a string $x = u\,v$ into the lexemes, say, $u \in$ *identifier* and $v \in$ *integer*, in such a way that u is the longest prefix of x belonging to class *identifier*. In the example, the rule assigns the whole string *beta*237 to class *identifier*.

By this prescription, the translation is made single-valued and can be computed by a finite deterministic transducer, augmented with the actions needed to evaluate the lexical semantic attributes.

Lexical Attributes

We have observed that some lexical classes carry a semantic attribute and different classes usually have different attribute domains. Therefore, at a first glance it would seem necessary to differentiate the semantic functions for each lexical class. However, it is often preferable to unify the treatment of lexical attributes as far as possible, in order to streamline the scanner organization: remember that a scanner has to be efficient because it is the innermost loop of the compiler. To this end, each lexical class is assigned the same attribute of type string, named *ss*, which contains the substring recognized as lexeme by the scanner.

For instance, the attribute *ss* of identifier *beta*237 is just a string "*beta*237" (or a pointer thereto). Then the finite transducer returns the translation:

$$\langle \text{class} = \textit{identifier},\ \text{ss} = \text{`}\textit{beta}237\text{'} \rangle$$

upon recognizing lexeme *beta*237. This pair clearly contains sufficient information, to pass as argument to a later invocation of an identifier-specific semantic function. The latter looks up string *ss* in the symbol table of the program under compilation; if it is not present, it inserts the string into the table and returns its position as a semantic attribute. Notice such identifier-specific semantic function is better viewed as a part of the attribute grammar of the syntax-directed translator, rather than of the scanner.

5.6.7.2 Attributed Recursive Descent Translator

Assume the syntax is suitable for deterministic top-down parsing. Attribute evaluation can proceed in lockstep with parsing if the functional dependencies of the grammar obey certain additional conditions beyond the one-sweep ones.

We recall that the one-sweep algorithm (Definition 5.42 on p. 443) visits in depth-first order the syntax tree, traversing the subtrees $t_1, \ldots, t_r$, for the current production $D_0 \rightarrow D_1 \ \ldots \ D_r$ in an order that may be different from the natural one. The order is a topological sorting, consistent with the dependencies between the attributes of nodes $1, \ldots, r$.

On the other hand, we know that a top-down parser constructs the tree in the natural order; i.e., subtree t_j is constructed after subtrees $t_1, \ldots, t_{j-1}$. It follows that, to combine the two processes, we must exclude any functional dependence that would enforce an attribute evaluation order other than the natural one, as stated next (Definition 5.45).

Definition 5.45 (*L-condition*)] A grammar satisfies the condition named L[24] if, for each production $p: D_0 \rightarrow D_1 \ldots D_r$, it holds:

1. The one-sweep condition 5.42 (p. 443) is satisfied.
2. The sibling graph $sibl_p$ contains no arc $D_j \rightarrow D_i$ with $j > i \geq 1$. □

Notice the second clause prevents a right attribute of node D_i to depend on any (left or right) attribute of a node D_j placed to its right in the production. As a consequence, the natural order 1, …, r is a topological order of the sibling graph and can be applied to visit the sibling subtrees. The next property relates the L-condition and deterministic parsing.

Property 5.46 (attribute grammar and deterministic parsing) *Let a grammar be such that:*

- *The syntax satisfies the* LL (1) *or, more generally, the* LL (k) *condition (see p. 322).*
- *The semantic rules satisfy the L-condition.*

Then, it is possible to construct a top-down deterministic parser with attribute evaluation, to compute the attributes at parsing time. □

The construction of the semantic evaluator, presented in the coming example (Example 5.47), is a straightforward combination of a recursive descent parser and an one-sweep recursive evaluator.

Example 5.47 (*recursive descent parser with attribute evaluation*) Revising Example 5.34 (p. 429), we write a grammar to convert a fractional number smaller than 1, from base two to base ten. The source language is defined by the regular expression below:

$$L = \bullet \ (0 \mid 1)^*$$

The meaning of a string, say, • 0 1 is the decimal number 0.25. The grammar is listed in Table 5.5. Notice that the value of a bit is weighted by a negative exponent, equal to its distance from the fractional point.

The syntax is deterministic *LL* (2), as it can be checked. Next we verify condition L, production by production.

$N \rightarrow \bullet \ D$ The dependence graph has one arc $v_1 \rightarrow v_0$, hence:
- There are no circuits in the graph.
- There is no path from left attribute v to right attribute l of the same sibling.
- In the graph there is no arc from attribute v of father to a right attribute l of a sibling.
- The sibling graph *sibl* has no arcs.

[24] The letter L stands for left to right.

Table 5.5 Grammar for converting fractional numbers (Example 5.47)

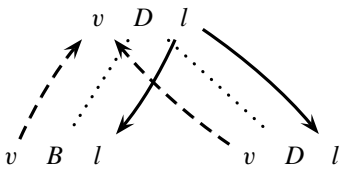
Grammar

Syntax	Left attributes	Right attributes
$N_0 \rightarrow \bullet\ D_1$	$v_0 := v_1$	$l_1 := 1$
$D_0 \rightarrow B_1\ D_2$	$v_0 := v_1 + v_2$	$l_1 := l_0 \quad l_2 := l_0 + 1$
$D_0 \rightarrow B_1$	$v_0 := v_1$	$l_1 := l_0$
$B_0 \rightarrow 0$	$v_0 := 0$	
$B_0 \rightarrow 1$	$v_0 := 2^{-l_0}$	

Attributes

Attribute	Meaning	Domain	Type	Assoc. symbols
v	Value	Real	Left	N, D, B
l	Length	Integer	Right	D, B

$D \rightarrow B\ D$ The dependence graph:



– has no circuit.
– has no path from left attribute v to right attribute l of the same sibling.
– has no arc from the left attribute v of the father to the right attribute v of a sibling.
– The sibling graph has no arcs.

$D \rightarrow B$ Same as above.
$B \rightarrow 0$ The dependence graph has no arcs.
$B \rightarrow 1$ The dependence graph has one arc $l_0 \rightarrow v_0$, which is compatible with one sweep, and there are not any brothers.

Similar to a parser, the program comprises three procedures N, D and B, having as their arguments the left attributes of the father node. To implement parser look-ahead, a procedure uses two variables to store the current character $cc1$ and the next

one $cc2$. Function *read* updates both variables. Variable $cc2$ determines the choice between the syntactic alternatives of D.

```
procedure N ( in ∅; out v0 )
    if cc1 = '•' then
        read
    else
        error
    end if
    l1 := 1          - - initialize a local var. with right attribute of D
    D ( l1, v0 )     - - call D to construct a subtree and compute v0
end procedure

procedure D ( in l0; out v0 )
    case cc2 of
        '0', '1' :  begin          - - case of rule D → B D
                        B ( l0, v1 )
                        l2 := l0 + 1
                        D ( l2, v2 )
                        v0 := v1 + v2
                    end
        '⊣' :       begin          - - case of rule D → B
                        B ( l0, v1 )
                        v0 := v1
                    end
        otherwise error
    end case
end procedure

procedure B ( in l0; out v0 )
    case cc1 of
        '0' :  v0 := 0          - - case of rule B → 0
        '1' :  v0 := 2^(-l0)    - - case of rule B → 1
        otherwise error
    end case
end procedure
```

To activate the analyzer, the compiler invokes the axiom procedure. □

Clearly, a skilled programmer could improve in several ways the previous schematic implementation.

5.6.7.3 Attributed Bottom-Up Parser

Supposing the syntax meets the *LR* (1) condition, we want to combine bottom-up syntax tree construction with attribute evaluation. Some problems have to be addressed: how to ensure that precedences on semantic function calls induced by attribute dependencies are consistent with the order of tree construction; when to compute the attributes; and where to store their values.

Considering first the problem of when semantic functions should be invoked, it turns out that *right* attributes cannot be evaluated during parsing, even assuming the grammar complies with the *L*-condition of Definition 5.45 (that was sufficient for top-down evaluation in one sweep). The reason is that a shift-reduce parser defers the choice of the production until it has to perform a reduction, when the parser is in a macro-state containing a final state of a machine (or, what is the same, the marked production is $D_0 \rightarrow D_1 \ \ldots \ D_r \bullet$). This is the earliest time the parser can choose the semantic functions to be invoked.

The next problem comes from attribute dependencies. Just before reduction, the parser stack contains r elements from top, which correspond to the syntactic symbols of the production right part. Assuming the values of all the attributes of $D_1 \ \ldots \ D_r$ are available, the algorithm can invoke the functions and return the values of the left attributes of D_0.

But there is a difficulty for the evaluation of the right attributes of $D_1 \ \ldots \ D_r$. Imagine that the algorithm is about to construct and decorate the subtree of D_1. To be in accordance with the one-sweep evaluation, every right attribute ρ_{D_1} should be available before evaluating the subtree rooted at D_1. But attribute ρ_{D_1} may depend on some right attribute ρ_0 of the father D_0, which is unavailable, because the syntax tree does not yet contain the upper part, including the node associated with the father. The simplest way to circumvent this obstacle is to assume the grammar does not use right attributes. This ensures that the left attributes of a node will only depend on the left attributes of the children, which are available at reduction time.

Coming to the question of memorization, the attributes can be stored in the stack, next to the items (m-states of the pilot machine) used by the parser. Thus, each stack element is a record, made of a syntactic field and one or more semantic fields containing attribute values (or perhaps pointers to values stored elsewhere); see the next example (Example 5.48).

Example 5.48 (*calculating machine without right attributes*) The syntax for certain arithmetic expressions is shown as a network in Fig. 5.17, and its pilot graph is shown in Fig. 5.18. In machine M_E the final states of the alternative rule paths $E \rightarrow T$ and $E \rightarrow E \ + \ T$ are unified and similarly in M_T. This syntax is *BNF*, and the pilot meets

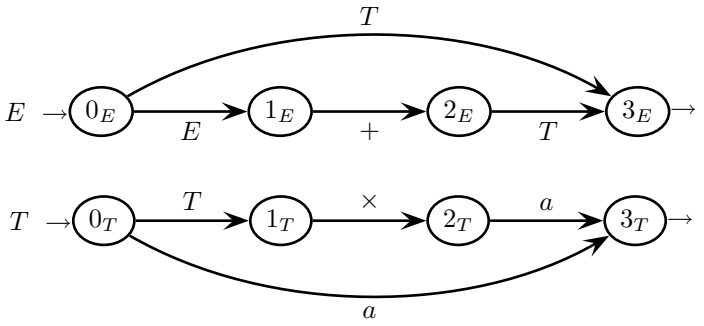


Fig. 5.17 Machine net $\{ M_E, M_T \}$ for the arithmetic expressions (Example 5.48)

the $LR(1)$ condition.[25] The following attribute grammar computes the expression value v or sets to true a predicate o in the case of overflow. Constant *maxint* is the largest integer the calculator can represent. A character a has an initialized attribute v with the value of the integer constant a. Both attributes are left:

syntax	*semantic functions*
$E_0 \to E_1 \ + \ T_2$	$o_0 := o_1$ **or** $(v_1 + v_2 > maxint)$ $\quad v_0 :=$ **if** o_0 **then** nil **else** $(v_1 + v_2)$
$E_0 \to T_1$	$o_0 := o_1 \ v_0 := v_1$
$T_0 \to T_1 \ \times \ a$	$o_0 := o_1$ **or** $(v_1 \times value\,(a) > maxint)$ $v_0 :=$ **if** o_0 **then** nil **else** $(v_1 \times value\,(a))$
$T_0 \to a$	$o_0 := false \ v_0 := value\,(a)$

We trace in Fig. 5.19 a computation of the pushdown machine, extended with the semantic fields. The source sentence is $a_3 + a_5$, where the subscript of a constant is its value. When the parser terminates, the stack contains attributes v and o of the root of the syntax tree. □

Right Attributes Independent of Father

Actually, prohibition to use right attributes may badly complicate the task of writing an attribute grammar. Attribute domains and semantic functions may turn less simple and natural, although in principle we know any translation can be specified without using right attributes.

[25] See also Example 5.15 on p. 406, where the source syntax is the same, but the machine graphs are drawn as trees by keeping separate the final states of each alternative; the pilot in Fig. 5.6 on p. 407 has more m-states but recognizes the same language as here.

syntax	*semantic functions*
$E_0 \to E_1 \ + \ T_2$	$o_0 := o_1$ **or** $(v_1 + v_2 > maxint)$ $v_0 :=$ **if** o_0 **then** nil **else** $(v_1 + v_2)$
$E_0 \to T_1$	$o_0 := o_1$ $v_0 := v_1$
$T_0 \to T_1 \ \times \ a$	$o_0 := o_1$ **or** $(v_1 \times value(a) > maxint)$ $v_0 :=$ **if** o_0 **then** nil **else** $(v_1 \times value(a))$
$T_0 \to a$	$o_0 := false$ $v_0 := value(a)$

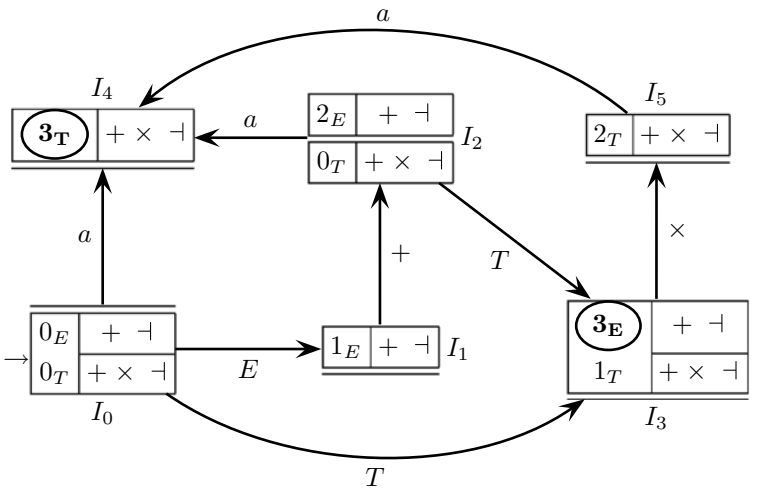


Fig. 5.18 Pilot graph of the arithmetic expressions (Example 5.48 and Fig. 5.17)

Grammar expressivity improves if right attributes are readmitted, although with the following limitation on their dependencies (Definition 5.49).

