

Texts in Computer Science



Stefano Crespi Reghizzi
Luca Breveglieri
Angelo Morzenti

Formal Languages and Compilation

Second Edition



Springer

Texts in Computer Science

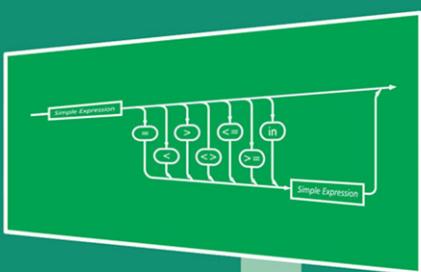
Editors

David Gries

Fred B. Schneider

For further volumes:

www.springer.com/series/3191



Stefano Crespi Reghizzi · Luca Breveglieri ·
Angelo Morzenti

Formal Languages and Compilation

Second Edition



Springer

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

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

Series Editors

David Gries
Department of Computer Science
Cornell University
Ithaca, NY, USA

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

Fred B. Schneider
Department of Computer Science
Cornell University
Ithaca, NY, USA

ISSN 1868-0941
Texts in Computer Science
ISBN 978-1-4471-5513-3
DOI 10.1007/978-1-4471-5514-0
Springer London Heidelberg New York Dordrecht

ISSN 1868-095X (electronic)
ISBN 978-1-4471-5514-0 (eBook)

© Springer-Verlag London 2009, 2013

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. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

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.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

The textbook derives from the well-known identically titled volume¹ published in 2009: two younger co-authors have brought their experience to enrich and streamline the book, without dulling its original structure. The selection of materials and the presentation style have not substantially changed. The book reflects many years of teaching compiler courses and of doing research on formal language theory and formal methods, on compiler and language technology, and to a lesser extent on natural language processing. The more important change concerns the central topic of language parsing. It is a completely new, systematic, and unified presentation of the most important parsing algorithms, including also parallel parsing.

Goals In the turmoil of information technology developments, the subject of the book has kept the same fundamental principles since half a century, and has preserved its conceptual importance and practical relevance. This state of affairs in a topic that is central to computer science 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 mathematics and 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 the application to compiling, 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 features many different aspects ranging from algorithmic wisdom to computer hardware. As a consequence, the textbooks have grown in size, and compete with respect to their coverage of the last developments on programming languages, processor architectures and clever mappings from the former to the latter.

To put things in order, it is better to separate such complex topics into two parts, *basic* and *advanced*, which to a large extent correspond to the two subsystems that make a compiler: the user-language specific front-end, and the machine-language 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 (imperative, functional, object oriented, etc.), the computer architecture related aspects, and the optimization methods used to improve the machine code produced by the compiler.

¹S. Crespi Reghizzi, *Formal Languages and Compilation* (Springer, London, 2009).

Organization and Features In other textbooks the bias towards 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 extensive use, and from avoiding the irritating variants and repetitions that are found in the original papers. Moving from these premises, we decided to present, in a simple minimalist way, the principles and methods used in designing language syntax and syntax-directed translators.

Chapter 2 covers regular expressions and context-free grammars, with emphasis on the structural adequacy of syntactic definitions and a careful analysis of ambiguity and how to avoid it.

Chapter 3 presents finite-state recognizers and their conversion back and forth to regular expressions and grammars.

Chapter 4 presents push-down machines and parsing algorithms, focusing attention on the *LL*, *LR* and Earley methods. We have substantially improved the standard presentation of such algorithms, by unifying the concepts and notations used in various approaches, and by extending the method coverage with a reduced definitional apparatus. An example that expert readers and instructors should appreciate, is the unification of the top-down (*LL*) and bottom-up (*LR*) parsing algorithms, as well as the tabular (Early) one, within a novel practical framework. In this way, the effort and space spared have made room for advanced methods typically not present in similar textbooks. First, our parsing algorithms apply to the Extended BNF grammars, which are the de facto standard in the language reference manuals. Second, we provide a parallel parsing algorithm that takes advantage of the many processing units of modern microprocessors, to speed-up the analysis of large files.

The book is not restricted to syntax: Chap. 5, the last one, studies translations, semantic functions (attribute grammars), and the static program analysis by data flow equations. This provides a comprehensive understanding of the compilation process, and covers the essential elements of a syntax-directed translator.

The presentation is illustrated by many small yet realistic examples and pictures, 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, without insisting too much on their formal definition. Algorithms are described in a pseudo-code to avoid the disturbing details of a programming language, yet they are straightforward to convert to executable procedures.

This book should be welcome by those willing to teach or to learn the essential concepts of syntax-directed compilation, without the need of relying on software tools and implementations. We believe that learning by doing is not always the best approach, and that over-commitment to practical work may sometimes obscure the conceptual foundations. In the case of formal languages, the elegance and simplicity of the underlying theory allows the 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, the 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.

Intended Audiences This is primarily a textbook targeted to *graduate* (or upper-division undergraduate) *students* in computer science or computer engineering. We list as prerequisites: familiarity with some programming language models and with algorithm design; and the ability to use elementary mathematical and logical notation. If the reader or student has already taken a course on theoretical computer science including a few elements of formal language and automata theory, the time and effort needed to learn the first chapters can be reduced. But it is fair to say that this book does not compete with the many available books on the theory of formal languages and computation: we usually do not include mathematical proofs of properties, and we rely instead on examples and informal arguments. Yet mathematically oriented readers will find here many motivating examples and applications.

A large collection of *problems and solutions* complementing the numerous examples in the book is available on the authors' course web site at Politecnico di Milano. Similarly, a comprehensive set of *lecture slides* is also available on the course web site.

The Authors Thank The colleagues who have taught this book to computer engineering students, Giampaolo Agosta and Licia Sbattella; the Italian National Research Group on Formal Languages, in particular Antonio Restivo, Alessandra Cherubini, Dino Mandrioli, Matteo Pradella and Pierluigi San Pietro; our past and present Ph.D. students and teaching assistants, Alessandro Barenghi, Marcello Bersani, Andrea Di Biagio, Simone Campanoni, Silvia Lovergine, Michele Scandale, Ettore Speziale, Martino Sykora and Michele Tartara. We also acknowledge the support of ST Microelectronics Company for R.&D. projects on compilers for advanced microprocessor architectures.

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

Milan, Italy
July 2013

Stefano Crespi Reghizzi
Luca Breveglieri
Angelo Morzenti

Contents

1	Introduction	1
1.1	Intended Scope and Audience	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	7
2.2.1	Alphabet and Language	7
2.2.2	Language Operations	11
2.2.3	Set Operations	13
2.2.4	Star and Cross	14
2.2.5	Quotient	16
2.3	Regular Expressions and Languages	17
2.3.1	Definition of Regular Expression	17
2.3.2	Derivation and Language	19
2.3.3	Other Operators	22
2.3.4	Closure Properties of <i>REG</i> Family	23
2.4	Linguistic Abstraction	24
2.4.1	Abstract and Concrete Lists	25
2.5	Context-Free Generative Grammars	28
2.5.1	Limits of Regular Languages	29
2.5.2	Introduction to Context-Free Grammars	29
2.5.3	Conventional Grammar Representations	32
2.5.4	Derivation and Language Generation	33
2.5.5	Erroneous Grammars and Useless Rules	35
2.5.6	Recursion and Language Infinity	37
2.5.7	Syntax Trees and Canonical Derivations	38
2.5.8	Parenthesis Languages	41
2.5.9	Regular Composition of Context-Free Languages	43
2.5.10	Ambiguity	45
2.5.11	Catalogue of Ambiguous Forms and Remedies	47

2.5.12 Weak and Structural Equivalence	54
2.5.13 Grammar Transformations and Normal Forms	56
2.6 Grammars of Regular Languages	67
2.6.1 From Regular Expressions to Context-Free Grammars . .	67
2.6.2 Linear Grammars	68
2.6.3 Linear Language Equations	71
2.7 Comparison of Regular and Context-Free Languages	73
2.7.1 Limits of Context-Free Languages	76
2.7.2 Closure Properties of <i>REG</i> and <i>CF</i>	78
2.7.3 Alphabetic Transformations	79
2.7.4 Grammars with Regular Expressions	82
2.8 More General Grammars and Language Families	85
2.8.1 Chomsky Classification	86
References	90
3 Finite Automata as Regular Language Recognizers	91
3.1 Introduction	91
3.2 Recognition Algorithms and Automata	92
3.2.1 A General Automaton	93
3.3 Introduction to Finite Automata	96
3.4 Deterministic Finite Automata	97
3.4.1 Error State and Total Automata	98
3.4.2 Clean Automata	99
3.4.3 Minimal Automata	100
3.4.4 From Automata to Grammars	103
3.5 Nondeterministic Automata	104
3.5.1 Motivation of Nondeterminism	105
3.5.2 Nondeterministic Recognizers	107
3.5.3 Automata with Spontaneous Moves	109
3.5.4 Correspondence Between Automata and Grammars . .	110
3.5.5 Ambiguity of Automata	111
3.5.6 Left-Linear Grammars and Automata	112
3.6 Directly from Automata to Regular Expressions: BMC Method .	112
3.7 Elimination of Nondeterminism	114
3.7.1 Construction of Accessible Subsets	115
3.8 From Regular Expression to Recognizer	119
3.8.1 Thompson Structural Method	119
3.8.2 Berry–Sethi Method	121
3.9 Regular Expressions with Complement and Intersection	132
3.9.1 Product of Automata	134
3.10 Summary of Relations Between Regular Languages, Grammars, and Automata	138
References	139

4 Pushdown Automata and Parsing	141
4.1 Introduction	141
4.2 Pushdown Automaton	142
4.2.1 From Grammar to Pushdown Automaton	143
4.2.2 Definition of Pushdown Automaton	146
4.3 One Family for Context-Free Languages and Pushdown Automata	151
4.3.1 Intersection of Regular and Context-Free Languages	153
4.3.2 Deterministic Pushdown Automata and Languages	154
4.4 Syntax Analysis	162
4.4.1 Top-Down and Bottom-Up Constructions	163
4.5 Grammar as Network of Finite Automata	165
4.5.1 Derivation for Machine Nets	171
4.5.2 Initial and Look-Ahead Characters	173
4.6 Bottom-Up Deterministic Analysis	176
4.6.1 From Finite Recognizers to Bottom-Up Parser	177
4.6.2 Construction of <i>ELR(1)</i> Parsers	181
4.6.3 <i>ELR(1)</i> Condition	184
4.6.4 Simplifications for <i>BNF</i> Grammars	197
4.6.5 Parser Implementation Using a Vector Stack	201
4.6.6 Lengthening the Look-Ahead	205
4.7 Deterministic Top-Down Parsing	211
4.7.1 <i>ELL(1)</i> Condition	214
4.7.2 Step-by-Step Derivation of <i>ELL(1)</i> Parsers	217
4.7.3 Direct Construction of Top-Down Predictive Parsers	231
4.7.4 Increasing Look-Ahead	240
4.8 Deterministic Language Families: A Comparison	242
4.9 Discussion of Parsing Methods	246
4.10 A General Parsing Algorithm	248
4.10.1 Introductory Presentation	249
4.10.2 Earley Algorithm	252
4.10.3 Syntax Tree Construction	261
4.11 Parallel Local Parsing	268
4.11.1 Floyd's Operator-Precedence Grammars and Parsers	269
4.11.2 Sequential Operator-Precedence Parser	274
4.11.3 Parallel Parsing Algorithm	277
4.12 Managing Syntactic Errors and Changes	283
4.12.1 Errors	284
4.12.2 Incremental Parsing	289
References	290
5 Translation Semantics and Static Analysis	293
5.1 Introduction	293
5.1.1 Chapter Outline	294
5.2 Translation Relation and Function	295
5.3 Transliteration	298

5.4	Purely Syntactic Translation	298
5.4.1	Infix and Polish Notations	300
5.4.2	Ambiguity of Source Grammar and Translation	303
5.4.3	Translation Grammars and Pushdown Transducers	305
5.4.4	Syntax Analysis with Online Translation	310
5.4.5	Top-Down Deterministic Translation by Recursive Procedures	311
5.4.6	Bottom-Up Deterministic Translation	313
5.4.7	Comparisons	320
5.5	Regular Translations	321
5.5.1	Two-Input Automaton	323
5.5.2	Translation Functions and Finite Transducers	327
5.5.3	Closure Properties of Translations	332
5.6	Semantic Translations	333
5.6.1	Attribute Grammars	335
5.6.2	Left and Right Attributes	338
5.6.3	Definition of Attribute Grammar	341
5.6.4	Dependence Graph and Attribute Evaluation	343
5.6.5	One Sweep Semantic Evaluation	347
5.6.6	Other Evaluation Methods	351
5.6.7	Combined Syntax and Semantic Analysis	352
5.6.8	Typical Applications of Attribute Grammars	360
5.7	Static Program Analysis	369
5.7.1	A Program as an Automaton	370
5.7.2	Liveness Intervals of Variables	373
5.7.3	Reaching Definitions	380
	References	387
	Index	389

1.1 Intended Scope and Audience

The information technology revolution was made possible by the invention of electronic digital machines, but 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 pervasively used in all branches of informatics to model situations or phenomena classifiable as time and space discrete systems. Formal grammars on the other hand originated in linguistic research and are widely applied in document processing, in particular for the Web.

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

The selection of topics and the presentation based on rigorous definitions and algorithms illustrated by many motivating examples should qualify 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 about 50 hours of lecturing suffice to cover the entire book.

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

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 classical *Flex* and *Bison*), 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 syntax-directed paradigm. 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 to machine instructions and for transforming a program in order to make the best use of the computational resources of a computer. The study of program transformation and optimization methods 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 generally 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 grammars. Both belong to formal language theory, a well-established chapter of theoretical computer science.

The first task of a compiler is to check the correctness of the source text, that is, whether it complies with the syntactic definition of the source language by certain grammar rules. In order to perform the 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 approach, such recognition algorithms can be conveniently described as mathematical machines or automata, in the tradition of the well-known Turing machine.

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 keywords, numbers, and in general the pieces of text corresponding to the lexical units or lexemes (or tokens) of the language.

Chapter 4 is devoted to the recognition problem for languages defined by context-free grammars. Recognition algorithms are first modeled as finite automata equipped with unbounded last-in-first-out memory or pushdown stack. For a compiler, the language recognizer is an essential component known as the syntax analyzer or parser. Its job is to check the syntactic correctness of a source text already subdivided into lexemes, and to construct a structural representation called a syntax tree. To reduce the parsing time for very long texts, the parser can be organized as a parallel program.

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

The formal models of translation and the methods used to implement semantic analyzers are in Chap. 5, which describes two kinds of transformations. Pure syntactic translations are modeled by finite or pushdown transducers. Semantic translations are performed by functions or methods that operate on the syntax tree of the source text. Such translations will be specified by a practical extension to context-free grammars, called attribute grammars. This approach, by combining the accuracy of formal syntax and the flexibility of programming, conveniently expresses the analysis and translation of syntax trees.

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

For sure compilers do much more than syntax-directed translation. Static program analysis is an important example, consisting in examining a program to determine, ahead of execution, some properties, or to detect errors not covered by 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. 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.

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 of them 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, to control a robot, and so on. Other languages serve as interfaces between devices, e.g., Postscript is a language produced by a text processor commanding a printer.

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.

For a language to be formalized (or formal), the form of sentences (or syntax) and their meaning (or semantics) must be precisely and algorithmically defined. In other words, it should be possible for a computer to check that sentences are grammatically correct, and to determine their meaning.

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

In this book the term *formal language* is used in a narrower sense that 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 only concerned with the form or syntax of sentences, not with meaning. A string (or text) is either valid or illegal, that is, it either belongs to the formal language or does not. Such theory makes a first important step towards the ultimate goal: the study of language translation and meaning, which will require additional methods.

2.1.2 Language Types

A language in this book 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 not 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 designers of buildings and objects 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 psychological theories. The formal approach to the syntax of this chapter has some interest for non-textual languages too.

Within computer science, the term language applies 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 discrete picture. Formal language theories have been proposed and used to various degrees also for such non-textual languages.¹

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 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. Clearly the exhaustive approach, to list all possible sentences or phrases, is unfeasible because the possibilities are infinite, since the length of sentences is usually unbounded. As a native language speaker, a programmer is not constrained by any strict limit on the length of phrases to be written. The 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).

¹ Just two examples and references: tree languages [5] and picture (or two-dimensional) languages [4, 6].

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 definition of the family of regular languages comes next.

Then the lists are introduced as a fundamental and pervasive syntax structure in all 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 transformations of rules (normal forms) are introduced, which should familiarize the reader with the modifications often needed for technical applications, to adjust a grammar without affecting the language it defines.

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 of 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 this rarely used model.

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 $\Sigma = \{a_1, a_2, \dots, a_k\}$ 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 Let $\Sigma = \{a, b\}$ be the alphabet. Some strings are: *aaba*, *aaa*, *abaa*, *b*.

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

Example 2.2 For the same alphabet $\Sigma = \{a, b\}$ three examples of languages follow:

$$L_1 = \{aa, aaa\}$$

$$L_2 = \{aba, aab\}$$

$$L_3 = \{ab, ba, aabb, abab, \dots, aaabbb, \dots\}$$

= set of strings having as many *a*'s as *b*'s

Notice that a formal language viewed as a set has two layers: at the first level there is an unordered set of non-elementary 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 *baaa* $\in L_3$ is a sentence of L_3 , whereas *abb* $\notin L_3$ is an *incorrect* string.

The *cardinality* or size of a language is the number of sentences it contains. For instance, $|L_2| = |\{aba, aab\}| = 2$. 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, L_1 and L_2 are finite, but L_3 is infinite.

One can observe a finite language is essentially a collection of words² sometimes called a *vocabulary*. A special finite language is the *empty* set or *language* \emptyset , which contains no sentence, $|\emptyset| = 0$. Usually, when a language contains just one element, the set braces are omitted writing e.g., *abb* instead of $\{abb\}$.

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

$$|aab|_a = 2, \quad |aba|_a = 2, \quad |baa|_c = 0$$

The *length* $|x|$ of a string *x* is the number of characters it contains, e.g.: $|ab| = 2$; $|abaa| = 4$.

Two strings

$$x = a_1 a_2 \dots a_h, \quad y = b_1 b_2 \dots b_k$$

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

$$aba \neq baa, \quad baa \neq ba$$

²In mathematical writings the terms *word* and *string* are synonymous, in linguistics a word is a string having a meaning.