Definition 5.49 (*A-condition*[26] *for bottom-up evaluation*) For each production $p\colon D_0 \to D_1 \ \ldots \ D_r$, the following must hold:

1. The L-condition (p. 450) for top-down evaluation is satisfied.
2. No right attribute ρ_{D_k} of a sibling depends on a right attribute σ_{D_0} of the father, with $1 \leq k \leq r$ □

[26]The letter *A* stands for ascending order.

stack	*string*								
I_0	a_3		$+$		a_5				$\dashv$
I_0	a_3	I_4	$+$		a_5				$\dashv$
I_0	T $v = 3$ $o = \mathit{false}$	I_3	$+$		a_5				$\dashv$
I_0	E $v = 3$ $o = \mathit{false}$	I_1	$+$		a_5				$\dashv$
I_0	E $v = 3$ $o = \mathit{false}$	I_1	$+$	I_2	a_5				$\dashv$
I_0	E $v = 3$ $o = \mathit{false}$	I_1	$+$	I_2	a_5	I_4			$\dashv$
I_0	E $v = 3$ $o = \mathit{false}$	I_1	$+$	I_2	T $v = 5$ $o = \mathit{false}$	I_3			$\dashv$
I_0	E $v = 3 + 5$ $= 8$ $o = \mathit{false}$								$\dashv$

Fig. 5.19 Shift-reduce parsing trace with attribute evaluation (Example 5.48)

Positively stated, the same condition becomes: a right attribute ρ_{D_k} may only depend on the right or left attributes of symbols $D_1 \ldots D_{k-1}$, with $1 \leq k \leq r$.

If a grammar meets the A-condition, the left attributes of nonterminal symbols $D_1, \ldots, D_r$ are available when a reduction is executed. Thus the remaining attributes can be computed in the following order:

1. Right attributes of the same nonterminal symbols, in the order $1, 2, \ldots, r$
2. Left attributes of father D_0

Notice this order differs from the scheduling of top-down evaluation, in that right attributes are computed later, during reduction.

Finally, we observe that this delayed evaluation gives more freedom to compute the right attributes in an order other than the natural left-to-right order necessarily applied by top-down parsers. This would allow to deal with more involved dependencies between the nodes of the sibling graph (p. 443), similarly to the one-sweep evaluators.

5.6.8 Typical Applications of Attribute Grammar

Syntax-directed translation is widely applied in compiler design. Attribute grammars provide a convenient modular notation for specifying the large number of local operations performed by the compiler, without actually getting into the implementation of the compiler itself. Since it would be too long to describe with some degree of completeness the semantic analysis operations for a programming language, we simply present a few typical interesting parts in a schematic manner. Actually, it is the case that the semantic analysis of programming languages comprises rather repetitive parts, but spelling them out in detail would not add to a conceptual understanding of compilation.

We selected for presentation the following: semantic checks, code generation and the use of semantic information for making parsing deterministic.

5.6.8.1 Semantic Check

The formal language L_F defined by the syntax is just a gross approximation by excess to the actual programming (or more generally technical) language L_T to be compiled; that is, the set inclusion $L_F \supset L_T$ holds. The left member is a context-free language, while the right one is informally defined by the language reference manual. Formally speaking, language L_T belongs to a more complex language family, the context-sensitive one. Without repeating the reasons presented at the end of Chap. 3, a context-sensitive syntax cannot be used in practice, and formalization must be contented with the context-free approximation.

To touch the nature of such approximations, imagine a programming language L_T. The sentences of L_F are syntactically correct, yet they may violate many prescriptions of the language manual, such as type compatibility between the operands of an expression, agreement between the actual and the formal parameters of a procedure, and consistence between a variable declaration and its use in an instruction.

A good way to check such prescriptions is by means of semantic rules that return boolean attributes called *semantic predicates*. A given source text violates a semantic prescription if the corresponding semantic predicate turns out to be false after evaluating the attributes. Then the compiler reports a corresponding error, referred to as a *static semantic error*.

In general, the semantic predicates functionally depend on other attributes that represent various program properties. For an example, consider the agreement between a variable declaration and its use in an assignment statement. In a typical program, declaration and use are arbitrarily distant substrings; therefore the compiler must store the type of the declared variable in an attribute, called *symbol table* or *environment*. This attribute will be propagated through the syntax tree, to reach any node where the variable is used in an assignment or in another statement. However such a propagation is just a fiction, because if it were performed by copying the table, then it would be too inefficient. In practice the environment is implemented by a global data structure (or object), which is in the scope of all the concerned semantic functions.

The next attribute grammar (Example 5.50) schematizes the creation of a symbol table and its use for checking the variables that are present in the assignments.

Example 5.50 (*symbol table and type checking*) The example covers the declarations of scalar and vector variables to be used in the assignments. For the sake of the example, we assume the following semantic prescriptions have to be enforced:

1. A variable may not be doubly declared.
2. A variable may not be used before declaration.
3. The only valid assignments are between scalar variables and between vector variables of identical dimension.

The attribute grammar is shown in Table 5.6. The syntax is in a quite abstract form and distinguishes the variable declarations from their uses. Attributes n and v are the name of a variable and the value of a constant, respectively. The symbol table is searched using as key the name n of a variable. For each declared variable, the symbol table contains a descriptor *descr* with the variable type (scalar or vector) and the vector dimension, if applicable. During its construction, the table is hosted by attribute t. Predicate *dd* denounces a double declaration. Predicate *ai* denounces a type incompatibility between the left and the right parts of an assignment. Attribute t, the symbol table, is propagated to the whole tree for the necessary local controls to take place.

A summary of the attributes follows:

attribute	*meaning*	*domain*	*type*	*assoc. symbols*
n	variable name	string	left	D, id
v	constant value	number	left	const
dd	double declaration	boolean	left	D
ai	left/right parts incompatible	boolean	left	A
descr	variable descriptor	record	left	D, L, R
t	symbol table	array of records	right	A, D, L, P, R

The semantic analyzer processes a declaration D, and if the declared variable is already present in the symbol table, it sets predicate *dd* to true. Otherwise, the variable descriptor is constructed and passed to the father node, along with the variable name.

The left and right parts L and R of an assignment A have an attribute *descr* (descriptor) that specifies the type of each part: variable (indexed or not) or constant. If a name does not exist in the symbol table, the descriptor is assigned an error code.

The semantic rules control the type compatibility of the assignment and return predicate *ai*. The control that the left and right parts of an assignment are compatible is specified in pseudo-code: the error conditions listed in items 2 and 3. make the predicate true.

Table 5.6 Grammar for checking the declaration of variables versus their use in the assignment statements (Example 5.50)

Syntax	Semantic Functions	Comment
$S \rightarrow P$	$t_1 := \emptyset$	Initialize tab. to empty
$P \rightarrow D\ P$	$t_1 := t_0$ $t_2 := insert\ (\ t_0,\ n_1,\ descr_1\ )$	Propag. tab. to subtree add description to tab.
$P \rightarrow A\ P$	$t_1 := t_0$ $t_2 := t_0$	Propag. tab. to subtrees
$P \rightarrow \varepsilon$		No functions used here

$D \rightarrow$ id	$dd_0 := present\ (\ t_0,\ n_{\text{id}}\ )$
	if $\neg dd_0$ **then**
	$\quad descr_0 :=$ 'sca'
	end if
	$n_0 := n_{\text{id}}$
		Declare scalar variable

$D \rightarrow$ id [const]	$dd_0 := present\ (\ t_0,\ n_{\text{id}}\ )$
	if $\neg dd_0$ **then**
	$\quad descr_0 :=$ ('vect', v_{const})
	end if
	$n_0 := n_{\text{id}}$
		Declare vector variable

$A \rightarrow L := R$	$t_1 := t_0$ $t_2 := t_0$ $ai_0 := \neg\ \langle\ descr_1$ is compatible with $descr_2\ \rangle$	Propag. tab. to subtrees
$L \rightarrow$ id	$descr_0 := \langle$ type of n_{id} in $t_0\ \rangle$	

$L \rightarrow$ id [id]	**if** $\left(\begin{array}{l}\langle \text{type of } n_{\text{id}_1} \text{ in } t_0 \rangle = \text{'vect' and} \\ \langle \text{type of } n_{\text{id}_2} \text{ in } t_0 \rangle = \text{'sca'}\end{array}\right)$ **then**
	$\quad descr_0 :=$ 'sca'
	else
	$\quad$ error
	end if
		Use indexed variable

$R \rightarrow$ id	$descr_0 := \langle$ type of n_{id} in $t_0\ \rangle$	Use sca./vect. variable
$R \rightarrow$ const	$descr_0 :=$ 'sca'	Use constant

$R \rightarrow$ id [id]	**if** $\left(\begin{array}{l}\langle \text{type of } n_{\text{id}_1} \text{ in } t_0 \rangle = \text{'vect' and} \\ \langle \text{type of } n_{\text{id}_2} \text{ in } t_0 \rangle = \text{'sca'}\end{array}\right)$ **then**
	$\quad descr_0 :=$ 'sca'
	else
	$\quad$ error
	end if
		Use indexed variable

For instance, in the syntactically correct text below:

$$\overbrace{a\,[10]}^{D_1}\ \overbrace{i}^{D_2}\ \overbrace{b}^{D_3}\ \overbrace{i := 4}^{A_4}\ \overbrace{c := a\,[i]}^{A_5:\, ai = true}\ \overbrace{c\,[30]}^{D_6}\ \overbrace{i}^{D_7:\, dd = true}\ \overbrace{a := c}^{A_8:\, ai = true}$$

a few semantic errors have been detected in assignments A_5, A_8 and in the declaration D_7. □

Many improvements and additions would be needed for a real compiler, and we mention a few of them. First, to make diagnostic more accurate, it is preferable to separate various error classes, e.g., undefined variable, incompatible type and wrong dimension.

Second, the compiler must tell the programmer the position (e.g., the line number) of each error occurrence. By enriching the grammar with other attributes and functions, it is not difficult to improve on diagnostic and error identification. In particular any semantic predicate, when it returns true in some tree position, can be propagated toward the tree root together with a node coordinate. Then in the root another semantic function will be in charge of writing comprehensive and readable error messages.

Third, there are other semantic errors that are not covered by this example: for instance the check that each variable is initialized before its first use in an expression, or that, if it is assigned a value, then it is used in some other statement. For such controls, compilers adopt a more convenient method instead of attribute grammars, called *static program analysis*, to be described at the end of this chapter.

Finally, a program that has passed all the semantic controls in compilation may still produce *dynamic* or *run-time errors* when executed. See for instance the following program fragment:

$$\textbf{array}\ a\,[10];\quad \ldots\quad \textbf{read}\,(i);\quad a\,[i] := \ldots$$

The **read** instruction may assign to variable i a value that falls out of the interval $1 \ldots 10$, a condition clearly undetectable at compilation time.

5.6.8.2 Code Generation

Since the final product of compilation is to translate a source program to a sequence of target instructions, their selection is an essential part of the process. The problem occurs in different settings and connotations, depending on the nature of the source and target languages, and on the distance between them. If the differences between the two languages are small, the translation can be directly produced by the parser, as we have seen in Sect. 5.4.1 (p. 391) for the conversion from infix to polish notation of arithmetic expressions.

On the other hand, it is much harder to translate a high-level language, say Java, to a machine language, and the large distance between the two makes it convenient to subdivide the translation process into a cascade of simpler phases. Each phase

translates an *intermediate language* or *representation* to another one. The first stage takes Java as source language, and the last phase produces machine code as target language. Compilers have used quite a variety of intermediate representations: textual representations in polish form, trees or graphs, representations similar to assembly language, etc.

An equally important goal of decomposition is to achieve *portability* with respect to the target and source language. In the first case, portability (also called *retargeting*) means the ease of modifying an existing compiler, when it is required to generate code for a different machine. In the second case, the modification comes from a change in the source language, say, from Java to *FORTRAN*. In both cases some phases of a multi-phase compiler are independent of the source or target languages, and can be reused at no cost.

The first phase is a syntax-directed translator, guided by the syntax of, say, Java. The following phases select machine instructions and transform the program, in order to maximize the execution speed of the target program, to minimize its memory occupation or, in some cases, to reduce the electric energy consumption.[27] Notice the first phase or phases of a compiler are essentially independent of the characteristic of the target machine; they comprise the so-called *front-end* compiler.

The last phases are machine dependent and are called the *back-end* compiler. The back-end actually contains several subsystems, including at least a machine code selection module and a machine register allocation one. The same front-end compiler is usually interfaced to several back-end compilers, each one oriented toward a specific target machine.

The next examples offer a taste of the techniques involved in translating from high-level to machine-level instructions, in the very simple case of control instructions. In a programming language, control statements prescribe the order and choice of the instructions to be executed. Constructs like *if then else* and *while do* are translated by the compiler to conditional and unconditional jumps. We assume the target language offers a conditional instruction jump-if-false with two arguments: a register *rc* containing the test condition and the label of the instruction to jump to.

In the syntax, the nonterminal L stands for a list of instructions. Clearly, each jump instruction may require a fresh label that differs from the already used labels: therefore, the translator needs an unbounded supply of labels. To create such new labels when needed, the compiler invokes a function *fresh* that at each invocation assigns a new label to the attribute n.

The translation of a construct is accumulated in attribute *tr*, by concatenating (sign •) the translations of the constituents and by inserting jump instructions with newly created labels. Such labels have the form e_397, f_397, i_23, …, where the integer suffix is the number returned by function *fresh*. Register *rc* is designated by the homonymous attribute of nonterminal *cond*.

[27] Code selecting phases are often designed by using specialized algorithms based on the recognition of patterns on an intermediate tree representation. For an introduction to such methods, see, e.g., the textbooks [14–16].