2.2.1.1 String Operations

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

$$x = a_1 a_2 \dots a_h, \quad y = b_1 b_2 \dots b_k$$

*concatenation*³ is defined as

$$x.y = a_1 a_2 \dots a_h b_1 b_2 \dots b_k$$

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

Example 2.3 For strings

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

we obtain

$$xy = \text{wellin}, \quad yx = \text{inwell} \neq xy$$

$$(xy)z = \text{wellin形成的} = x(yz) = \text{well.informed} = \text{wellinformed}$$

Concatenation is clearly non-commutative, that is, the identity $xy \neq yx$ does not hold in general. The *associative* property holds:

$$(xy)z = x(yz)$$

This 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:

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

2.2.1.2 Empty String

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

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

for every string x . From equality (2.1) it follows the empty string has length zero:

$$|\varepsilon| = 0$$

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

³Also termed *product* in mathematical works.

The empty string should not be confused with the empty set: in fact \emptyset as a language contains no string, whereas the set $\{\varepsilon\}$ contains one, the empty string.

A language L is said to be *nullable* iff it includes the empty string, i.e., iff $\varepsilon \in L$.

2.2.1.3 Substrings

Let $x = u y v$ be the concatenation of some, possibly empty, strings u, y, v . Then y is a *substring* of x ; moreover, u is a *prefix* of x , and v is a *suffix* of x . A substring (prefix, suffix) is called *proper* if it does not coincide with string x .

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

Example 2.4 The string $x = aabacba$ contains the following components:

prefixes: $a, aa, aab, aaba, aabac, aabacb, aabacba$

suffixes: $a, ba, cba, acba, bacba, abacba, aabacba$

substrings: all prefixes and suffixes and the internal strings such as $a, ab, ba, bacb, \dots$

Notice that bc is not a substring of x , although both b and c occur in x . The initial of length two is $Ini_2(aabacba) = aa$.

2.2.1.4 Mirror Reflection

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

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

The following identities are immediate:

$$(x^R)^R = x \quad (xy)^R = y^R x^R \quad \varepsilon^R = \varepsilon$$

2.2.1.5 Repetitions

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

$$x^m = \underbrace{xx \dots x}_{m \text{ times}}$$

By stipulation the zero power of any string is defined to be the empty string.

The complete definition is

$$\begin{cases} x^m = x^{m-1}x, & m > 0 \\ x^0 = \varepsilon & \end{cases}$$

Examples:

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

When writing formulas, the string to be repeated must be parenthesized, if longer than one. Thus to express the second power of ab , i.e., $abab$, one should write $(ab)^2$, not ab^2 , which is the string abb .

Expressed differently, we assume the power operation takes *precedence* over concatenation. Similarly reflection takes precedence over concatenation: e.g., ab^R returns ab , since $b^R = b$, while $(ab)^R = ba$.

2.2.2 Language Operations

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

The reflection of a language L is the set of strings that are the reflection of a sentence:

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

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

Similarly the set of proper prefixes of a language L is

$$\text{Prefixes}(L) = \{ y \mid x = yz \wedge x \in L \wedge y \neq \varepsilon \wedge z \neq \varepsilon \}$$

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 inadvertent truncation of a sentence.

A language is *prefix-free* if none of the proper prefixes of sentences is in the language; i.e., if the set $\text{Prefixes}(L)$ is disjoint from L .

Thus the language $L_1 = \{x \mid x = a^n b^n \wedge n \geq 1\}$ is prefix-free since every prefix takes the form $a^n b^m$, $n > m \geq 0$ 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 its prefix $a^3 b$.

Similarly, operations on two strings can be extended to two languages, by letting the first and second argument span the respective language, for instance *concatenation*:

tion of languages L' and L'' is defined as

$$L'L'' = \{xy \mid x \in L' \wedge y \in L''\}$$

From this the extension of the m th power operation on a language is straightforward:

$$\begin{aligned} L^m &= L^{m-1}L, \quad m > 0 \\ L^0 &= \{\varepsilon\} \end{aligned}$$

Some special cases follow from previous definitions:

$$\begin{aligned} \emptyset^0 &= \{\varepsilon\} \\ L.\emptyset &= \emptyset.L = \emptyset \\ L.\{\varepsilon\} &= \{\varepsilon\}.L = L \end{aligned}$$

Example 2.6 Consider the languages:

$$\begin{aligned} L_1 &= \{a^i \mid i \geq 0, \text{ even}\} = \{\varepsilon, aa, aaaa, \dots\} \\ L_2 &= \{b^j a \mid j \geq 1, \text{ odd}\} = \{ba, bbba, \dots\} \end{aligned}$$

We obtain

$$\begin{aligned} L_1 L_2 &= \{a^i.b^j a \mid (i \geq 0, \text{ even}) \wedge (j \geq 1, \text{ odd})\} \\ &= \{\varepsilon ba, a^2 ba, a^4 ba, \dots, \varepsilon b^3 a, a^2 b^3 a, a^4 b^3 a, \dots\} \end{aligned}$$

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

$$\{x \mid x = y^m \wedge y \in L\} \subseteq L^m, \quad m \geq 2$$

Thus for $L = \{a, b\}$ with $m = 2$ the left part is $\{aa, bb\}$ and the right part is $\{aa, ab, ba, bb\}$.

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

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

may also be defined as

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

Notice that sentences shorter than k are obtained using the empty string of the base language.

Slightly changing the example, the language $\{x \mid 1 \leq |x| \leq 3\}$ is defined, using concatenation and power, by the formula

$$L = \{a, b\}\{\varepsilon, a, b\}^2$$

2.2.3 Set Operations

Since a language is a set, the classical set operations, union (\cup), intersection (\cap), and difference (\setminus), apply to languages; set relations, 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 strings of 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:

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

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

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

that is, the set of the strings of 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,

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

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

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

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, on the other side by the complement of the set of even-length strings with alphabet $\{a\}$:

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

Moving to set difference, consider 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 differences are

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

which represents the set of strings having the same number, excluding 1, of occurrences of letters a, b, c :

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

the set of strings having one c and the same number of occurrences of a, b , excluding 1.

2.2.4 Star and Cross

Most artificial and natural languages include sentences that can be lengthened at will, 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.

The *star*⁴ operation is defined as the union of all the powers of the base language:

$$L^* = \bigcup_{h=0...∞} L^h = L^0 \cup L^1 \cup L^2 \cup \dots = \varepsilon \cup L \cup L^2 \cup \dots$$

Example 2.9 For $L = \{ab, ba\}$

$$L^* = \{\varepsilon, ab, ba, abab, abba, baab, baba, \dots\}$$

Every string of the star can be segmented into substrings which are sentences of the base language L .

Notice that starting with a finite base language, L , the “starred” language L^* is infinite.

It may happen that the starred and base language are identical as in

$$L = \{a^{2n} \mid n \geq 0\} \quad L^* = \{a^{2n} \mid n \geq 0\} \equiv L$$

An interesting special case is when the base is an alphabet Σ , then the star Σ^* contains all the strings⁵ obtained by concatenating terminal characters. This language is the same as the previous *universal language* of alphabet Σ .⁶

⁴Also known as Kleene’s star and as closure by concatenation.

⁵The length of a sentence in Σ^* is unbounded but it may not be considered to be infinite. A specialized branch of this theory (see Perrin and Pin [11]) is devoted to so-called infinitary or omega-languages, which include also sentences of infinite length. They effectively model the situations when an eternal system can receive or produce messages of infinite length.

⁶Another 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).

It is clear that any formal language is a subset of the universal language of the same alphabet; the relation:

$$L \subseteq \Sigma^*$$

is often written to say that L is a language of alphabet Σ .

A few useful properties of the star:

$L \subseteq L^*$	(monotonicity)
if $(x \in L^* \wedge y \in L^*)$ then $xy \in L^*$	(closure by concatenation)
$(L^*)^* = L^*$	(idempotence)
$(L^*)^R = (L^R)^*$	(commutativity of star and reflection)

Example 2.10 (Idempotence) The monotonicity property affirms any language is included in its star. But for language $L_1 = \{a^{2n} \mid n \geq 0\}$ the equality $L_1^* = L_1$ follows from the idempotence property and the fact that L_1 can be equivalently defined by the starred formula $\{aa\}^*$.

For the empty language and empty string we have the identities:

$$\emptyset^* = \{\varepsilon\}, \quad \{\varepsilon\}^* = \{\varepsilon\}$$

Example 2.11 (Identifiers) Many artificial languages assign a name or identifier to each entity (variable, file, document, subprogram, object, etc.). A usual naming rule prescribes that an identifier should be a string with initial character in $\{A, B, \dots, Z\}$ and containing any number of letters or digits $\{0, 1, \dots, 9\}$, such as CICLO3A2.