Table 5.7 Grammar for translating conditional *if then else* instructions to conditional jumps with labels (Example 5.51)

Syntax	Semantic functions
$F \rightarrow I$	$n_1 := fresh$

$I \rightarrow$ **if** (*cond*)
then
L_1
else
L_2
end if
	$tr_0 := tr_{cond}$ •
	jump-if-false rc_{cond}, e_n_0 •
	tr_{L_1} • jump f_n_0 •
	e_n_0 : •
	tr_{L_2} •
	f_n_0 :

In the next examples (Examples 5.51 and 5.52), we respectively illustrate these issues with conditional and iterative instructions.

Example 5.51 (*conditional instruction*) The grammar of the conditional statement *if then else*, generated by nonterminal I, is shown in Table 5.7. For brevity we omit the translation of a boolean condition *cond* and of the other language constructs. A complete compiler should include the grammar rules for all of them.

We exhibit the translation of a program fragment, by assuming the label counter is set to $n = 7$:

if ($a > b$)	$tr\,(a > b)$
then	jump-if-false rc, e_7
$a := a - 1$	$tr\,(a := a - 1)$ jump f_7
else	e_7:
$a := b$	$tr\,(a := b)$
end if	f_7:
…	… – rest of the program

Remember that $tr\,(\ldots)$ is the machine language translation of a construct. Register rc is not chosen here, but when the compiler translates expression $a > b$ and selects a register to put the expression result into. □

Next, we show the translation of a loop with initial condition.

Table 5.8 Grammar for translating iterative *while do* instructions to conditional jumps with labels (Example 5.52).

Syntax	Semantic functions
$F \rightarrow W$	$n_1 := fresh$
$W \rightarrow$ **while** (*cond*) **do** L **end while**	$tr_0 :=$ i_n_0 : • tr_{cond} • jump-if-false rc_{cond}, f_n_0 • tr_L • jump i_n_0 • f_n_0 :

Example 5.52 (*iterative instruction*) The grammar for a *while do* statement, shown in Table 5.8, is quite similar to that of Example 5.51 and does not need further comments.

It suffices to display the translation of a program fragment (assuming that function *fresh* returns value 8):

while ($a > b$)	i_8: $tr\,(a > b)$
do	jump-if-false rc, f_8
$a := a - 1$	$tr\,(a := a - 1)$ jump i_8
end while	f_8:
…	… – rest of the program

Other conditional and iterative statements, e.g., multi-way conditionals and loops with final conditions, require similar sets of compiler rules. □

However, the straightforward translations obtained are often inefficient and need improvement, which is done by the optimizing phases of the compiler.[28] A trivial example is the condensation of a chain of unconditional jumps into a single jump instruction.

5.6.8.3 Semantics-Directed Parsing

In a standard compilation process, we know that parsing comes before semantic analysis: the latter operates on the syntax tree constructed by the former. However, in some circumstances, syntax is ambiguous and parsing cannot be successfully executed on its own, because it would produce too many trees and thus would puzzle

[28] The optimizer is by far the most complex and expensive part of a modern compiler; the reader is referred to, e.g., the textbooks [14–17].

the semantic evaluator. Actually, this danger concerns just a few technical languages, because the majority is designed so that their syntax is deterministic. But in natural language processing the change of perspective is dramatic, because the syntax of human languages by itself is very ambiguous.

We are here considering the case when the reference syntax of the source language is indeterministic or altogether ambiguous, so that a deterministic look-ahead parser cannot produce a unique parse of the given text. A synergic organization of syntax and semantic analysis overcomes this difficulty, as explained next.

Focusing on artificial rather than natural languages, a reasonable assumption is that no sentence is semantically ambiguous; i.e., every valid sentence has a unique meaning. Of course, this does not exclude a sentence from being syntactically ambiguous. But then the uncertainty between different syntax trees can be solved at parsing time, by collecting and using semantic information as soon as it is available.

For top-down parsing, we recall that the critical decision is the choice between alternative productions, when their *ELL* (1) guide sets (p. 299) overlap on the current input character. Now we propose to help the parser solve the dilemma by testing a semantic attribute termed a *guide predicate*, which supplements the insufficient syntactic information. Such a predicate has to be computed by the parser, which is enhanced with the capability to evaluate the relevant attributes.

Notice this organization resembles the multi-sweep attribute evaluation method described on p. 446. The whole set of semantic attributes is divided into two parts assigned for evaluation to cascaded phases. The first set includes the guide predicates and the attributes they depend on; this set must be evaluated in the first phase, during parsing. The remaining attributes may be evaluated in the second phase, after the, by now unique, syntax tree has been passed to the phase two evaluator.

We recall the requirements for the first set of attributes to be computable at parsing time: the attributes must satisfy the L-condition (p. 450). Consequently, the guide predicate will be available when it is needed for selecting one of the alternative productions, for expanding a nonterminal D_i in the production $D_0 \rightarrow D_1 \ldots D_i \ldots D_r$, with $1 \leq i \leq r$. Since the parser works depth-first from left to right, the part of the syntax tree from the root down to the subtrees $D_1 \ldots D_{i-1}$ is then available.

Following the L-condition, the guide predicate may only depend on the right attributes of D_0 and on other (left or right) attributes of any symbol, which in the right part of the production precedes the root of the subtree D_i under construction.

The next example illustrates the use of guide predicates.

Example 5.53 (*a language without punctuation marks*) The syntax of the historical Pascal-like language *PLZ-SYS*[29] did without commas and any punctuation marks, thus causing many syntactic ambiguities, in particular in the parameter list of a procedure. In this language a parameter is declared with a type and more parameters may be grouped together by type.

[29] Designed in the 1970's for an eight-bit microprocessor with minimal memory resources.

Procedure P contains five identifiers in its parameter list, which can be interpreted in three ways:

$$P \textbf{ proc } (X\ Y\ T1\ Z\ T2) \begin{cases} 1. & X \text{ has type } Y \text{ and } T1,\ Z \text{ have type } T2 \\ 2. & X,\ Y \text{ have type } T1 \text{ and } Z \text{ has type } T2 \\ 3. & X,\ Y,\ T1,\ Z \text{ have type } T2 \end{cases}$$

Insightfully, the language designers prescribed that type declarations must come before procedure declarations. For instance, if the type declarations occurring before the declaration of procedure P are the following:

$$\textbf{type } T1 = \textbf{record } \ldots \textbf{ end} \qquad \textbf{type } T2 = \textbf{record } \ldots \textbf{ end}$$

then case 1 is excluded, because Y is not a type and $T1$ is not a variable. Similarly case 3 is excluded, and therefore the ambiguity is solved.

It remains to be seen how the knowledge of the preceding type declarations can be incorporated into the parser, to direct the choice between the several possible cases. Within the declarative section D of language *PLZ-SYS*, we need to consider just two parts of the syntax: the type declarations T and the procedure heading I (we do not need to concern us here with procedure bodies). Semantic rules for type declarations will insert type descriptors into a symbol table t, managed as a left attribute. As in earlier examples, attribute n is the name or key of an identifier.

Upon terminating the analysis of type declarations, the symbol table is dispatched toward the subsequent parts of the program and, in particular, to the procedure heading declarations. For downward and rightward propagation, the left attribute t is (virtually) copied into a right attribute *td*. Then the descriptor *descr* of each identifier allows the parser to choose the correct production rule.

To keep the example small, we make drastic simplifications: the scope (or visibility) of the declared entities is global to the entire program; every type is declared as a record not further specified; we omit the control of double declarations; we do not insert in the symbol table the descriptors for the declared procedures and their arguments.

The grammar fragment is listed in Table 5.9. Since type and variable identifiers are syntactically undistinguishable, as both the *type_id* and *var_id* nonterminals expand to a generic identifier id, both the alternative rules of nonterminal V start with the string id id, because:

$$V \rightarrow var_id\ V \qquad \text{- - starts with } var_id \text{ followed by } var_id$$
$$V \rightarrow var_id \qquad \text{- - starts with } var_id \text{ followed by } type_id$$

Therefore these alternative rules violate the $LL(2)$ condition. Next the parser is enhanced with a semantic test, and consequently it is allowed to choose the correct alternative.

Table 5.9 Grammar for using type declaration for disambiguation (Example 5.53)

Syntax	Semantic functions
– declarative part $D \rightarrow T\ I$	– symbol table is copied from T to I $td_I := t_T$
– type declaration $T \rightarrow$ **type** id $=$ **record** ... **end** T	– descriptor is inserted in table $t_0 := insert\,(t_2,\ n_{\text{id}},\ \text{`type'})$
$T \rightarrow \varepsilon$	$t_0 := \emptyset$
– procedure heading $I \rightarrow$ id **proc** (L) I	– table is passed to L and to $I \equiv 3$ $td_L := td_0 \qquad td_3 := td_0$
$I \rightarrow \varepsilon$	
– parameter list $L \rightarrow V\ type_id\ L$	– table is passed to V and to $L \equiv 3$ $td_V := td_0 \qquad td_3 := td_0$
$L \rightarrow \varepsilon$	
– variable list (of same type) $V \rightarrow var_id\ V$	– table is passed to var_id and to $V \equiv 2$ $td_1 := td_0 \qquad td_2 := td_0$
$V \rightarrow var_id$	$td_1 := td_0$
$type_id \rightarrow$ id	
$var_id \rightarrow$ id	

Let $cc1$ and $cc2$, respectively, be the current terminal character (or rather lexeme) and the next one. The guide predicates for each alternative rule are listed below:

#	*production*	*guide predicate*	
1 1'	$V \rightarrow var_id\ V$	⟨ the descr. of $cc2$ in table td_0 ⟩ $\neq$ 'type' ⟨ the descr. of $cc1$ in table td_0 ⟩ $\neq$ 'type'	**and**
2 2'	$V \rightarrow var_id$	⟨ the descr. of $cc2$ in table td_0⟩ $=$ 'type' ⟨ the descr. of $cc1$ in table td_0 ⟩ $\neq$ 'type'	**and**

The mutually exclusive clauses 1 and 2 act as guide predicates, to select one alternative out of 1 and 2. Clauses 1' and 2' are a semantic predicate controlling that the identifier associated with *var_id* is not a type identifier.

Furthermore, it would be possible to add a semantic predicate to production $L \rightarrow V\ type_id\ L$, in order to check whether the sort of $type_id \equiv cc2$ in the table is equal to "type". □

With such help from the values of the available semantic attributes, the top-down parser deterministically constructs the tree.

5.7 Static Program Analysis

In this last part of the book we describe a technique for program analysis and optimization, used by all the compilers translating a programming language and also by many software engineering tools.

Imagine that the front-end compiler has translated a program to an intermediate representation closer to the machine or the assembly language. The intermediate program is then analyzed by other compiler phases, the purpose and functionality of which differ depending on these circumstances:

verification	to further examine the correctness of the program
optimization	to transform the program into a more efficient version, for instance by optimally assigning machine registers to program variables
scheduling and parallelizing	to change the instruction order, for a better exploitation of processor pipelines and multiple functional units, and for avoiding that such resources be at times idle and at times overcommitted

Although very different, such cases use a common representation of the program, called a *control-flow graph*, similar to a program flowchart. It is convenient to view this graph as describing the state-transition function of a finite automaton. Here our standpoint is entirely different from syntax-directed translation, because the automaton is not used to formally specify a programming language, but just for the given program on which attention is focused. A string recognized by the control-flow automaton denotes an execution trace of that program, i.e., a sequence of machine operations.

Static analysis consists of the study of certain properties of control-flow graphs, by using various methods that come from logic, automata theory and statistic. In our concise presentation we mainly consider the logical approach.

5.7.1 A Program as an Automaton

In a program control-flow graph, each node is an instruction. At this level the instructions are usually simpler than in a high-level programming language, since they are a convenient intermediate representation produced by the front-end compiler. Further simplifying matters, we assume the instruction operands are simple variables and constants; that is, there are not any aggregate data types. Typical instructions are assignments to variables, and elementary arithmetic, relational and boolean expressions, usually with at most one operator.

In this book we only consider *intraprocedural* analysis, meaning that the control-flow graph describes one subprogram at a time. More advanced studies[30] are *interprocedural*: they analyze the properties of a full program involving multiple procedures and their invocations.

If the execution of an instruction p can be immediately followed by the execution of an instruction q, the graph has an arc directed from p to q. Thus an arc represents the immediate precedence relation between instructions: p is the *predecessor*, and q is the *successor*.

The first instruction a program executes is the *entry* point, represented by the *initial node* of the graph. For convenience, we assume the initial instruction does not have any predecessors. On the other hand, an instruction having no successors is a program *exit* point or *final node* of the graph.

Unconditional instructions have at most one successor. Conditional instructions have two successors (and more than two for instructions such as the *switch* statement of the *C* language). An instruction with two or more predecessors is a *confluence* of as many arcs of the graph.

A control-flow graph is not a faithful program representation, but just an abstraction: it suffices for extracting the properties of interest, but some information is missing, as explained next.

- The true/false value determining the successor of a conditional instruction such as *if then else* is typically not represented.
- An unconditional *go to* instruction is not represented as a node, but simply as the arc to the successor instruction.
- An operation (arithmetic, read, write, etc.) performed by an instruction is replaced by the following abstraction:

  – A value assignment to a variable, by means of a statement such as an assignment or a read instruction, is said to *define* that variable.
  – If a variable occurs in an expression, namely in the right part of an assignment statement, or in a boolean expression of a conditional, or in the argument list

[30] The already cited books on compiler techniques cover also static analysis, and a more specific reference is [18].

of a write instruction, we say the statement *uses* (or makes reference to) that variable.
– Therefore, a node representing a statement p in the graph is associated with two sets: the set $def(p)$ of defined variables and the set $use(p)$ of used variables.

Notice that in this model the actual operations performed by a statement, say, multiplication versus addition, are typically overlooked.

Consider for instance a statement $p: a := a \oplus b$, where $\oplus$ is an unspecified binary operator. The instruction is represented in the control-flow graph by a node carrying the following information:

$$def(p) = \{ a \} \qquad use(p) = \{ a, b \}$$

In this abstract model the statements **read** (a) and $a := 7$ are undistinguishable, as they carry the same associate information: $def = \{ a \}$ and $use = \emptyset$.

In order to clarify the concepts and to describe some applications of the method, we present a more complete example (Example 5.54).

Example 5.54 (*flowchart and control-flow graph*) In Fig. 5.20 we see a subprogram with its flowchart and its control-flow graph. In the abstract control-flow graph we need not list the actual instructions, but just the sets of defined and used variables. Instruction 1 has no predecessors and is the subprogram entry or initial node. Instruction 6 has no successors and is the program exit or final node. Node 5 has two successors, whereas node 2 is at the confluence of two predecessors. The sets $use(1)$ and $def(5)$ are empty. □

Language of Control-Flow Graph

We consider the finite automaton A, represented by a control-flow graph. Its terminal alphabet is the set I of program instructions, each one schematized by a triple ⟨ *label*, *defined* variables, *used* variables ⟩ such as:

$$\langle 2,\ def(2) = \{ b \},\ use(2) = \{ a \} \rangle$$

For the sake of brevity, we often denote such instruction by the first component only, i.e., the instruction *label* or number. Notice that two nodes containing the same instruction, say $x := y + 5$, are made distinct by their labels.

We observe that the arcs are unlabeled and all the information is attached to the nodes. This should remind us of the local automata studied on p. 152, where the terminal characters of the alphabet are written inside the states. Clearly, all the arcs entering the same node "read" the same character. Since each state is identified by the instruction label, we do not have to assign it an explicit name; the same as in the syntax charts on p. 237.

The initial state is marked by an entering arrow (dart). The final states are those without a successor.

```
program
        a := 1
e_1:    b := a + 2
        c := b + c
        a := b × 3
        if a < m goto e_1
        return c
```

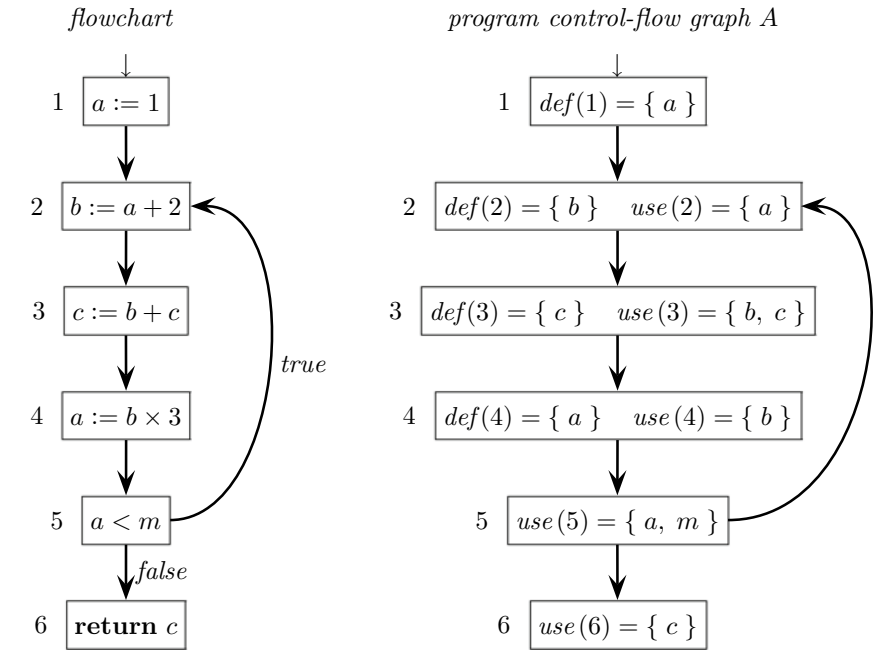


Fig. 5.20 Program, flowchart and abstract control-flow graph of Example 5.54

The formal language $L(A)$ recognized by the automaton contains the strings over alphabet I that label a path from the entry node to an exit node. Such path denotes a sequence of instructions, which may be executed when the program is run.

Obviously, each node number is distinct since it corresponds to a different instruction label. This confirms that the formal language $L(A)$ belongs to the family of local languages (Definition 3.28 on p. 151), which are a rather restricted subset of the regular language family *REG*.

In the previous example, the alphabet is $I = \{ 1, 2, 3, 4, 5, 6 \}$. A recognized path is the following:

$$1 \to 2 \to 3 \to 4 \to 5 \to 2 \to 3 \to 4 \to 5 \to 6 \equiv 1\,2\,3\,4\,5\,2\,3\,4\,5\,6$$

The set of such recognized paths is the language $L(A) = 1\ (2\,3\,4\,5)^{+}\ 6$.

Conservative Approximation

Actually, the automaton specifies only an approximation of the valid execution paths of a program. Not all the recognized paths are really executable by the program, because our model disregards the boolean condition that selects a successor node of a conditional statement. A trivial example is the following program fragment:

$$1\colon \textbf{if}\ a * * 2 \geq 0\ \textbf{then}\ 2\colon istr_2\ \textbf{else}\ 3\colon istr_3$$

where the formal language accepted by the automaton contains two paths $\{\ 1\,2,\ \ 1\,3\ \}$, but path 1 3 is not executable because a square is never negative.

As a consequence of such an approximation, static analysis may sometimes reach pessimistic conclusions: in particular, it may discover errors in a never-executed path.

Of course, it is in general undecidable whether a path of a control-flow graph will ever be executed, because this would be equivalent to deciding whether there exists a value assignment to the input variables that causes the execution of that path. The latter problem can be reduced to the halting problem of a Turing machine, which is undecidable.

As it is generally impossible to know which program paths are executable or not, it would be much worse if the static analysis erred by disregarding some path that turns out to be executable, because then it may fail to detect some real errors.

In conclusion, the decision to examine all the recognized paths (from the initial node to a final one) is a *conservative approximation* to program analysis, which may cause the diagnosis of non-existing errors or the prudential assignment of unnecessary resources, but it never misses real error conditions or real resource requirements.

A usual hypothesis in static analysis is that the automaton is clean (p. 124); i.e., each instruction is on a path from the initial node to a final one. Otherwise one or more of the following anomalies may occur in the program: some executions never terminate, or some instructions are never executed (the program contains so-called *unreachable code*).

5.7.2 Liveness Intervals of a Variable

A professional compiler performs several passes of analysis over the intermediate representations of a program in order to improve it. A very interesting analysis, which allows a variety of profitable optimizations, is the study of the liveness intervals of program variables.

Definition 5.55 (*variable liveness*) A variable a is *live* on the exit from a program node p, if in the program control-flow graph there exists a path from p to a node q. not necessarily distinct from p, such that:

- The path does not traverse an instruction r, with $r \neq q$, that defines variable a, i.e., such that $a \in def(r)$.

- Instruction q uses variable a, i.e., $a \in use\,(q)$. □

For brevity we say that the variable is *live-out* of node p. In other words, a variable is live-out of a certain node if some instruction that may be successively executed makes use of the value the variable has in the former node.

To grasp the purpose of this definition, imagine that instruction p is the assignment $a := b \oplus c$, and suppose we want to know if some instruction makes use of the value assigned to the variable a in p. The question can be rephrased as: is variable a live-out of node p? If not, the assignment p is *useless* and can be deleted without affecting the program semantics. Furthermore, if none of the variables used by p is used in some other instruction, all the instructions assigning a value to such variables may become useless after deleting instruction p. The next example (Example 5.56) illustrates the situation.

Example 5.56 (*Example* 5.54 *continued*) For the example of Fig. 5.20 on p. 470, we reproduce in Fig. 5.21 the program control-flow graph with the live variable sets for each arc, also referred to as a *program point*. Observe in the picture the variables that are live-in each program point. Thus variable c is live on the entrance to node 1, because there exists path 1 2 3 such that $c \in use\,(3)$, and neither node 1 nor node 2 defines c.

It is customary to say that variable a is live-in the *intervals* (i.e., paths) 1 2 and 4 5 2; it is not live-in the intervals 2 3 4 and 5 6, and so on.

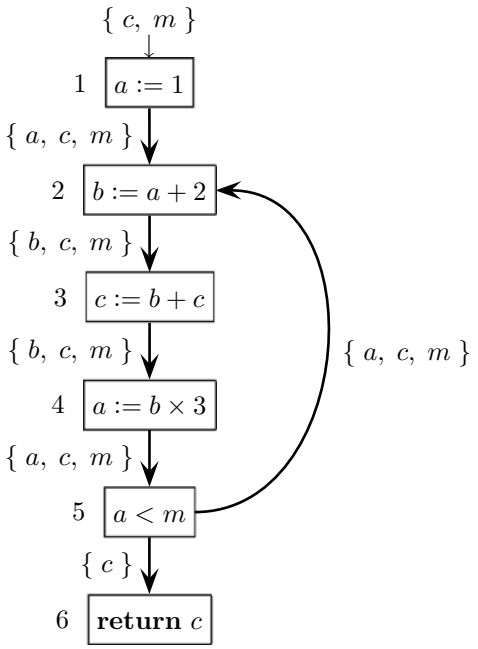


Fig. 5.21 Control-flow graph of the program with the sets of variables live-out of the nodes (Example 5.54)

More precisely, we say that a variable is *live-out for a node* if it is live on any arc outgoing from the node. Similarly, a variable is *live-in for a node* if it is live on some arc entering the node. For instance, the variables $\{ a, c, m \} \cup \{ c \}$ are live-out for node 5. □

5.7.2.1 Computing Liveness Intervals

Let I be the instruction set. Let $D(a) \subseteq I$ and $U(a) \subseteq I$, respectively, be the sets of the instructions that define and use some variable a. For instance, in the running example (Fig. 5.20 on p. 470) it is $D(b) = \{ 2 \}$ and $U(b) = \{ 3, 4 \}$. The liveness condition will be first expressed in terms of formal language operations and then by means of a more expressive set-theoretical notation.

It is not difficult to see that variable a is live-out for node p if, and only if, for the language $L(A)$ accepted by the automaton, the following condition holds: language $L(A)$ contains a sentence $x = u\, p\, v\, q\, w$, where u and w are (possibly empty) arbitrary instruction sequences, p is any instruction, v is a possibly empty instruction sequence not containing a definition of a and instruction q uses variable a. The above conditions are formalized as follows:

$$u,\ w \in I^* \ \wedge\ p \in I \ \wedge\ v \in \big(I \setminus D(a) \big)^* \ \wedge\ q \in U(a) \tag{5.4}$$

Observe again that the set difference contains all the instructions that do not define variable a, whereas instruction q uses a.

The set of all the strings x that meet condition (5.4), denoted as L_p, is a subset of the language $L(A)$ recognized by A, i.e., $L_p \subseteq L(A)$. Moreover language L_p is regular, because it can be defined by the following intersection:

$$L_p = L(A) \ \cap\ R_p \tag{5.5}$$

where language R_p is regular and is defined by the extended (i.e., with a set difference) regular expression R_p below:

$$R_p = I^*\ p\ \big(I \setminus D(a) \big)^*\ U(a)\ I^* \tag{5.6}$$

Formulas (5.5) and (5.6) prescribe that letter p must be followed by a letter q taken from set $U(a)$, and that all the letters (if any) intervening between p and q must not belong to set $D(a)$.

It follows that in order to decide whether variable a is live-out of node p, one has to check that language L_p is not empty. We know one way of doing it: we construct the recognizer of language L_p, that is, the product machine for the intersection (5.5), as explained in Chap. 3 on p. 190. If this machine does not contain any path from the initial state to a final one, then language L_p is empty.

Anyway, such a procedure is not practical, when taking into account the large dimension of the real programs to be analyzed. Therefore we introduce another specialized method, which not only performs more efficiently, but computes at once all

the live variables in all the program points. The new method systematically examines all the paths from the current program point to some instruction that uses some variable.

The liveness computation will be expressed as a system of *data-flow equations*. Consider a node p of a control-flow graph or program A. A first equation expresses the relation between the variables live-out $live_{out}(p)$ and those live-in $live_{in}(p)$. A second equation expresses the relation between the variables live-out of a node and those live-in for its successors.

We denote by $succ(p)$ the set of the (immediate) successors of node p and by $var(A)$ the set of all the variables of program A.

Data-Flow Equations

For each final (exit) node p:

$$live_{out}(p) = \emptyset \tag{5.7}$$

For any other node p:

$$live_{in}(p) = use(p) \cup \big(live_{out}(p) \setminus def(p)\big) \tag{5.8}$$

$$live_{out}(p) = \bigcup_{\forall q \in succ(p)} live_{in}(q) \tag{5.9}$$

Comments:

- In Eq. (5.7) no variable is live-out of any node of the (sub)program graph. Here we have disregarded the output parameters (if any) of the subprogram, which are typically used after exiting the subprogram and thus can be considered to be live-out of the final node, though in another subprogram.
- For Eq. (5.8) a variable is live-in for p if it is used in p or if it is live-out for p but not defined by p. Consider instruction 4: $a := b \times 3$ (Fig. 5.20 on p. 470). Out of 4, variables a, m and c are live, because for each one there exists a path that reaches a use of that variable without traversing a node that defines the same variable. On entering node 4, the following variables are live: b because it is used in 4; and c and m because they are live-out for 4 and not defined in 4. On the contrary, variable a is not live-in for 4 because it is defined in 4, though it is live-out for 4.
- For Eq. (5.9), node 5 has successors 2 and 6; then the variables live-out for 5 are those (namely a, c and m) live-in for 2 and the one (namely c) live-in for 6.

5.7.2.2 Solution of Data-Flow Equations

Given a control-flow graph, it is straightforward to write the two Eqs. (5.8) and (5.9) for each instruction. For a graph with $|I| = n \geq 1$ nodes, the resulting system has $2 \times n$ equations with $2 \times n$ unknowns, i.e., $live_{in}(p)$ and $live_{out}(p)$ for each instruction $p \in I$. Each unknown is a set of variables, and the solution to be computed is a pair of vectors, each one containing n sets.