Using the alphabets

$$\Sigma_A = \{A, B, \dots, Z\}, \quad \Sigma_N = \{0, 1, \dots, 9\}$$

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

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

To introduce a variance, prescribe that the length of identifiers should not exceed 5. Defining $\Sigma = \Sigma_A \cup \Sigma_N$, the language is

$$I_5 = \Sigma_A(\Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \Sigma^4) = \Sigma_A(\varepsilon \cup \Sigma \cup \Sigma^2 \cup \Sigma^3 \cup \Sigma^4)$$

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

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

Cross A useful though dispensable operator, derived from the star, is the *cross*:⁷

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

It differs from the star because the union is taken excluding power zero. The following relations hold:

$$\begin{aligned} L^+ &\subseteq L^* \\ \varepsilon \in L^+ &\quad \text{if and only if} \quad \varepsilon \in L \\ L^+ &= LL^* = L^*L \end{aligned}$$

Example 2.12 (Cross)

$$\begin{aligned} \{ab, bb\}^+ &= \{ab, b^2, ab^3, b^2ab, abab, b^4, \dots\} \\ \{\varepsilon, aa\}^+ &= \{\varepsilon, a^2, a^4, \dots\} = \{a^{2n} \mid n \geq 0\} \end{aligned}$$

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

Example 2.13 The strings four or more characters long may be defined by:

- concatenating the strings of length four with arbitrary strings: $\Sigma^4 \Sigma^*$
- or by constructing the fourth power of the set of nonempty strings: $(\Sigma^+)^4$

2.2.5 Quotient

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

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

Example 2.14 Let

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

The quotients are

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

⁷Or nonreflective closure by concatenation.

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

Other 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 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 circuit⁸ 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, using the union, concatenation, and star operations; then the family will be defined by certain simple grammar rules; last, Chap. 3 describes the algorithm for recognizing regular languages in the form of an abstract machine or automaton.⁹

2.3.1 Definition of Regular Expression

A language of alphabet $\Sigma = \{a_1, a_2, \dots, a_n\}$ is *regular* if it can be expressed by applying a finite number of times the operations of concatenation, union, and star, starting with the unitary languages¹⁰ $\{a_1\}, \{a_2\}, \dots, \{a_n\}$ or the empty language \emptyset .

More precisely a *regular expression* (r.e.) is a string r containing the terminal characters of alphabet Σ and the metasymbols:¹¹

$$\cdot \text{ concatenation} \quad \cup \text{ union} \quad * \text{ star} \quad \emptyset \text{ empty set} \quad ()$$

in accordance with the following rules:

1. $r = \emptyset$
2. $r = a, a \in \Sigma$

⁸A digital component incorporating a memory.

⁹The language family can also be defined by the form of the logical predicates characterizing language sentences, as e.g., in [13].

¹⁰A unitary language contains one sentence.

¹¹In order to prevent confusion between terminals and metasymbols, the latter should not be in the alphabet. If not, metasymbols must be suitably recoded to make them distinguishable.

Table 2.1 Language denoted by a regular expression

Expression r	Language L_r
1. ε	$\{\varepsilon\}$
2. $a \in \Sigma$	$\{a\}$
3. $s \cup t$ or also $s \mid t$	$L_s \cup L_t$
4. $s.t$ or also st	L_s, L_t
5. s^*	L_s^*

3. $r = (s \cup t)$
 4. $r = (s.t)$ or $r = (st)$
 5. $r = (s)^*$

where s and t are r.e.

Parentheses may often be dropped by imposing the following precedence when applying operators: first star, then concatenation, and last union.

For improving expressivity, the symbols ε (empty string) and cross may be used in an r.e., since they are derived from the three basic operations by the identities $\varepsilon = \emptyset^*$ and $s^+ = s(s)^*$.

It is customary to write the union cup ‘ \cup ’ symbol as a vertical slash ‘ $|$ ’, called *alternative*.

Rules (1) to (5) compose the syntax of r.e., to be formalized later by means of a grammar (Example 2.31, p. 30).

The meaning or denotation of an r.e. r is a language L_r over alphabet Σ , defined by the correspondence in Table 2.1.

Example 2.15 Let $\Sigma = \{1\}$, where 1 may be viewed as a pulse or signal. The language denoted by expression:

$$e = (111)^*$$

contains the sequences multiple of three:

$$L_e = \{1^n \mid n \bmod 3 = 0\}$$

Notice that dropping the parentheses the language changes, due to the precedence of star over concatenation:

$$e_1 = 111^* = 11(1)^*, \quad L_{e_1} = \{1^n \mid n \geq 2\}$$

Example 2.16 (Integers) Let $\Sigma = \{+, -, d\}$ where d denotes any decimal digit 0, 1, ..., 9. The expression

$$e = (+ \cup - \cup \varepsilon)dd^* \equiv (+ \mid - \mid \varepsilon)dd^*$$

produces the language

$$L_e = \{+, -, \varepsilon\}\{d\}\{d\}^*$$

of integers with or without a sign, such as +353, -5, 969, +001.

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

A language is *regular* if it is denoted by a regular expression. The collection of all regular languages is called the *family REG* of *regular* languages.

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

Comparing the *REG* and *FIN* families, it is easy to see that every finite language is regular, $FIN \subseteq REG$. In fact, a finite language is the union of finitely many strings x_1, x_2, \dots, x_k , each one being the concatenation of finitely many characters, $x_i = a_1 a_2 \dots a_{n_i}$. The structure of the r.e. producing a finite language is then a union of k terms, made by concatenation of n_i characters. But *REG* includes nonfinite languages too, thus proving strict inclusion of the families, $FIN \subset REG$.

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

2.3.2 Derivation and Language

We formalize the mechanism by which an r.e. produces a string of the language. Supposing for now the given r.e. e is fully parenthesized (except for atomic terms), we introduce the notion of *subexpression* (s.e.) in the next example:

$$e_0 = \left(\overbrace{((a \cup (bb))^*)}^{e_1} \overbrace{\left((c^+) \cup \underbrace{(a \cup (bb))}_s \right)}^{e_2} \right)$$

This r.e. is structured as concatenation of two parts e_1 and e_2 , to be called subexpressions. In general an s.e. f of an r.e. e is a well-parenthesized substring immediately occurring inside the outermost parentheses. This means no other well-parenthesized substring of e contains f . In the example, the substring labeled s is not s.e. of e_0 but is s.e. of e_2 .

When the r.e. is not fully parenthesized, in order to identify the subexpressions one has to insert (or to imagine) the missing parentheses, in agreement with operator precedence.

Notice that three or more terms, combined by union, need not to be pairwise parenthesized, because the operation is associative, as in

$$(c^+ \cup a \cup (bb))$$

The same applies to three or more concatenated terms.

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

1. $e_k, 1 \leq k \leq m$, is a choice of the union $(e_1 \cup \dots \cup e_k \cup \dots \cup e_m)$
2. $e^m = \underbrace{e \dots e}_m, m \geq 1$, is a choice of the expressions e^* , e^+
 m times
3. the empty string is a choice of e^*

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

Definition 2.17 (Derivation¹²) We say e' derives e'' , written $e' \Rightarrow e''$, if:

$$e'' \text{ is a choice of } e'$$

or

$$e' = e_1 \dots e_k \dots e_m \quad \text{and} \quad e'' = e_1 \dots e''_k \dots e_m$$

where e''_k is a choice of $e_k, 1 \leq k \leq m$

A derivation can be applied two or more times in a row. We say e_0 derives e_n in n steps, written:

$$e_0 \xrightarrow{n} e_n$$

if:

$$e_0 \Rightarrow e_1, \quad e_1 \Rightarrow e_2, \quad \dots, \quad e_{n-1} \Rightarrow e_n$$

The notation:

$$e_0 \xrightarrow{+} e_n$$

states that e_0 derives e_n in some $n \geq 1$ steps. The case $n = 0$ corresponds to the identity $e_0 = e_n$ and says the derivation relation is reflective. We also write

$$e_0 \xrightarrow{*} e_n \quad \text{for } (e_0 \xrightarrow{+} e_n) \vee (e_0 = e_n)$$

Example 2.18 (Derivations) Immediate derivations:

$$a^* \cup b^+ \Rightarrow a^*, \quad a^* \cup b^+ \Rightarrow b^+, \quad (a^* \cup bb)^* \Rightarrow (a^* \cup bb)(a^* \cup bb)$$

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 $e' = (a^* \cup bb)^*$, to choose $(a^2 \cup bb)^*$, because a^* is not an s.e. of e' . Although 2 is a correct choice for the star, such premature choice would rule out the derivation of a valid sentence such as $a^2 bba^3$.

Multi-step derivations:

$$\begin{aligned} a^* \cup b^+ &\Rightarrow a^* \Rightarrow \varepsilon \text{ that is } a^* \cup b^+ \xrightarrow{2} \varepsilon \text{ or also } a^* \cup b^+ \xrightarrow{+} \varepsilon \\ a^* \cup b^+ &\Rightarrow b^+ \Rightarrow bbb \text{ or also } (a^* \cup b^+) \xrightarrow{+} bbb \end{aligned}$$

¹²Also called *implication*.

Some expressions produced by derivation from an expression r contain the meta-symbols union, star, and cross; some others just terminal characters or the empty string (and maybe some 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

$$L_r = \{x \in \Sigma^* \mid r \xrightarrow{*} x\}$$

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

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

Example 2.19 Consider the derivations:

1. $a^*(b \cup c \cup d)f^+ \Rightarrow aaa(b \cup c \cup d)f^+ \Rightarrow aaacf^+ \Rightarrow aaacf$
2. $a^*(b \cup c \cup d)f^+ \Rightarrow a^*cf^+ \Rightarrow aaacf^+ \Rightarrow aaacf$

Compare derivations (1) and (2). In (1) the first choice takes the leftmost s.e. (a^*), whereas in (2) another s.e. ($b \cup c \cup d$) is taken. Since the two steps are independent of each other, they can be applied in any order. By a further step, we obtain r.e. $aaacf^+$, and the last step produces sentence $aaacf$. The last step, being independent from the others, could be performed before, after, or in between.

The example has shown that many different but equivalent orders of choice may derive the same sentence.

2.3.2.1 Ambiguity of Regular Expressions

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

Example 2.20 (Ambiguous regular expression) The language of alphabet $\{a, b\}$, characterized by the presence of at least one a , is defined by

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

where the compulsory presence of a is evident. Now sentences containing two or more occurrences of a can be obtained by multiple derivations, which differ with respect to the character identified with the compulsory one of the r.e. For instance, sentence aa offers two possibilities:

$$\begin{aligned} (a \cup b)^*a(a \cup b)^* &\Rightarrow (a \cup b)a(a \cup b)^* \Rightarrow aa(a \cup b)^* \Rightarrow aa\varepsilon = aa \\ (a \cup b)^*a(a \cup b)^* &\Rightarrow \varepsilon a(a \cup b)^* \Rightarrow \varepsilon a(a \cup b) \Rightarrow \varepsilon aa = aa \end{aligned}$$

A sentence (and the r.e. deriving it) is said to be *ambiguous*, if and only if can be obtained through two structurally different derivations. Thus, sentence aa is ambiguous, while sentence ba is not ambiguous, because there exists only one set of choices, corresponding to derivation:

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

We next provide a simple sufficient condition for a sentence (and its r.e.) to be ambiguous. To this end, we number the letters of a r.e. f , obtaining a *numbered r.e.*:

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

which defines a regular language of alphabet $\{a_1, b_2, a_3, a_4, b_5\}$.

An r.e. f is *ambiguous* if the language defined by the corresponding numbered r.e. f' contains distinct strings x, y such that they become identical when the numbers are erased.¹³ For instance, strings a_1a_3 and a_3a_4 of language f' prove the ambiguity of aa .

Ambiguous definitions are a source of trouble in many settings. They should be avoided in general, although they may have the advantage of concision over unambiguous definitions. The concept of ambiguity will be thoroughly studied for grammars.

2.3.3 Other Operators

When regular expressions are used in practice, it may be convenient to add to the *basic operators* (union, concatenation, star) the derived operators power and cross.

For further expressivity other derived operators may be practical:

Repetition from k to $n > k$ times: $[a]_k^n = a^k \cup a^{k+1} \cup \dots \cup a^n$

Option: $[a] = (\varepsilon \cup a)$

Interval of ordered set: to represent any digit in the ordered set $0, 1, \dots, 9$ the short notation is $(0 \dots 9)$. Similarly the notation $(a \dots z)$ and $(A \dots Z)$ stand for the set of lower (respectively, upper) case letters.

Sometimes, other set operations are also used: intersection, set difference, and complement. Expressions using such operators are called *extended r.e.*, although the name is not standard, and one has to specify the allowed operators.

Example 2.21 (Extended r.e. with intersection) This operator provides a straightforward formulation of the fact that valid strings must simultaneously obey two conditions. To illustrate, let $\{a, b\}$ be the alphabet and assume a valid string must (1) contain substring bb and (2) have even length. The former condition is imposed by the r.e.:

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

the latter by the r.e.:

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

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

$$((a \mid b)^* bb(a \mid b)^*) \cap ((a \mid b)^2)^*$$

¹³Notice that the above sufficient condition does not cover cases where an empty (sub)string is generated through distinct derivations, as happens, for instance, with the r.e. $(a^* \mid b^*)c$.

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

$$((a \mid b)^2)^* bb ((a \mid b)^2)^* \mid (a \mid b) ((a \mid b)^2)^* bb (a \mid b) ((a \mid b)^2)^*$$

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) Consider the set L of strings of alphabet $\{a, b\}$ not containing aa as substring. The complement of the language is

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

easily defined by the r.e.: $(a \mid b)^* aa (a \mid b)^*$, whence the extended r.e.

$$L = \neg((a \mid b)^* aa (a \mid b)^*)$$

The definition by a basic r.e.:

$$(ab|b)^*(a \mid \varepsilon)$$

is, subjectively, less readable.

Actually it is not by coincidence that both preceding examples admit also an r.e. without intersection or complement. A theoretical result to be presented in Chap. 3 states that an r.e. extended with complement and intersection produces always a regular language, which by definition can be defined by a non-extended r.e. as well.

2.3.4 Closure Properties of REG Family

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

Property 2.23 The family REG of regular languages is closed by the operators concatenation, union, and star (therefore also by derived operators such as cross).

The property descends from the very definition of r.e. and of REG (p. 19). In spite of its theoretical connotation, the property has practical relevance: two regular languages can be combined using the above operations, at no risk of losing the nice features of the class of regular languages. This will have an important practical consequence, to permit compositional design of algorithms used to check if an input string is valid for a language. Furthermore we anticipate the REG family is closed by intersection, complement, and reflection too, which will be proved later.

The next statement provides an alternative definition of family REG .

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

The proof is simple. Suppose by contradiction a family exists $F \subset REG$, which is closed by the same operators and contains all finite languages. Consider any language L_e defined by an r.e. e ; the language is obtained by repeated applications of the operators present in e , starting with some finite languages consisting of single characters. It follows from the hypothesis that language $L(e)$ belongs also to family F , which then contains any regular language, contradicting the strict inclusion $F \subset REG$.

We anticipate other families exist which are closed by the same operators of Property 2.23. Chief among them is the family CF of context-free languages, to be introduced soon. From Statement 2.24 follows a containment relation between the two families, $REG \subset CF$.

2.4 Linguistic Abstraction

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:¹⁴

consider a concept without thinking of a specific example.

Abstracting away from the actual characters representing a language construct we perform a linguistic abstraction. This is a language transformation that replaces the terminal characters of the concrete language with other ones taken from an abstract alphabet. Abstract characters should be simpler and suitable to represent similar constructs from different artificial languages.¹⁵

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, substitution (later defined). Starting from the abstract language, a concrete or real language is obtained by the reverse transformation, metaphorically 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 compiler functions are designed to process abstract, instead of concrete,

¹⁴From WordNet 2.1.

¹⁵The idea of language abstraction is inspired by research in linguistics aiming at discovering the underlying similarities of human languages, disregarding morphological and syntactic differences.

language constructs. Thus parts of, say, a C compiler can be reused for similar languages like FORTRAN or Pascal.

The surprisingly few abstract paradigms in use, will be presented in this chapter, starting from the ones conveniently specified by regular expressions, the lists.

2.4.1 Abstract and Concrete Lists

An abstract *list* contains an unbounded number of elements e of the same type. It is defined by the r.e. e^+ or e^* , if elements can be missing.

An element for the moment should be viewed as a terminal character; but in later refinements, the element may become a string from another formal language: think e.g., of a list of numbers.

Lists with Separators and Opening/Closing Marks In many concrete cases, adjacent elements must be separated by strings called *separators*, s in abstract syntax. Thus in a list of numbers, a separator should delimit the end of a number and the beginning of the next one.

A *list with separators* is defined by the r.e. $e(se)^*$, saying the first element can be followed by zero or more pairs se . The equivalent definition $(es)^*e$ 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 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

$$ie(se)^*f$$

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

Instruction block:

begin instr₁; instr₂; ... instr_n end

where *instr* possibly stands for assignment, go to, if-statement, write-statement, etc. Corresponding abstract and concrete terms are

Abstract alphabet	Concrete alphabet
i	<i>begin</i>
e	<i>instr</i>
s	$;$
f	<i>end</i>

Procedure parameters: as in

$$\underbrace{\text{procedure } \text{WRITE}}_i(\underbrace{\text{par}_1}_e, \underbrace{\text{par}_2, \dots, \text{par}_n}_s) \underbrace{)}_f$$

Should an empty parameter list be legal, as e.g., *procedure WRITE()*, the r.e. becomes $i[e(se)^*]f$.

Array definition:

$$\underbrace{\text{array } \text{MATRIX}'}_i[\underbrace{\text{int}_1}_e, \underbrace{\text{int}_2, \dots, \text{int}_n}_s] \underbrace{'}_f$$

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

2.4.1.1 Substitution

The above examples illustrate the mapping from concrete to abstract symbols. 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 termed the *source*, with a sentence of another language called the *target*. As always Σ is the source alphabet and $L \subseteq \Sigma^*$ the source language. Consider a sentence of L containing one or more occurrences of a source character b :

$$x = a_1 a_2 \dots a_n \quad \text{where for some } i, a_i = b$$

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, that is, a language of alphabet $(\Sigma \setminus \{b\}) \cup \Delta$, defined as

$$\{y \mid y = y_1 y_2 \dots y_n \wedge (\text{if } a_i \neq b \text{ then } y_i = a_i \text{ else } y_i \in L_b)\}$$

Notice all characters other than b do not change. By the usual approach the substitution can be defined on the whole source language, by applying the operation to every source sentence.

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

$$ie(se)^*f$$

and the substitutions to be applied are tabulated below:

Abstract char.	Imagine
i	$L_i = \text{procedure } \langle \text{procedure identifier} \rangle()$
e	$L_e = \langle \text{parameter identifier} \rangle$
s	$L_s = ,$
f	$L_f =)$

For instance, the opening mark i is replaced with a string of language L_i , where the procedure identifier has to agree with the rules of the technical language.

Clearly the target languages of the substitution depend on the syntax sugar of the concrete language intended for.