To solve the equation system we use iteration, by taking the empty set as the initial approximation (with $i = 0$) for every unknown:

$$\forall\ p \in I \qquad live_{in}(p) = \emptyset \qquad live_{out}(p) = \emptyset$$

Let $i \geq 0$ be the current iteration. In each equation of the system (5.8), (5.9), we replace the unknowns occurring in the right-hand sides with the values of the current iteration, and thus we obtain the values of next iteration $i + 1$. If at least one unknown differs from the previous iteration, then we execute one more iteration; otherwise we terminate and the last vector pair computed is a solution of the equation system.

This solution is termed the *least fixed point* of the transformation that computes a new vector from one of the preceding iterations.

To see why a finite number of iterations always suffices to converge to the least fixed-point solution, observe the following:

- The cardinality of every set $live_{in}(p)$ and $live_{out}(p)$ is bounded by the number of program variables $|var(A)|$.
- Every iteration may only add some variables to some sets or leave them unchanged, but it never removes any variable from a set; in other words, the transformation is monotonic nondecreasing with respect to set inclusion.
- If an iteration does not change any set, the algorithm terminates.

We illustrate the algorithm on the running example.

Example 5.57 (*running example (Example 5.54) continued: iterative computation of live variables*) First, we compute by inspection the sets of instructions that define (D) and use (U) program variables:

var.	D	U
a	1, 4	2, 5
b	2	3, 4
c	3	3, 6
m	$\emptyset$	5

Next, the equations for the program (Fig. 5.20 on p. 470) are written in Table 5.10. The names of the unknowns are shortened to $in(p)$ and $out(p)$ instead of $live_{in}(p)$ and $live_{out}(p)$, respectively. Then we compute and tabulate the successive approximations, starting from the empty sets; at each iteration we first compute the *in* values and then the *out* values. The least fixed point is reached after five iterations: it would be easy to verify that one more iteration would not change the last result. □

It is important to note that the convergence speed to the fixed point is very sensitive to the order, although the solution does not depend on the processing order of nodes.

Table 5.10 Liveness equations of the program in Fig. 5.20 (Example 5.57)

Equations		
1	$in\,(1) = out\,(1) \setminus \{\,a\,\}$	$out\,(1) = in\,(2)$
2	$in\,(2) = \{\,a\,\} \cup (\,out\,(2) \setminus \{\,b\,\})$	$out\,(2) = in\,(3)$
3	$in\,(3) = \{\,b,\ c\,\} \cup (\,out\,(3) \setminus \{\,c\,\})$	$out\,(3) = in(\,4)$
4	$in\,(4) = \{\,b\,\} \cup (\,out\,(4) \setminus \{\,a\,\})$	$out\,(4) = in\,(5)$
5	$in\,(5) = \{\,a,\ m\,\} \cup out\,(5)$	$out\,(5) = in\,(2) \cup in\,(6)$
6	$in\,(6) = \{\,c\,\}$	$out\,(6) = \emptyset$

Unknowns computed at each iteration

	$in = out$	in	out	in	out	in	out	in	out	in	out
1	$\emptyset$	$\emptyset$	a	$\emptyset$	a, c	c	a, c	c	a, c, m	c, m	a, c, m
2	$\emptyset$	a	b, c	a, c	b, c	a, c	b, c, m	a, c, m	b, c, m	a, c, m	b, c, m
3	$\emptyset$	b, c	b	b, c	b, m	b, c, m	b, c, m	b, c, m	b, c, m	b, c, m	b, c, m
4	$\emptyset$	b	a, m	b, m	a, c, m	b, c, m	a, c, m	b, c, m	a, c, m	b, c, m	a, c, m
5	$\emptyset$	a, m	a, c	a, c, m	a, c	a, c, m	a, c	a, c, m	a, c, m	a, c, m	a, c, m
6	$\emptyset$	c	$\emptyset$	c	$\emptyset$	c	$\emptyset$	c	$\emptyset$	c	$\emptyset$

We mention the time complexity of the iterative algorithm.[31] The worst-case complexity is $\mathcal{O}\,(n^4)$, where n is the number of nodes, i.e., of instructions of the subprogram. In practice, for many realistic programs the computational complexity is close to linear in time.

5.7.2.3 Application of Liveness Analysis

We show two classical widespread applications of the previous analysis: memory allocation for variables and detection of useless instructions.

Memory Allocation

Liveness analysis is best applied to decide if two variables can reside in the same memory cell (or in the same machine register). It is evident that if two variables are live-in the same program point, both values must be present in the memory when execution reaches that point, because they may have future uses. Therefore their values cannot reside in the same cell: we then say the two variables *interfere*.

Conversely, if two variables do not interfere, that is, they are never live-in the same program point, then the same memory cell or register can be used to keep their values.

[31] For a proof refer for instance to any of [14, 15, 17].

Example 5.58 (*interference and register assignment*) In the control-flow graph of Fig. 5.21 on p. 472, we see that variables a, c and m occur in the same set $live_{in}$ (2); therefore the three variables pairwise interfere. Similarly, the variable pairs (b, c), (b, m) and (c, m) interfere in the set $live_{in}$ (3). On the other hand, no set contains variables a and b, which therefore do not interfere.

As stated before, two interfering variables must reside in different memory cells. It follows that each one of the variables c and m needs a separate cell, while both variables a and b may reside in the same cell, which has to be different from the previous two cells because variable a interferes with c and m. In conclusion we have found that three cells suffice to store the values of four program variables. □

Current compilers optimally assign registers to program variables by means of heuristic methods, by relying on the interference relation.

Useless Definition

An instruction defining a variable is *useless* if the value assigned to the variable is never used by any instruction. This is tantamount to saying that the value is not live-out for the defining instruction. Therefore, to verify that a definition of variable a by an instruction p is not useless, we have to check whether variable a is present in the set $live_{out}(p)$.

The program of Fig. 5.20 on p. 470 does not have any useless definitions, in contrast with the next example.

Example 5.59 (*useless variable definition*) Consider the program in Fig. 5.22. The picture lists the live variables in and out of each instruction. Variable c is not live-out for node 3; hence instruction 3 is useless. Useless instructions can be erased by the compiler. The elimination of instruction 3 brings two benefits: the program is shorter and faster to execute, and variable c disappears from the sets *in* (1), *in* (2), *in* (3) and *out* (5). This reduces the interferences between variables and may bring a reduction in the number of registers needed, which is often a bottleneck for program performance. □

This is an example of the frequently occurring phenomenon of chain reaction optimizations triggered by a simple program transformation.

5.7.3 Reaching Definition

Another basic and widely applied type of static analysis is the search for variable definitions that reach some program point.

To introduce the idea by an application, consider an instruction that assigns a constant value to variable a. The compiler examines the program to see if the same constant can be replaced for the variable in the instructions using a. The benefit of the replacement is manyfold. First, a machine instruction having a constant as operand (a so-called immediate operand) is often faster. Second, substituting with a

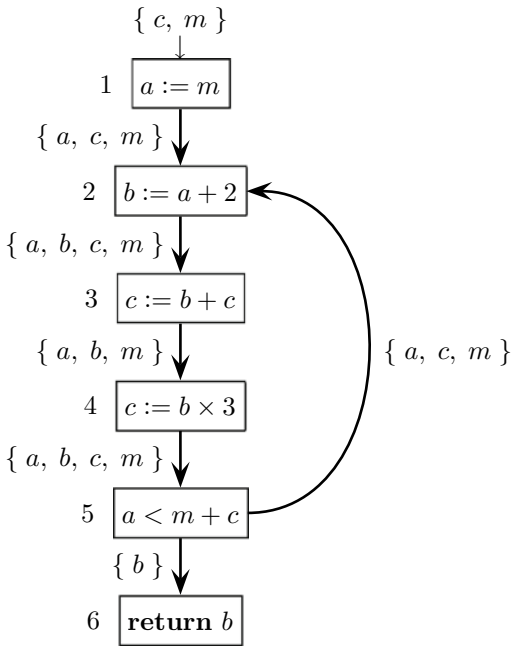


Fig. 5.22 Control-flow graph with live sets applied to detect the useless definitions in the program (Example 5.59)

constant a variable occurring in an expression may produce an expression where all the operands are constant. Then, the expression value can be computed at compile time, with no need to generate any machine code for it. Lastly, since the replacement eliminates one or more uses of a, it shortens the liveness intervals and reduces the interferences between variables; thus the pressure on the processor registers is consequently reduced, too.

The above transformation is termed *constant propagation*. In order to develop it, we need a few conceptual definitions, which are also useful for other program optimizations and verifications.

Consider an instruction $p\colon a := b \oplus c$ that defines variable a. For brevity, we denote such a variable definition as a_p, while $D(a)$ denotes the set of all the definitions of the same variable a in the subprogram under analysis. The following definition formalizes the concept.

Definition 5.60 (*reaching definition*) We say that the *defn* of a variable a in an instruction q, i.e., a_q, *reaches the entrance* of an instruction p, if there exists a path from q to p such that it does not traverse a node (distinct from q) that defines variable a. □

When this happens, instruction p may use the value of variable a computed in the instruction q.

Referring to automaton A, i.e., to the control-flow graph of the subprogram, the condition above (Definition 5.60) can be restated more precisely as follows. Defi-

nition a_q reaches instruction p if language $L(A)$ contains a sentence of the form $x = u\ q\ v\ p\ w$, where u and w are (possibly empty) arbitrary instruction sequences, p is any instruction, v is a possibly empty instruction sequence not containing any definition of a and instruction q defines variable a. The above conditions are formalized in this way:

$$u,\ w \in I^* \ \wedge\ q \in D(a)\ \wedge\ v \in \big(I \setminus D(a)\big)^* \ \wedge\ p \in I \tag{5.10}$$

Notice that the instructions p and q may coincide.

Looking again at the program on p. 470, reproduced in Fig. 5.23, we find that definition a_1 reaches the entrance of instructions 2, 3 and 4, but not the entrance of instruction 5. Definition a_4 reaches the entrance of instructions 5, 6, 2, 3 and 4.

Data-Flow Equations for Reaching Definition

To compute the reaching definitions in all the program points, we set up a system of equations similar to those for liveness analysis.

If node p defines variable a, we say that any other definition a_q of the same variable in another node q, with $q \neq p$, is *suppressed* by p. Formally, the set of definitions suppressed by instruction p is the following:

$$\begin{cases} sup(p) = \left\{ a_q \;\middle|\; \begin{array}{l} q \in I\ \wedge\ q \neq p\ \wedge \\ a \in def(q)\ \wedge\ a \in def(p) \end{array} \right\} & \text{if } def(p) \neq \emptyset \\ sup(p) = \emptyset & \text{if } def(p) = \emptyset \end{cases}$$

Notice the set $def(p)$ may contain more than one name if instruction p defines multiple variables, such as the statement **read** $(a,\ b,\ c)$.

The sets of definitions reaching the entrance to and exit from a node p are denoted as $in(p)$ and $out(p)$, respectively. The set of the (immediate) predecessor nodes of p is denoted as $pred(p)$.

Data-Flow Equations

For the initial node 1:

$$in(1) = \emptyset \tag{5.11}$$

For any other node $p \in I$:

$$out(p) = def(p)\ \cup\ \big(in(p) \setminus sup(p)\big) \tag{5.12}$$

$$in(p) = \bigcup_{\forall\, q \in pred(p)} out(q) \tag{5.13}$$

Comments:

- Equation (5.11) assumes for simplicity that no variables are passed as input parameters to the subprogram. Otherwise, more accurately, set $in(1)$ should contain all the definitions, external to the subprogram, of the input parameters.

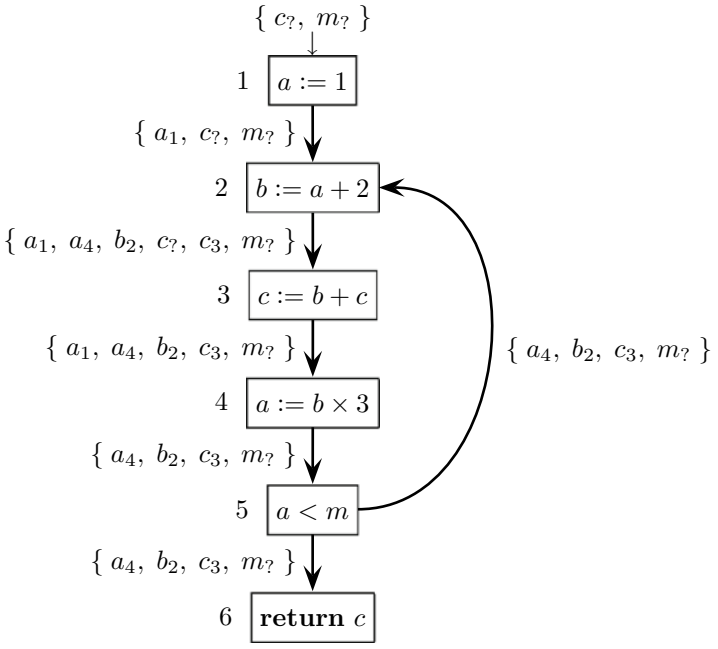


Fig. 5.23 Control-flow graph with reaching definitions (Example 5.61)

- Equation (5.12) inserts into the exit from p all the local definitions of p and the definitions reaching the entrance to p, provided the latter are not suppressed by p.
- Equation (5.13) states that any definition reaching the exit of some predecessor node reaches also the entrance to p.

Similar to the liveness equations, the reaching definition system can be solved by iteration until the computed solution converges to the first fixed point. In the starting iteration all the unknown sets are empty.

We illustrate the situation in the next example (Example 5.61) with a program containing a loop.

Example 5.61 (*reaching definition*) Observe in Fig. 5.23 the same program control-flow graph on p. 470, with the reaching definition sets computed by solving the equation system listed in Table 5.11.

Variables c and m are the input parameters of the subprogram, and we may assume they are externally defined in the calling subprogram at some unknown points denoted $c_?$ and $m_?$. For instance, notice that the external definition $c_?$ of variable c does not reach the entrance of instruction 4, since it is suppressed by instruction 3.

Table 5.11 Data-flow equations for the reaching definitions in the program (Example 5.61 in Fig. 5.23).

$$
\begin{aligned}
in\,(1) &= \{\, c_?,\ m_? \,\} \\
out\,(1) &= \{\, a_1 \,\} \cup (\, in\,(1) \setminus \{\, a_4 \,\}\,) \\
in\,(2) &= out\,(1) \cup out\,(5) \\
out\,(2) &= \{\, b_2 \,\} \cup (\, in\,(2) \setminus \emptyset\,) = \{\, b_2 \,\} \cup in\,(2) \\
in\,(3) &= out\,(2) \\
out\,(3) &= \{\, c_3 \,\} \cup (\, in\,(3) \setminus \{\, c_? \,\}\,) \\
in\,(4) &= out\,(3) \\
out\,(4) &= \{\, a_4 \,\} \cup (\, in\,(4) \setminus \{\, a_1 \,\}\,) \\
in\,(5) &= out\,(4) \\
out\,(5) &= \emptyset \cup (\, in\,(5) \setminus \emptyset\,) = in\,(5) \\
in\,(6) &= out\,(5) \\
out\,(6) &= \emptyset \cup (\, in\,(6) \setminus \emptyset\,) = in\,(6)
\end{aligned}
$$

We list the constant terms occurring in the equations:

node	*instruction*	*def*	*sup*
1	$a := 1$	a_1	a_4
2	$b := a + 2$	b_2	$\emptyset$
2	$c := b + c$	c_3	$c_?$
4	$a := b \times 3$	a_4	a_1
5	$a < m$	$\emptyset$	$\emptyset$
6	return c	$\emptyset$	$\emptyset$

At iteration 0 all the sets are empty. After a few iterations, the unknown values converge to the sets shown in Table 5.11. □

5.7.3.1 Constant Propagation

Carrying further the previous example (Example 5.61 and Fig. 5.23), we look for opportunities to replace a variable by a constant value. An instance of the problem is the question: can we replace the variable a in the instruction 2 with the constant 1 assigned by instruction 1 (i.e., definition a_1)? The answer is negative because the set $in\,(2)$ of reaching definitions contains another definition of a, namely a_4, which implies that some computation may use for a the value defined in the instruction 4. Therefore the program containing instruction $b := 1 + 2$ instead of $b := a + 2$

would not be equivalent to the original one, which is quite evident since a run can execute the loop body.

Generalizing this reasoning, it is easy to state a condition: in the instruction p, it is safe to replace with a constant k a variable a used in p if the following conditions hold:

1. There exists an instruction $q : a := k$ that assigns constant k to variable a, such that definition a_q reaches the entrance of p.
2. No other definition a_r of variable a reaches the entrance of p, with $r \neq q$.

In the next example (Example 5.62), we show some program improvements produced by a constant propagation and by the induced simplifications.

Example 5.62 (*optimization following constant propagation*) Fig. 5.24 shows a simple program control-flow graph and lists the reaching definition sets and the live

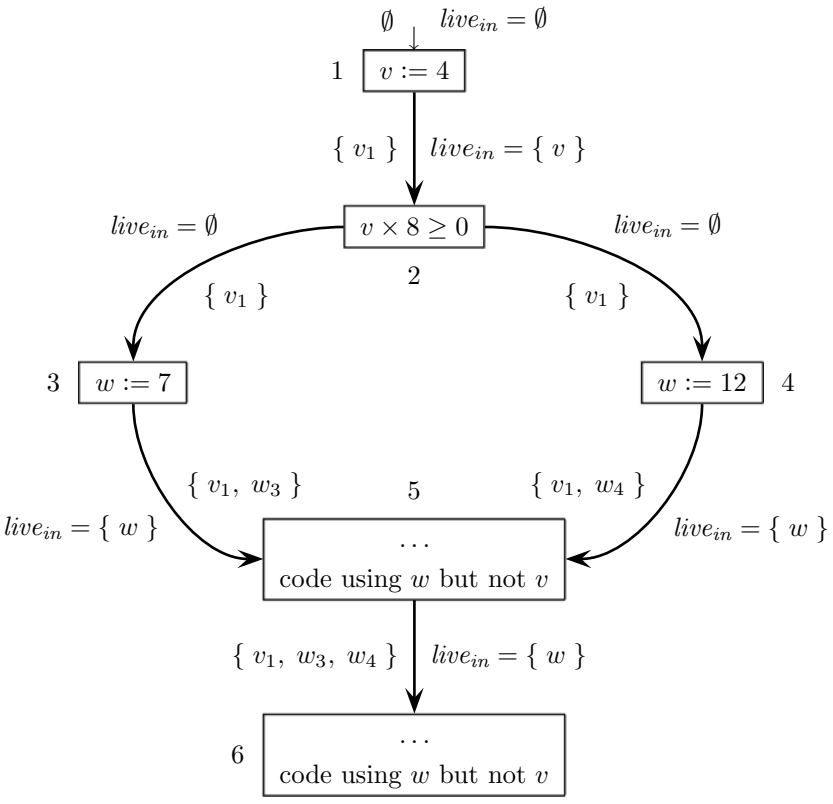


Fig. 5.24 A control-flow graph with reaching definitions and live variables before constant propagation (Example 5.62)

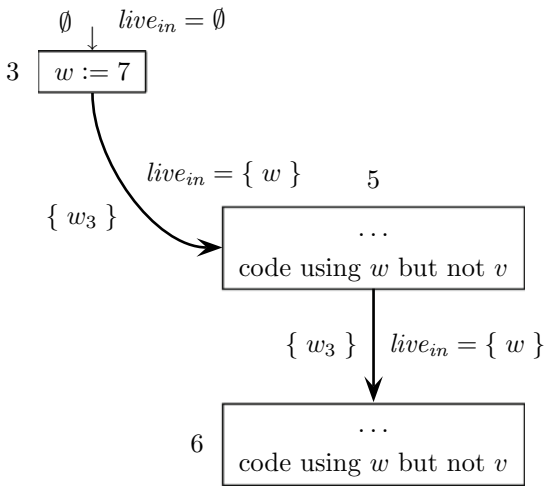


Fig. 5.25 Program control-flow graph of Fig. 5.24 after the optimizations induced by constant propagation

variable sets in the relevant points of the program. Observe that the only definition of variable v reaching the entrance of node 2 is v_1. By the previous condition, it is legal to replace variable v with constant 4 in the conditional instruction 2, which afterward becomes the constant boolean expression $4 \times 8 \geq 0$. Now, variable v ceases to be live-out for assignment 1, which becomes useless and can be deleted. But program simplification does not end here. The compiler can compute the value of the constant expression[32] $8 \times 4 = 32 \geq 0 = \mathit{true}$, thus determining which one of the successor legs of statement 2 will be taken, say the one to the left. Then the right leg will never be taken and can be deleted. We say that instruction 4 becomes *unreachable* or *dead code*, since no computation starting in the program entry will ever reach it. Now, the conditional instruction 2 is redundant and can be eliminated. After these transformations, the simplified program is shown in Fig. 5.25. Now the analysis could proceed to determine if constant $w = 7$ can be safely propagated to the rest of the program. □

5.7.3.2 Availability of Variables and Initializations

A basic correctness check a compiler should perform is to control that the variables are initialized before their first use. More generally, a variable used in some instruction must, on entrance to the instruction, have a value computed by means of a valid assignment (or by another statement type that can define variables). Otherwise, we say that the variable is not *available*, and a compiler-time error occurs.

Coming back to the program control-flow graph of Fig. 5.23 on p. 480, observe that node 3 uses variable c, but on the path 1 2 3 no instruction executed before node 3

[32] Anticipating a computation to compiler time is termed *constant folding*.

assigns a value to c. This is not necessarily an error: if variable c is an input parameter of the subprogram, its value is supplied by the subprogram invocation statement. In such a case, variable c has a value on the entrance to node 3 and no error occurs. The same discussion applies to variable m.

Variable b is available on the entrance to node 3 because its value was defined in node 2 by an assignment, which uses variable a; the latter is in turn available on the entrance to 2, following the initialization in the node 1.

This sort of reasoning becomes intricate, and we need to clarify the concept of availability in the next definition (Definition 5.63). For simplicity we assume the subprogram does not have any input parameters.

Definition 5.63 (*variable availability*) A variable a is *available on the entrance to node* p, i.e., just before the execution of instruction p, if in the program control-flow graph every path from the initial node 1 to the entrance of p contains a statement that defines variable a. □

By comparing this notion with the concept of reaching definition introduced on p. 478, we notice a difference in the quantification over paths. If the definition a_q of variable a reaches the entrance of node p, there necessarily exists a path from node 1 to node p that traverses the defining instruction q. But this does not guarantee that variable a is available on the entrance to p, because we cannot exclude that there exists another path from 1 to p, which avoids node q as well as any other node that defines variable a.

It follows that the condition of a variable definition being available on a node entrance is more constraining than that of the variable definition reaching the same node.

To compute the variables available on the entrance to node p, let us examine more closely the set of definitions reaching the exits of the predecessors of p. If for every node q that is a predecessor of p, the set of reaching definitions $out\,(q)$ on the exit from q contains (at least) one definition of variable a, then variable a is available on the entrance to p. In that case we also say that *some definition* of a *always reaches* node p.

It is simple to convert the latter condition into an effective test that the variables are correctly initialized. For an instruction q, the new notation $out'\,(q)$ denotes, as $out\,(q)$ did, the set of the variable definitions reaching the exit from q, but with their subscripts deleted. For instance, for $out\,(q) = \{\, a_1,\ a_4,\ b_3,\ c_6 \,\}$ we have $out'\,(q) = \{\, a,\ b,\ c \,\}$.

Badly Initialized Variable

An instruction p *is not well initialized* if the following predicate holds:

$$\exists\, q \in pred\,(p) \ \text{ such that } \ use\,(p) \nsubseteq out'\,(q) \tag{5.14}$$

The condition says that there exists a node q predecessor of p, such that the set of definitions reaching its exit does not include all the variables used in p. Therefore,

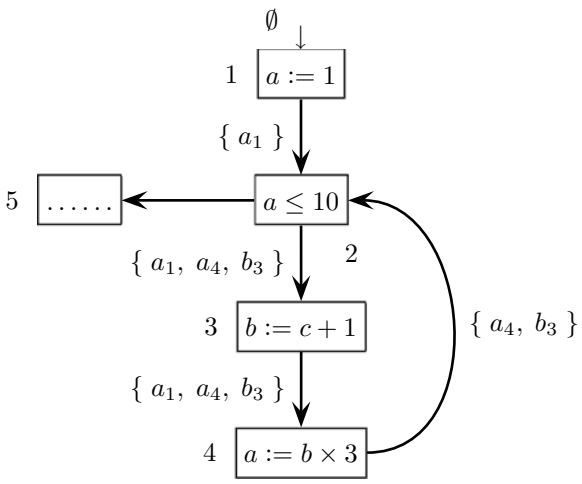


Fig. 5.26 Control-flow graph with the available variables for Example 5.64

when the program execution runs on a path through q, one or more variables used in p do not have a value. The next example (Example 5.64) illustrates the situation.

Example 5.64 (*detecting uninitialized variables*) Observe the program control-flow graph in Fig. 5.26, completed with the sets of reaching definitions. Condition (5.14) is false in the node 2, since for every predecessor (nodes 1 and 4) the set out' contains a definition of a, which is the only variable used in 2. On the other hand, the condition is true in the node 3 because no definition of variable c reaches the exit from 2. Our analysis has thus detected a program error: instruction 3 uses an uninitialized variable, namely c.

To find all the remaining initialization errors, we can proceed as follows. We replace any erroneous instructions discovered so far, such as node 3, by a dummy no-operation instruction. Then we update the computation of the reaching definition sets and we evaluate condition (5.14) again. By so doing, we would discover that instruction 4 is not well initialized because definition b_3 is not really available, although it is present in the set *out* (3), because instruction 3 has already been marked as ineffective. Then also instruction 4 becomes ineffective. Continuing in the same manner, no other errors would be discovered. □

The previous analysis allows to catch at compile time many errors that had gone unnoticed during the preceding phases of parsing and semantic analysis, with the benefit that they will not cause hard-to-understand run-time errors or raise exceptions during program execution.

To conclude, static analysis[33] encompasses many more conditions and properties than the cases of liveness and reaching definitions we have been able to present. It is a powerful general method for analyzing programs before execution, in order to optimize them or to verify their correctness.

References

1. Berstel J (1979) Transductions and context-free languages. Teubner, Stuttgart
2. Sakarovitch J (2009) Elements of automata theory. Cambridge University Press, Cambridge, England
3. Aho A, Ullman J (1972) The theory of parsing, translation, and compiling, volume 1: parsing. Prentice-Hall, Englewood Cliffs, NJ
4. Aho A, Ullman J (1973) The theory of parsing, translation and compiling, volume 2: compiling. Prentice-Hall, Englewood Cliffs, NJ
5. Yang W (1996) Mealy machines are a better model of lexical analyzers. Comput Lang 22:27–38
6. Tennent RD (1991) Semantics of programming languages. Prentice-Hall, Englewood Cliffs
7. Winskel G (1993) The formal semantics of programming languages. MIT Press, Cambridge, MA
8. Winskel G (1968) Semantics of context-free languages. Math Syst Theory 2(2):127–145
9. Reps TW, Teitelbaum T, Demers AJ (1983) Incremental context-dependent analysis for language-based editors. ACM TOPLAS 5(3):449–477
10. Knuth DE (1971) Semantics of context-free languages (errata corrige). Math Syst Theory 5(2):95–99
11. Engelfriet J (1984) Attribute grammars: attribute evaluation methods. In: Lorho B (ed) Methods and tools for compiler construction. Cambridge University Press, Cambridge, UK, pp 103–138
12. Engelfriet J, Filé G (1989) Passes, sweeps, and visits in attribute grammars. J ACM 36(4):841–869
13. Crespi Reghizzi S, Psaila G (1998) Grammar partitioning and modular deterministic parsing. Comput Lang 24(4):197–227
14. Aho A, Lam M, Sethi R, Ullman J (2006) Compilers: principles, techniques and tools. Prentice-Hall, Englewoof Cliffs, NJ
15. Appel A (2002) Modern compiler implementation in Java. Cambridge University Press, Cambridge, UK
16. Srikant YN, Shankar P (2002) Compiler design handbook: optimizations and machine code generation. CRC Press Inc, Boca Raton, FL, USA
17. Muchnick S (1997) Advanced compiler design and implementation. Morgan Kaufmann, San Francisco
18. Nielson F, Nielson HR, Hankin C (2010) Principles of program analysis. Springer, Berlin

[33] A book on the theory of static program analysis is [18]. For a survey of applications in compilation, see, e.g., [14,15,17].