Notice the four substitutions are independent and can be applied in any order.

Example 2.27 (Identifiers with underscore) In certain programming languages, long mnemonic identifier names can be constructed by appending alphanumeric strings separated by a low dash: thus *LOOP3_OF_35* is a legal identifier. More precisely the first string must initiate with a letter, the others may contain letters and digits, and adjacent dashes are forbidden, as well as a trailing dash.

At first glance the language is a nonempty list of strings s , separated by a dash:

$$s(_s)^*$$

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

$$i(_s)^*$$

Substituting to i the language $(A \dots Z)(A \dots Z \mid 0 \dots 9)^*$, and to s the language $(A \dots Z \mid 0 \dots 9)^+$, 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.

Hierarchical or Precedence Lists A 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. The present abstract structure, called a hierarchical list, is restricted to lists with a bounded number of levels. The case when levels are unbounded is studied later using grammars, under the name of nested structures.

A hierarchical list is also called a *list with precedences*, because a list at level k bounds its elements more strongly than the list at level $k - 1$; in other words the elements at higher level must be assembled into a list, and each becomes an element at next lower level.

Each level may have opening/closing marks and a separator; such delimiters are usually distinct level by level, in order to avoid confusion.

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

$$list_1 = i_1 list_2 (s_1 list_2)^* f_1$$

$$list_2 = i_2 list_3 (s_2 list_3)^* f_2$$

...

$$list_k = i_k e_k (s_k e_k)^* f_k$$

Notice the last level alone may contain atomic elements. But a common variant permits at any level k atomic elements e_k to occur side by side with lists of level $k + 1$. Some concrete examples follow.

Example 2.28 (Two hierarchical lists)

Block of print instructions:

begin instr₁; instr₂; ... instr_n end

where $instr$ is a print instruction, $WRITE(var_1, var_2, \dots, var_n)$, i.e., a list (from Example 2.25). There are two levels:

- Level 1: list of instructions $instr$, opened by *begin*, separated by semicolon and closed by *end*.
- Level 2: list of variables var separated by comma, with $i_2 = WRITE$ (and $f_2 =$).

Arithmetic expression not using parentheses: the precedence levels of operators determine how many levels there are in the list. For instance, the operators \times , \div and $+$, $-$ are layered on two levels and the string

$$3 + \underbrace{5 \times 7 \times 4 - 8 \times 2 \div 5}_{\text{term}_1} + 3 \underbrace{- 8}_{\text{term}_2}$$

is a two-level list, with neither opening nor closing mark. At level one we find a list of terms ($e_1 = \text{term}$) separated by the signs $+$ and $-$, i.e., by lower precedence operators. At level two we see a list of numbers, separated by higher precedence signs \times , \div . One may go further and introduce a third level having the exponentiation sign “**” as separator.

Hierarchical structures are of course 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 has a distinct separator, possibly in order to warn the listener of an utterance that the list is approaching the end. Furthermore, 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 a list of paragraphs, and so on.

2.5 Context-Free Generative Grammars

We start the study of the context-free language family, 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 sci-

ence 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 lastly positions this formal model in the classical hierarchy of grammars and computational models due to Chomsky.

2.5.1 Limits of Regular Languages

Regular expressions, though quite practical for describing list and related paradigms, falls short of the capacity needed to define other frequently occurring constructs. A case are the block structures (or nested parentheses) offered by many technical languages, schematized by

$$\begin{array}{c} \text{begin } \text{begin } \text{begin} \dots \text{end } \text{begin} \dots \text{end} \dots \text{end } \text{end} \\ \underbrace{\hspace{10em}}_{\text{begin}} \quad \underbrace{\hspace{2.5em}}_{\text{begin}} \quad \underbrace{\hspace{2.5em}}_{\text{end}} \end{array}$$

Example 2.29 (Simple block structure) In brief, let $\{b, e\}$ be the alphabet, and consider a somewhat limited case of nested structures, such that all opening marks precede all closing marks. Clearly, opening/closing marks must have identical count:

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

We argue that this language cannot be defined by a regular expression, deferring the formal proof to a later section. In fact, since strings must have all b 's left of any e , either we write an overly general r.e. such as b^+e^+ , which accepts illegal strings like b^3e^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 $(be)^+$, illegal strings like *bebe* creep in.

For defining this and other 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*¹⁶ is a set of simple rules that can be repeatedly applied in order to generate all and only the valid strings.

¹⁶Sometimes the term *grammar* has a broader connotation than *syntax*, as when rules for computing the meaning of sentences are added to rules for enumerating them. When necessary, the intended meaning of the term will be made clear.

Example 2.30 (Palindromes) The language to be defined:

$$L = \{uu^R \mid u \in \{a, b\}^*\} = \{\varepsilon, aa, bb, abba, baab, \dots, abbbba, \dots\}$$

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

$$pal \rightarrow \varepsilon \quad pal \rightarrow a \ pal \ a \quad pal \rightarrow b \ pal \ b$$

The arrow ‘ \rightarrow ’ is a metasymbol, exclusively used to separate the left from the right part of a rule.

To derive the strings, just replace symbol ‘pal’, termed *nonterminal*, with the right part of a rule, for instance:

$$pal \Rightarrow a \ pal \ a \Rightarrow ab \ pal \ ba \Rightarrow abb \ pal \ bba \Rightarrow \dots$$

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:

$$abb \ pal \ bba \Rightarrow abbebba = abbbba$$

(Incidentally the language of palindromes is not regular.)

Next we enrich the example into a list of palindromes separated by commas, exemplified by sentence *abba, bbaabb, aa*. The grammar adds two list-generating rules to the previous ones:

$$\begin{array}{ll} list \rightarrow pal, list & pal \rightarrow \varepsilon \\ list \rightarrow pal & pal \rightarrow a \ pal \ a \\ & pal \rightarrow b \ pal \ b \end{array}$$

The first rule says: the concatenation of palindrome, comma, and list produces a (longer) list. The second says 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 designated 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 defining a language over a fixed terminal alphabet, say $\Sigma = \{a, b\}$, is a formula, that is, a string over the alphabet $\Sigma_{r.e.} = \{a, b, \cup, ^*, \emptyset, (,)\}$; such strings can be viewed in turn as sentences of a language.

Following the definition of r.e. on p. 17, this language is generated by the following syntax $G_{r.e.}$:

1. $expr \rightarrow \emptyset$
2. $expr \rightarrow a$
3. $expr \rightarrow b$
4. $expr \rightarrow (expr \cup expr)$
5. $expr \rightarrow (expr \ expr)$
6. $expr \rightarrow (expr)^*$

where numbering is only for reference. A derivation is

$$\begin{aligned} \text{expr} &\Rightarrow_4 (\text{expr} \cup \text{expr}) \Rightarrow_5 ((\text{expr expr}) \cup \text{expr}) \Rightarrow_2 ((a \text{expr}) \cup \text{expr}) \Rightarrow_6 \\ &\Rightarrow ((a(\text{expr})^*) \cup \text{expr}) \Rightarrow_4 ((a((\text{expr} \cup \text{expr}))^*) \cup \text{expr}) \Rightarrow_2 \\ &\Rightarrow ((a((a \cup \text{expr}))^*) \cup \text{expr}) \Rightarrow_3 ((a((a \cup b))^*) \cup \text{expr}) \Rightarrow_3 \\ &\Rightarrow ((a((a \cup b))^*) \cup b) = e \end{aligned}$$

Since the generated string can be interpreted as an r.e., it defines a second language of alphabet Σ :

$$L_e = \{a, b, aa, ab, aaa, aba, \dots\}$$

the set of strings starting with letter a , plus string b .

A word of caution: this example displays two levels of languages, since the syntax defines certain strings to be understood as definitions of other languages. To avoid terminological confusion, we say the syntax stays at the *metalinguistic* level, that is, over the linguistic level; or that the syntax is a *metagrammar*.

To set the two levels apart, it helps to consider the alphabets: at meta-level the alphabet is $\Sigma_{r.e.} = \{a, b, \cup, ^*, \emptyset, (,)\}$, 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 in, say, 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 illustration of language versus metalanguage is provided by XML, the metanotation used to define a variety of Web document types.

Definition 2.32 A *context-free (CF)* (or type 2 or *BNF*¹⁷) grammar G is defined by four entities:

1. V , *nonterminal alphabet*, a set of symbols termed *nonterminals* (or metasymbols).
2. Σ , *terminal alphabet*.
3. P , a set of syntactic *rules* (or *productions*).
4. $S \in V$, a particular nonterminal termed *axiom*.

A rule of P is an ordered pair $X \rightarrow \alpha$, with $X \in V$ and $\alpha \in (V \cup \Sigma)^*$.

Two or more rules:

$$X \rightarrow \alpha_1, \quad X \rightarrow \alpha_2, \quad \dots, \quad X \rightarrow \alpha_n$$

with the same left part X can be concisely grouped in

$$X \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n \quad \text{or} \quad X \rightarrow \alpha_1 \cup \alpha_2 \cup \dots \cup \alpha_n$$

We say $\alpha_1, \alpha_2, \dots, \alpha_n$ are the *alternatives* of X .

¹⁷Type 2 comes from Chomsky's classification. The 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 programming language definition.