Index

Symbols
2I-automaton, 416, 417
ε-move, 130

A
Abstract syntax tree, 45, 394, 429
Acceptance
  by empty stack, 202
  with final state, 207
Accepting mode of pushdown machine, 208
Accessible
  state, 124, 246
  subsets, 144
A-condition for attribute evaluation, 455
Acyclic attribute grammar, 440, 441
Acyclicity test, 441
Acyclic LST definition, 185
Algol 60, 60
Algorithm
  Berry-Sethi
    algorithm, 161–163
    parser, 172, 179
  bottom-up parser construction, *see* ELR(1) parser construction
  cleaning
    of a finite automaton, 124
    of a grammar, 39
  composition of local automata, 154
  conversion of translation grammar to postfix form, 409
  deterministic automaton from a regular expression construction, 158
  determinization of finite automaton, 163
  Earley parser, 358
  elimination
    of copy rules, 68
    of nullable nonterminals, 67
    of repeated right parts in grammar rules, 70
    of spontaneous moves, 143
  ELR(1) parser construction, 260
    pointerless, 294
    with vector stack, 272
  finite automaton complementation, 189
  finite automaton determinization, *see* powerset construction
  from grammar to nondeterministic pushdown automaton, 202
  from regular expression to deterministic finite automaton, *see* Berry-Sethi algorithm
  from regular expression to deterministic finite parser, *see* Berry-Sethi parser
  from regular expression to grammar, 82
  from unilinear grammar to regular expression, 87
  local recognizer construction, 152
  merge of kernel-equivalent m-states, 291
  minimization of finite deterministic automaton, 125
  normalization of a grammar
    into Chomsky form, 72
    into Greibach form, 79
    into operator form, 74

© Springer Nature Switzerland AG 2019

S. Crespi Reghizzi et al., *Formal Languages and Compilation*,
Texts in Computer Science, https://doi.org/10.1007/978-3-030-04879-2

into position restricted form, 80
into real-time form, 78
one-sweep evaluator construction, 443
operator precedence parallel parser costruction, 343
operator precedence sequential parser construction, 331
pilot graph construction, 252
powerset construction, 145
predictive parser construction
as DPDA, 311
by recursive procedures, 313
predictive pushdown transducer construction, 400
reduction
of a grammar, *see* cleaning of a grammar
of an automaton, *see* cleaning of a finite automaton
syntax tree construction, 367
top-down parser construction, *see* predictive parser construction
topological sorting, 440
transformation of grammar recursion
immediate left recursion, 76
non-immediate left recursion, 77
Alphabet, 8
unary, 92
Alphabetic homomorphism, 97, 388
Alternative, 19, 34
Ambiguity, 52
conditional instruction, 60, 392, 396
decision in r.e., 186
degree, 54
EBNF, 104
inherent, 62, 147, 221
in r.e., 170, 182
of automaton, 137, 221
of bilateral recursion, 55
of circular derivation, 395
of concatenation, 58
of regular expression, 22, 84, 165
of translation, 395
of union, 56
source grammar, 396
versus nondeterminism, 221
Ambiguity degree
of r.e., 170
ANTLR, 324, 352
Aperiodic language, 193
Arden identity, 88
Arithmetic expression, 42, 43, 53, 56, 63, 65, 69, 103
calculating machine, 453
infix, 404
machine net, 235
parenthesis-free, 86
polish, 393
syntax chart, 238
Artificial language, 5
Attribute, 428, 430
inherited, *see* right attribute
left, *see* left attribute
lexical, *see* lexical attribute
right, *see* right attribute
synthesized, *see* left attribute
Attribute grammar, 429, 457
A-condition, 455
acyclic, 440
applications, 457
definition, 435
L-condition, 450
multi-sweep, 446, 464
one-sweep, 442, 443
Attribute subgrammar, 446
Automaton
ambiguous, 137, 221
clean, 124
configuration, 118
equivalent, 119
finite, 120
finite deterministic, 122
for local language, 152
from regular expression, 147, 148, 150
generalized, 139
local, 152
minimization, 126
nondeterministic, 133
product, 190
real-time, 212
to regular expression, 139
two-way, 424
unidirectional, 118
with spontaneous moves, 135
Available variables, 483
Axiom, 33, 34

B

Back-end compiler, 461
Base of macro-state, 252
Berry, 161
Berry-Sethi, 147, 161

algorithm, 161–163
parser, 172, 179
Berstel, 150, 386
Binary operator, 391
Block structure, 31
BMC, 139
BNF grammar, 34, 277
Boolean grammar, 109
Bottom-up
attribute evaluation, 455
deterministic translation, 406
parser with attribute evaluator, 453
syntax analysis, 231
Brzozowski and McCluskey, 139
BS, *see* Berry-Sethi algorithm
BSP, *see* Berry-Sethi parser
BuildTree (BT), 366

C

Cancellation rule of Dyck, 49
Candidate, 243
identifier, 259
Cardinality of language, 8
Cartesian product of automata, 190
CF family, 84, 95, 99, 101, 209
closure properties, 51
Choice of a regular expression, 20
Chomsky
hierarchy, 36, 105
normal form, 36, 72
Circular derivation, 40, 395
Clean automaton, 124, 471
Cleaning
of a finite automaton, 124
of a grammar, 39
Cleaveland, 109
Closing mark, 27
Closure
function, 243
in Earley parser, 354
of macro-state, 252
Closure properties
of REG, 25
of CF, 51, 95
of CF and REG, 95
of DET, 216
of input-driven L., 227
under substitution, 99
under translation, 426
under transliteration, 99
Code generation, 460
Codes and ambiguity, 58
Coke-Younger-Kasami algorithm, 352
Compaction of pilot, 290
Compiler
decomposition, 446
language, 428
Complement
automaton, 188
of language, 13
of regular language, 188
Completion in Earley parser, 356
Computation, 133, 206
label, 122, 133
length, 133
of nondeterministic automaton, 133
of Turing machine, 119
Concatenation, 9
of context-free languages, 51, 96
of Dyck languages, 58
of languages, 12
Condensed skeleton tree, 45
Conditional instruction, 60, 392, 396
ambiguity, 60, 392, 396
Configuration
of automaton, 118
of pushdown automaton, 201, 206
of Turing machine, 119
Conflict
convergence, 251
reduce-reduce, 251
shift-reduce, 251
Conjunctive grammar, 109
Conservative approximation, 471
Constant
folding, 483
propagation, 478, 481
Context-dependent language, 105
Context-free
deterministic, 215, 216
grammar
definition, 34
from pushdown automaton, 210
introduction, 32
translation, 389
Context-sensitive language, 105
Control-flow graph, 467, 469
automaton, 469
language, 469
Control instruction translation, 461
Convergence conflict, 251, 253, 255
Convergent transitions, 253

Copy-free normal form, 67
Copy rule, 67
Correspondence between finite automaton and unilinear grammar, 137
Cross operation, 16

D
Dart, 306
Data-flow equations, 474, 479
  liveness, 474
  reaching definitions, 479
  solution, 474
Dead code, 483
Decidable language, 119
Decimal constant, 121, 123, 124, 135, 137
Decision algorithm, 116
Decompiler, 388
Decorated tree, 431, 433
Degree of operator, 391
Dependence graph, 433, 438, 442
  of decorated tree, 433, 438
  of semantic function, 438
Derivation, 37
  circular, 40
  EBNF, 103
  immediate, 21
  left, 46
  of regular expression, 21
  right, 46
  self-nesting, 91
Deterministic
  finite automaton, 122
  language, 215, 216
    unambiguity, 220
  language families comparison, 346
  pushdown automaton, 215
    subclasses, 221
  simple grammar, 223
Deterministic Finite Automaton (DFA), 122
Deterministic Pushdown Automaton (DPDA), 215
Determinization of automaton, 142, 162, 163
DET family, 215, 216, 348
Dictionary, 8
Difference of languages, 13
Digrams, 150
Directed Acyclic Graph (DAG), 178, 181
Distance of strings, 375
Distinctly parenthesized grammar, 50, 223
Distinguishability of states, 126
Document Type Definition (DTD), 228
Double Greibach normal form, 80
Double service lemma, 218
Dyck
  cancellation rule, 49
  language, 49, 84, 96
    concatenation, 58
    translation, 403
Dynamic error, 460

E
Earley
  algorithm, 352, 357
  closure, 354
  introductory example, 353
  nonterminal shift, 356
  parser, 357
    completeness, 361
    completion algorithm, 357
    computational complexity, 364
    correctness, 361
    function BuildTree, 367
    grammar ambiguity, 372
    grammar unambiguity, 373
    nullable nonterminals, 369
    optimization, 373
    syntax analysis algorithm, 358
    syntax tree construction, 366
    syntax tree construction complexity, 371
    terminal shift algorithm, 358
    vector, 357
  terminal shift, 355
  vector, 357
Early scanning, 277, 278
EBNF grammar, 100, 233
  ambiguity, 104
  derivation, 103
  translation, 405
Editing distance, 375
ELL(1)
  condition, 286
    direct, 306
    violation, 309
  parser
    direct construction, 305
    step by step construction, 288
  parsing, 282
  PCFG, 298
  violation remedy, 351
ELL(k)
  condition, 323

parser, 324
ELR(1)
condition, 254
grammar, 349
language, 349
parser
pointerless, 294
vector-stack, 271
parser construction, 252
ELR(k)
grammar, 349
language, 349
relation with ELR(1), 349
Empty string, 10
Encryption, 388
End-mark in conditional instructions, 396
Engelfriet, 446
Environment, 457
Epsilon move, 130
Equations of unilinear grammar, 87, 88
Equivalence
of finite automata, 123, 128
of grammars
generalized structural, 65
strong, 65
weak, 64
Equivalent regular expressions, 22
Error
dynamic, 374
objective, 374
recovery, 376, 377
panic mode, 377
panic mode with token insertion, 378
reporting, 376
semantic, 374
state, 123
static, 374
subjective, 374
syntactic, 374, 375
treatment, 373
type, 374
Expansion of nonterminal, 66
Extended regular expression, 188, 191
External attribute, 436

F

Finals, 150
Finding a word in text, 134
FIN family, 20
Finite
automaton, 120
deterministic, 122
left-linear grammar, 138
transducer, 421, 422
opposite passes, 424
with look-ahead, 425
transducer for BSP, 178
Finite-state family, 136
Flex, 425
Floyd, 219
operator precedence languages, 335, 337
Followers, *see* set of followers
Follow set, *see* set of followers
Formal language, 6
Free monoid, 15
Front-end compiler, 461
FSA, 120

G

General
automaton, 117
parser, 352
Generalized automaton, 139
Goal-oriented, *see* predictive
Grammar
ambiguity, 53
from ambiguous r.e., 84
attribute, 429
BNF, 34, 277
Boolean, 109
Chomsky classification, 105
clean, 39
cleaning, 39
conjunctive, 109
context-free, 32
context-sensitive, 105
EBNF, 100, 233
equivalent, 38
errors, 39
extended context-free, 100, 233
homogeneous, 36, 77
invertible, 69
left-linear, 85
linear, 84
marked BNF grammar rule, 236
non-left-recursive, 76
normal form, 37, 65
of regular language, 82
of Van Wijngarten, 109
operator, 74
parenthesized, 50, 223
representations, 35

right-linear, 85, 129
right-linearized, 239
simple deterministic, 223, 346
strictly unilinear, 86
target, 389
translation, 389, 399
type 0, 105
type 1, 105
type 2, 105
type 3, 85, 105
unilinear, 85
Graph
context-sensitive, 106
dependence, 433
local automaton, 152
parser control-flow, 298
pilot, 252
program control-flow, 468
reachability, 39
sibling, 442
state-transition, 120
syntax chart, 237
syntax tree, 43
Graphical computation for ELL
guide set, 317
prospect set, 316
set of initials, 315
Graphical method for ELL, 314
examples, 318
pilot look-ahead, 322
Greibach normal form, 36, 78
Guide
predicate, 464
set, 299, 306
graphical computation, 317
graphical method, 314

H

Handle, 245, 259
Heilbrunner grammar complement, 217
Hierarchical list, 29
Hierarchy of Chomsky, 105
Homomorphism
alphabetic, 97
nonalphabetical, 98
HTML, 228

I

ID family, 227
Incremental
compilation, 379
parser, 379
Infinite ambiguity degree
of r.e., 170
Infix
operator, 392
to postfix translation, 404
Inherent ambiguity, 62, 147, 221
Inherited attribute, *see* right attribute
Initial
of a string, 10
state uniqueness, 136
Initials, *see* set of initials
Input-driven
family, 227
language, 223, 225, 227
Instruction scheduling, 467
Interference between variables, 476
Intermediate
language, 461
representation, 461, 467
Internal attribute, 436
Interprocedural static analysis, 468
Intersection
of context-free and regular language, 213
of context-free languages, 96
of regular languages, 188, 190
Intraprocedural static analysis, 468
Inverse translation, 387
Invertible grammar, 69, 223
IO-automaton, 421
sequential, 422
Item, *see* candidate
Iteration operator, 16

J

JavaScript Object Notation (JSON), 48, 228, 329
input-driven model, 228
Jumps, 461

K

Kernel
equivalent macro-states, 253
of macro-state, 252, 253
Knuth, 244, 267, 282, 348