Table 2.2 Different styles for writing grammars

Nonterminals	Terminals	Examples
Words between angle brackets, for instance: $< \text{sentence} >$, $< \text{list of sentences} >$	Written as they are, without special marks	$< \text{if sentence} > \rightarrow$ $\text{if } < \text{cond} > \text{ then } < \text{sentence} >$ $\text{else } < \text{sentence} >$
Words written as they are, without special marks; may not contain blank spaces, for instance: sentence , list_of_sentences	Written in black, in italic or quoted, for instance: a then 'a' 'then'	$\text{if_sentence} \rightarrow$ if cond then sentence else sentence or $\text{if_sentence} \rightarrow$ 'if' 'cond' 'then' 'sentence' 'else' 'sentence'
Uppercase Latin letters; terminal and nonterminal alphabets disjoint	Written as they are, without special marks	$F \rightarrow \text{if } C \text{ then } D \text{ else } D$

2.5.3 Conventional Grammar Representations

To prevent confusion, the metasymbols ‘ \rightarrow ’, ‘ $|$ ’, ‘ \cup ’, ‘ ε ’ should not be used for terminal and nonterminal symbols; moreover, the terminal and nonterminal alphabets should be disjoint. In professional and scientific practice a few different styles are used to represent terminals and nonterminals, as specified in Table 2.2.

The grammar of Example 2.30 in the first style becomes:

$$\begin{aligned} &< \text{sentence} > \rightarrow \varepsilon \\ &< \text{sentence} > \rightarrow a < \text{sentence} > a \\ &< \text{sentence} > \rightarrow b < \text{sentence} > b \end{aligned}$$

Alternative rules may be grouped together:

$$< \text{sentence} > \rightarrow \varepsilon | a < \text{sentence} > a | b < \text{sentence} > b$$

If a technical grammar is large, of the order of some hundred rules, it should be written with care in order to facilitate searching for specific definitions, making changes, and cross referencing. Nonterminals should be identified by self-explanatory names and rules should be divided into sections and numbered for reference.

On the other hand, in very simple examples, the third style is more suitable, i.e., to have disjoint short symbols for terminals and nonterminals.

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

- lowercase Latin letters near the beginning of the alphabet { a, b, \dots } for terminal characters;
- uppercase Latin letters { A, B, \dots, Z } for nonterminal symbols;
- lowercase Latin letters near the end of the alphabet { r, s, \dots, z } for strings over Σ^* (i.e. including only terminals);
- lowercase Greek letters { α, β, \dots } for strings over the combined alphabets $(V \cup \Sigma)^*$.

Table 2.3 Classification of grammar rules

Class and description	Examples
<i>Terminal</i> : RP contains terminals or the empty string	$\rightarrow u \mid \epsilon$
<i>Empty (or null)</i> : RP is empty	$\rightarrow \epsilon$
<i>Initial</i> : LP is the axiom	$S \rightarrow$
<i>Recursive</i> : LP occurs in RP	$A \rightarrow \alpha A \beta$
<i>Left-recursive</i> : LP is prefix of RP	$A \rightarrow A \beta$
<i>Right-recursive</i> : LP is suffix of RP	$A \rightarrow \beta A$
<i>Left and right-recursive</i> : conjunction of two previous cases	$A \rightarrow A \beta A$
<i>Copy or categorization</i> : RP is a single nonterminal	$A \rightarrow B$
<i>Linear</i> : at most one nonterminal in RP	$\rightarrow u B v \mid w$
<i>Right-linear</i> (type 3): as linear but nonterminal is suffix	$\rightarrow u B \mid w$
<i>Left-linear</i> (type 3): as linear but nonterminal is prefix	$\rightarrow B v \mid w$
<i>Homogeneous normal</i> : n nonterminals or just one terminal	$\rightarrow A_1 \dots A_n \mid a$
<i>Chomsky normal</i> (or homogeneous of degree 2): two nonterminals or just one terminal	$\rightarrow BC \mid a$
<i>Greibach normal</i> : one terminal possibly followed by nonterminals	$\rightarrow a\sigma \mid b$
<i>Operator normal</i> : two nonterminals separated by a terminal (operator); more generally, strings devoid of adjacent nonterminals	$\rightarrow AaB$

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

The classification is based on the form of the right part *RP* of a rule, excepting the recursive classes that also consider the left part *LP*. We omit any part of a rule that is irrelevant for the classification.

Left- and right-linear forms are also known as *type 3* grammars from Chomsky classification. Most rule types will occur in the book; the remaining ones are listed for general reference.

We shall see that some of the grammar forms can be forced on any given grammar, leaving 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, where δ and η are any, possibly empty strings. Let $A \rightarrow \alpha$ be a rule of G and let $\gamma = \delta \alpha \eta$ be the string obtained replacing in β nonterminal A with the right part α .

The relation between such two strings is called *derivation*. We say that β *derives* γ for grammar G , written:

$$\beta \xrightarrow{G} \gamma$$

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

Consider now a chain of derivations of length $n \geq 0$:

$$\beta_0 \Rightarrow \beta_1 \Rightarrow \dots \Rightarrow \beta_n$$

shortened to

$$\beta_0 \xrightarrow{n} \beta_n$$

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

To express derivations of any length we write:

$$\beta_0 \xrightarrow{*} \beta_n \quad (\text{resp. } \beta_0 \xrightarrow{+} \beta_n)$$

if the length of the chain is $n \geq 0$ (resp. $n \geq 1$).

The *language generated* or defined by a grammar G starting from nonterminal A is

$$L_A(G) = \{x \in \Sigma^* \mid A \xrightarrow{+} x\}$$

It is the set of terminal strings deriving in one or more steps from A .

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

$$L(G) = L_S(G) = \{x \in \Sigma^* \mid S \xrightarrow{+} x\}$$

In some cases we need to consider derivations producing strings still containing nonterminals. A *string form* generated by G starting from nonterminal $A \in V$ is a string $\alpha \in (V \cup \Sigma)^*$ such that $A \xrightarrow{*} \alpha$. In particular, if A is the axiom, the string is termed *sentential form*. Clearly a sentence is a sentential form devoid of nonterminals.

Example 2.33 (Book structure) The grammar defines the structure of a book, containing a front page (f) and a nonempty series (derived from nonterminal A) of chapters; each one starts with a title (t) and contains a nonempty series (derived from B) of lines (l). Grammar G_l :

$$\begin{aligned} S &\rightarrow fA \\ A &\rightarrow AtB \mid tB \\ B &\rightarrow lB \quad | \quad l \end{aligned}$$

Some derivations are listed. From A the string form $tBtB$ and the string $tlltl \in L_A(G_l)$; from the axiom S sentential forms $fAtlB$, $ftBtB$ and sentence $flltl$.

The language generated from B is $L_B(G_l) = l^+$; the language $L(G_l)$ generated by G_l is defined by the r.e. $f(tl^+)^+$, showing the language is in the *REG* family. In fact, this language is a case of an abstract hierarchical list.

A language is *context-free* if a context-free grammar exists that generates it. The family of context-free languages 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 The next grammar G_{l2} is clearly equivalent to G_l of Example 2.33:

$$\begin{aligned} S &\rightarrow fX \\ X &\rightarrow XtY \mid tY \\ Y &\rightarrow lY \quad | \quad l \end{aligned}$$

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

$$\begin{aligned} S &\rightarrow fA \\ A &\rightarrow AtB \mid tB \\ B &\rightarrow Bl \quad | \quad l \end{aligned}$$

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

$$B \xrightarrow[G_l]{n} l^n \quad \text{and} \quad B \xrightarrow[G_{l3}]{n} l^n$$

generate the same language $L_B = l^+$.

2.5.5 Erroneous Grammars and Useless Rules

When writing a grammar attention should be paid that all nonterminals are defined and that each one effectively contributes to the production of some sentence. In fact, 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, i.e., there exists derivation $S \xrightarrow{*} \alpha A \beta$.
2. Every nonterminal A is *well-defined*, i.e., it generates a nonempty language, $L_A(G) \neq \emptyset$.

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

Grammar Cleaning The algorithm operates in two phases, first pinpointing the undefined nonterminals, then the unreachable ones. Lastly the rules containing nonterminals of either type can be canceled.

Phase 1. Compute the set $DEF \subseteq V$ of well-defined nonterminals.

The set DEF is initialized with the nonterminals of terminal rules, those having a terminal string as right part:

$$DEF := \{A \mid (A \rightarrow u) \in P, \text{ with } u \in \Sigma^*\}$$

Then the next transformation is applied until convergence is reached:

$$DEF := DEF \cup \{B \mid (B \rightarrow D_1 D_2 \dots D_n) \in P\}$$

where every D_i is a terminal or a nonterminal symbol present in DEF .

At each iteration two outcomes are possible:

- a new nonterminal is found having as right part a string of symbols that are well-defined nonterminals or terminals, or else
- the termination condition is reached (no new nonterminal is found).

The nonterminals belonging to the complement set $V \setminus DEF$ are undefined and should be eliminated.

Phase 2. A nonterminal is reachable from the axiom, if, and only if, there exists a path in the following graph, which represents a relation between nonterminals, called *produce*:

$$A \xrightarrow{\text{produce}} B$$

saying that A produces B if, and only if, there exists a rule $A \rightarrow \alpha B \beta$, where A, B are nonterminals and α, β are any strings.

Clearly C is reachable from S if, and only if, in this graph there exists an oriented path from S to C . The unreachable nonterminals are the complement with respect to V . They should be eliminated because they do not contribute to the generation of any sentence.