L

Language, 8
abstraction, 26
artificial, 5
complement, 13

conjunctive, 110
context-free, 32, 38, 105, 209
context-sensitive, 105, 106
decidable, 119
Dyck, 49
empty, 8
equation, 87
finite, 8
formal, 6
formalized, 5
generated, 37
infinite, 8
recursive derivations, 41
input-driven, 223, 225
local, 151
locally testable, 151
nullable, 10
recursive, 119
recursively enumerable, 119
regular, 20
replica, 94, 110
with center, 108, 110
source, 386
substitution of, 28
target, 99, 386
unary alphabet, 92
universal, 13
visibly pushdown, 223, 225
with three equal powers, 93, 106, 110
with two equal powers, 90
L-condition for attribute evaluation, 450
Least fixed point, 475
Left
attribute, 432, 435
derivation, 46
quotient, 17, 281
recursion, 36
elimination, 77
immediate, 76
LR(1), 282
non-immediate, 77
recursive grammar
top-down parsing, 285
recursive rule, 76
elimination, 76
Left-linear grammar, 85, 138
Leftmost derivation, *see* left derivation
Left/right terminal set, 327
Length of string, 9
Levenshtein distance, 375
Lex, 425
Lexeme, 447
Lexical
analysis, 447
attribute, 436, 448, 449
class, 447, 448
closed and open, 448
finite and non-, 448
level, 447
segmentation, 448
Linear
grammar, 84
language equation, 87
regular expression, 156
syntax tree segments, 176, 177
Linearized Syntax Trees (LST), 168, 169, 172, 181
construction, 181
Linguistic abstraction, 26
List
abstract, 27
concrete, 27
hierarchical, 29
with precedence, 29
with separators, 27
Live-in, 473
Liveness, 471, 474, 476
equations, 474
interval, 471, 473
Live-out, 473
Live variable, 471, 483
LL(1)
grammar, 346
relation with LR(1), 347
LL(2) example, 323
LL(k)
grammar, 347
relation with LR(k), 347
Local
automaton, 152
language, 151, 193, 469
automaton, 152
composition of, 153
normalized automaton, 153
set, 156
regular expression, 156
testability, 151
Locally testable language, 193
Longest prefix rule, 448
Look-ahead
extending, 275
increasing, 322

set, 243
graphical method, 322
LR(1)
family, 348
grammar, 348
transformation, 275, 279
parser
relation with ELR(1), 267
superflous features, 270
relation with DET, 348
LR(2) to LR(1), 277, 279
LR(k)
early scanning, 277
grammar, 275
left quotient, 281
transformation, 282
Lukasiewicz, 392

M

M.r.e., 166
Machine net, 233
Macro-state, 250
base, 252
closure, 252
kernel, 252
Marked
abstract syntax tree, 167
BNF grammar rule, 236, 277, 279
regular expression, 166
syntax tree, 168
Marked Abstract Syntax Tree (MAST), 167
Marked Syntax Tree (MST), 168
McNaughton, 194
Meaning, 428
Memory allocation, 476
Metagrammar, 34
Metalanguage of regular expression, 33
Minimal automaton, 125
Minimization of finite automaton, 126
Mirror reflection, 10
Mixfix operator, 392
M-state, *see* macro-state
Multiple transition property, 253
Multi-sweep semantic evaluator, 446

N

Nerode relation, 126
Nested structure, 31
Network of finite machines, 233
Nivat theorem, 419
Noncounting language, 193
Nondeterminism motivation, 131
Nondeterministic
automaton conversion to deterministic, 142, 162, 163
finite automaton, 130, 133, 136
pushdown automaton, 202, 203
union, 219
Nondeterministic Finite Automata (NFA), 133
Non-left-recursive grammar, 76
Non-LR(k), 282
Non-nullable normal form, 66, 67
Nonterminal, 32
alphabet, 34
expansion, 66
nullable
graphical method, 315
shift, 248
in Earley parser, 356
Normal form
Chomsky, 72
copy-free, 67
Greibach, 78
non-nullable, 66, 67
real-time, 78
without repeated right parts, 69
Nullable
language, 10
nonterminal, 66
regular expression, 157
Numbered
input alphabet, 158
regular expression, 156
Numerical constant, 121, 123, 124, 135, 137

O

One-sweep
attribute evaluation, 442
attribute grammar, 442, 443
semantic evaluator construction, 443
Opening mark, 27
Operator
binary, 391
degree, 391
infix, 392
mixfix, 392
postfix, 392
prefix, 392
unary, 392
variadic, 391
Operator grammar, 74

normal form, 36, 74
Operator precedence
grammar, 326
languages, 335, 337
relation, 326, 327
sequential parser, 330, 331
Opposite passes, 424
Optimization of program, 385, 467, 477, 481
Ordered Attribute Grammar (OAG), 446

P
Palindrome, 32, 50, 390
nondeterminism, 220
pushdown machine, 207
Panic mode error recovery, 377
Papert, 194
Parallel parser, 325, 343
Parenthesis
language, 47, 223
redundant, 395
Parenthesized
expression, 45
grammar, 50, 223
tree, 45, 366
Parser, 230
Berry-Sethi, 172, 179
choice criteria, 350
Coke-Younger-Kasami, 352
Earley, 357
for regular expressions, *see* Berry-Sethi
general, 352
local, 325, 326
parallel, 325, 339, 343
predictive, 310, 311
recursive descent, 313
semantics-directed, 352, 463
top-down and bottom-up, 231
top-down deterministic, 311
with attribute evaluation, 447
with attribute evaluator, 450
with translation, 402
Parser Control-Flow Graph (PCFG), 290, 298
direct construction, 305
Pilot
compaction, 290
construction, 252
graph, 252
machine of translator, 406
Pin, 150
PLZ-SYS, 464
Pointerless ELR(1) parser, 294
Polish notation, 56, 391, 392
Portability of compiler, 461
Position restricted normal form, 80
Postaccessible state, 124
Postfix
normal form, 408
operator, 392
Power
of language, 12
of string, 11
Powerset construction, 145
Precedence
of operators, 11
relation, 326
Prediction in Earley parser, 354
Predictive
parser, 310
direct construction, 305
parser automaton, 311
pushdown automaton, 202
pushdown transducer, 400
Prefix, 10
operator, 392
Prefix-free language, 12
Problematic regular expression, 171, 182
Product
machine, 190
of automata, 190
Production, 34
Program
analysis, 467
optimization, 385, 467, 477, 481
Projection, 98
Prospect set, 299, 306
graphical computation, 316
Pumping property, 89
Pure syntactic translation, 389
Pushdown automaton
accepting modes, 208
conversion to grammar, 210
definition, 205
determinism, 215
deterministic subclasses, 221
forms of nondeterminism, 215
on-line, 212
real-time, 206, 212, 222
time complexity, 204
Pushdown Automaton (PDA), 120, 200, 202, 205, 209
Pushdown IO-automaton, 397

Pushdown transducer, 397, 399, 400
from translation grammar, 399
nondeterministic, 400

Q
Quotient
of grammars, 281
of languages, 17

R
Rabin and Scott, 416
machine, 416
Rational translation, *see* regular translation
Reachable
nonterminal, 39
state, 124
Reaching definition, 477, 483
always, 484
Real-time
normal form, 78
property, 206, 212
pushdown automaton, 222
Recognition algorithm, 116
Recognize, *see* accept
Recursion bilateral, 55
Recursive
derivation, 41
descent, 449
parser, 284, 313
parser with attributes, 449
translator, 403
machine net, 233
semantic evaluator, 445
Recursive descent translator, 404
Reduce move, 245
Reduce-reduce conflict, 251, 255, 277
Reduce-shift conflict, *see* shift-reduce conflict
Reflection, 10
REG family, 20, 84, 95, 100
closure properties, 25
included in CF, 84
Register assignment, 477
Regular expression, 18
ambiguity, 22, 84, 85, 170
Berry-Sethi method, 147
extended, 24, 191
from automaton, 139
language defined, 22
linear, 156
metalanguage, 33
nullability, 157
numbered, 23, 156
problematic, 171
Thompson method, 147, 148
with complement, 188
with intersection, 188
Regular expression to automaton, 147
Berry-Sethi, 161
structural method, 148
Thompson, 148
Regular language, 20
intersection, 188
pumping property, 89
Regular translation, 414
expression, 416
Repeated right parts, 69
Replica, 94, 108, 110
Reps, 432
Retargeting, 461
Reversal, 10
as translation, 389
of context-free language, 52, 96
of language, 11
recognizer of, 132
Right
attribute, 432, 434, 435, 453
derivation, 46
recursion, 36
LR(1), 282
Right-linear grammar, 85, 129
Right-linearized grammar, 269, 312
Rightmost derivation, *see* right derivation
Rule
Chomsky normal, 36
copy, 36, 67
empty, 36
Greibach normal, 36
homogeneous, 36
homogeneous binary, 72
left-linear, 36
left-recursive, 36
linear, 36
recursive, 36
right-linear, 36
right-recursive, 36
self-embedding, 36
self-nesting, 36
subcategorization, 36
terminal, 36
with operators, 36
Run-time error, 460

S
Sakarovitch, 141, 191, 386
Scan in Earley parser, 355
Scanner, 447
Scheduling of instructions, 467
Segments of linear syntax tree, 176, 177
Self-nesting, 47
  derivation, 91
Semantic, 5, 383, 427, 428
  analysis, 428
  attribute, 428, 431
    lexical, 448
  check, 457
  error, 457
  evaluator, 428, 442
    multi-sweep, 446
    one-sweep, 442
    recursive, 445
  function, 430, 435
  interpretation, 64
  metalanguage, 428, 435
  predicate, 457
  procedure, 445
  rule, 430
  translation, 427
Semantics-directed parser, 352, 463
Sentence, 8
  ambiguous, 23
Sentential form, 38
Sequential
  function, 423
  transducer, 422
Set
  guide, 317
  look-ahead, 322
  nullable nonterminals, 315
  of followers, 161, 241
  of initials, 150, 242, 315
    graphical computation, 315
  prospect, 316
Sethi, 161
Set operations, 13
Shift
  move, 244
  terminal or nonterminal, 251
Shift-reduce
  conflict, 251, 255, 279
  parser with attribute evaluator, 453
  translator, 406
Sibling graph, 442
Simple deterministic grammar, 223, 346
Single-transition property, 286, 290
Single-valued translation, 396
Sink state, *see* trap state
Skeleton tree, 45
  condensed, 45
Slice, 175
Source
  grammar, 389
    ambiguity, 396
  language, 386
Spontaneous move, 130, 135, 205
  elimination, 142, 143
Stack
  candidate, 259
  macro-state, 259
  m-state, 259
Stack m-states (sms), 259
Star
  of context-free language, 51, 96
  operation, 14
  properties, 15
Star-free language, 193
State
  accessible, 124
  distinguishable, 125
  postaccessible, 124
  reachable, 124
  trap, 123
  useful, 124
  useless, 124
State-transition diagram, 120
  pushdown machine, 207
Static error, 457
Static program analysis, 460, 467, 486
  interprocedural, 468
  intraprocedural, 468
STP, *see* single-transition property
Strictly unilinear grammar, 86
String, 8
  empty, 10
  form, 37
  matching, 165
Strong equivalence of grammars, 65
Structural adequacy, 64, 86
Subcategorization rule, 67
Subjacent automaton, 418, 422, 423, 426
  pushdown, 397
Substitution, 28, 98
  closure property, 99
Substring, 10
Suffix, 10

Symbol table, 374, 457
Syntactic
  analysis, 230
  error, 375
  level, 447
  procedure, 283, 313
  semantic analyzer, 450
  support, 430, 435
  translation, 389
    scheme, 389
Syntactic-semantic analysis, 447
Syntax, 5, 6
  abstract, 26
  analysis, 230
    bottom-up and top-down, 231
  chart, 237, 469
  tree, 43
    abstract, 394, 429
    as translation, 412
Syntax-directed
  compiler, 428
  translation, 148, 389, 428
Synthesized attribute, *see* left attribute

T
Target
  grammar, 389
  language, 99, 386
Terminal
  alphabet, 34
  shift in Earley parser, 355
  symbol, 8
Thompson, 147, 148
Token, 447
TOL, 444
Tomita, 352
Top-down
  deterministic translation, 403
  parsing, 282
  syntax analysis, 231
Topological sort, 440
TOR, 443
TOS, 443
Transducer, *see* translator
Transduction, *see* translation
Transition networks, 237
Translation
  closure properties, 426
  function, 386, 387, 423
    sequential, 423
  grammar, 389, 399, 420
    EBNF, 405
    postfix normal form, 408
    rule normalization, 399
    to pushdown transducer, 399
  infix to postfix, 404
  rational, *see* regular translation
  regular, *see* regular translation
  relation, 386
  scheme, 389
  single-valued, 396
  syntactic, 389
Translator
  Berry-Sethi, 172, 179
  bottom-up, 406
  comparison, 413
  top-down, 403
  two-way, 424
  with recursive procedures, 403
Transliteration, 97, 388
  to words, 98
Trap state, 123
Tree
  construction, 366
  decorated, 431
  of regular expression, 165
  of regular phrase, 165
  pattern matching, 461
  syntax, 43
Turing machine, 105, 119
Two-input automaton, 416, 417
Two-pass compiler, 429
Type checking, 458
Type 0 grammar, 105
Type 1 grammar, 105
Type 2 grammar, 105
Type 3 grammar, 85, 105

U
Unary
  alphabet, 92
  operator, 392
Undecidability
  deterministic context-free language, 350
  deterministic pushdown automaton, 350
  LR(k), 350
Undistinguishability of states, 126
Unilinear grammar, 85, 136, 137
  equations, 87
Uninitialized variable, 485
Union of context-free languages, 51, 96
Universal language, 13, 15

Unreachable code, 471, 483
Useful state, 124
Useless
  assignment, 472
  state, 124
    elimination, 124
  variable definition, 477
Uzgalis, 109

V
Van Wijngarten, 109
Variable
  availability, 483
  definition, 468
    suppressed, 479
  initialization, 483, 484
  use, 469
Variadic operator, 391
Vector-stack parser, 271
Vocabulary, 8
VW grammar, 109

W
Weak equivalence of grammars, 64
Well-defined nonterminal, 39
Word, 8

X
XML, 47, 228

Y
Yang, 425