Quite often the following requirement is added to the above cleanliness conditions.

- G should not permit *circular derivations* $A \stackrel{+}{\Rightarrow} A$.

The reason is such derivations are inessential because, if string x is obtained by means of a circular derivation $A \Rightarrow A \Rightarrow x$, it can also be obtained by the shorter derivation $A \Rightarrow x$.

Moreover, circular derivations cause ambiguity (a negative phenomenon later discussed).

In this book we assume grammars are always clean and non-circular.

Example 2.35 (Unclean examples)

- The grammar with rules $\{S \rightarrow aASb, A \rightarrow b\}$ generates nothing.
- The grammar G with rules $\{S \rightarrow a, A \rightarrow b\}$ has an unreachable nonterminal A ; the same language $L(G)$ is generated by the clean grammar $\{S \rightarrow a\}$.
- Circular derivation: The grammar with rules $\{S \rightarrow aASb \mid A, A \rightarrow S \mid c\}$ presents the circular derivation $S \Rightarrow A \Rightarrow S$. The grammar $\{S \rightarrow aSSb \mid c\}$ is equivalent.

- Notice that circularity may also come from the presence of an empty rule, as for instance in the following grammar fragment:

$$X \rightarrow XY \mid \dots \quad Y \rightarrow \varepsilon \mid \dots$$

Finally we observe that a grammar, although clean, may still contain redundant rules, as the next one:

Example 2.36 (Double rules)

1. $S \rightarrow aASb$
2. $S \rightarrow aBSb$
3. $S \rightarrow \varepsilon$
4. $A \rightarrow c$
5. $B \rightarrow c$

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

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 form of grammar rules. In order to generate an unbounded number of strings, the grammar must be able to derive strings of unbounded length. To this end, recursive rules are necessary, as next argued.

An $n \geq 1$ steps derivation $A \xrightarrow{n} xAy$ is called *recursive* (*immediately recursive* if $n = 1$); similarly nonterminal A is called recursive. If x (resp. y) is empty, the recursion is termed *left* (resp. *right*).

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

Proof Clearly without recursive derivations, any derivation has bounded length, therefore $L(G)$ would be finite.

Conversely, assume G offers a recursive derivation $A \xrightarrow{n} xAy$, with not both x and y empty by the non-circularity hypothesis. Then the derivation $A \xrightarrow{+} x^m A y^m$ exists, for every $m \geq 1$. Since G is clean, A can be reached from the axiom by a derivation $S \xrightarrow{*} uAv$, and also A derives at least one terminal string $A \xrightarrow{+} w$. Combining the derivations, we obtain

$$S \xrightarrow{*} uAv \xrightarrow{+} ux^m A y^m v \xrightarrow{+} ux^m w y^m v \quad (m \geq 1)$$

that generates an infinite language.

In order to see whether a grammar has recursions, we examine the binary relation *produce* of p. 36: a grammar does not have a recursion if, and only if, the graph of the relation has no circuit. \square

We illustrate by two grammars generating a finite and infinite language.

Example 2.38 (Finite language)

$$S \rightarrow aBc \quad B \rightarrow ab \mid Ca \quad C \rightarrow c$$

The grammar does not have a recursion and allows just two derivations, defining the finite language $\{aabbc, acac\}$.

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:

$$G = (\{E, T, F\}, \{i, +, *\}, (), P, E)$$

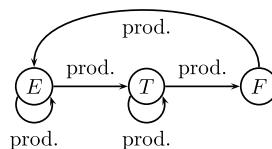
contains the rules:

$$E \rightarrow E + T \mid T \quad T \rightarrow T * F \mid F \quad F \rightarrow (E) \mid i$$

The language

$$L(G) = \{i, i + i + i, i * i, (i + i) * i, \dots\}$$

is the set of arithmetic expressions over the letter i , with signs of sum and product and parentheses. Nonterminal F (factor) is non-immediately recursive; T (term) and E (expression) are immediately recursive, both to the left. Such properties are evident from the circuits in the graph of the *produce* relation:



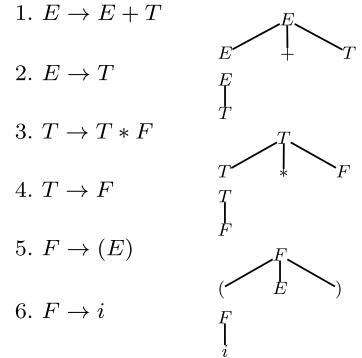
Since the grammar is clean and non-circular, the language is infinite.

2.5.7 Syntax Trees and Canonical Derivations

The process of derivation can be visualized as a syntax tree for better legibility. A *tree* is an oriented and ordered graph not containing a circuit, such that every pair of nodes is connected by exactly one oriented path. An arc $\langle N_1 \rightarrow N_2 \rangle$ defines the $\langle \text{father}, \text{son} \rangle$ relation, customarily visualized from top to bottom as in genealogical trees. The *siblings* of a node are ordered from left to right. The *degree* of a node is the number of its siblings. A tree contains one node without father, termed *root*.

Consider an internal node N : the *subtree* with root N is the tree having N as root and containing all siblings of N , all of their siblings, etc., that is, all *descendants*

Fig. 2.1 Grammar rules and corresponding tree fragments



of N . Nodes without sibling are termed *leaves* or *terminal nodes*. The sequence of all leaves, read from left to right, is the *frontier* of the tree.

A *syntax tree* has as root the axiom and as frontier a sentence.

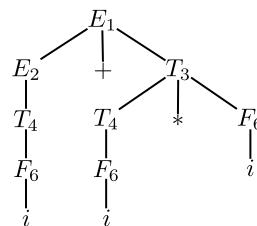
To construct the tree consider a derivation. For each rule $A_0 \rightarrow A_1 A_2 \dots A_r$, $r \geq 1$ used in the derivation, draw a small tree having A_0 as root and siblings $A_1 A_2 \dots A_r$, which may be terminals or nonterminals. If the rule is $A_0 \rightarrow \varepsilon$, draw one sibling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

Example 2.40 (Syntax tree) The grammar is reproduced in Fig. 2.1 numbering the rules for reference in the construction of the syntax tree.

The derivation:

$$\begin{aligned}
 E &\Rightarrow \underset{1}{E} + T \Rightarrow \underset{2}{T} + T \Rightarrow \underset{4}{F} + T \Rightarrow \underset{6}{i} + T \Rightarrow \underset{3}{i} + \underset{4}{T} * F \Rightarrow \\
 &\quad \underset{6}{i} + F * F \Rightarrow \underset{6}{i} + \underset{6}{i} * F \Rightarrow \underset{6}{i} + i * i
 \end{aligned} \tag{2.2}$$

corresponds to the following syntax tree:



where the labels of the rules applied are displayed. Notice that the same tree represents other equivalent derivations, like the next one:

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

and many others, which differ in the order rules are applied. Derivation (2.2) is termed *left* and derivation (2.3) is termed *right*.

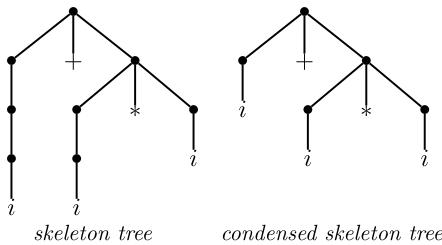
A syntax tree of a sentence x can also be encoded in a text, by enclosing each subtree between brackets.¹⁸ Brackets are subscripted with the nonterminal symbol. Thus the preceding tree is encoded by the *parenthesized expression*:

$$[[[[i]_F]_T]_E + [[i]_F]_T * [i]_F]_T]_E$$

or by

$$\begin{array}{c} \overbrace{i}^F + \overbrace{i}^F * \overbrace{i}^F \\ \overbrace{\overbrace{T}^F}^T \quad \overbrace{\overbrace{T}^F}^T \\ \overbrace{\overbrace{E}^T}^T \end{array}$$

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



or the corresponding parenthesized string:

$$[[[[i]] + [[i]] * [i]]]$$

A further simplification of the skeleton tree consists in shortening non-bifurcating paths, resulting in the *condensed skeleton tree* (right). The nodes fused together represent copy rules of the grammar. The corresponding parenthesized sentence is

$$[[i] + [[i]] * [i]]]$$

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.¹⁹

¹⁸ Assuming brackets not to be in the terminal alphabet.

¹⁹ A reference to the theory of tree languages is [5].

2.5.7.1 Left and Right Derivations

A derivation:

$$\beta_0 \Rightarrow \beta_1 \Rightarrow \dots \Rightarrow \beta_p$$

where:

$$\beta_i = \delta_i A_i \eta_i \quad \text{and} \quad \beta_{i+1} = \delta_i \alpha_i \eta_i$$

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

In words, a right (resp. left) derivation expands at each step the rightmost (resp. leftmost) nonterminal. A letter r or l may be subscripted to the arrow sign, to make explicit the order of the derivation.

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

Returning to the preceding example, the derivation (2.2) is leftmost and is denoted by $E \xrightarrow[l]{+} i + i * i$. The derivation (2.3) is rightmost, whereas the derivation

$$\begin{aligned} E &\xrightarrow[l,r]{} E + T \xrightarrow[r]{} E + T * F \xrightarrow[l]{} T + T * F \xrightarrow{} T + F * F \xrightarrow[r]{} \\ &T + F * i \xrightarrow[l]{} F + F * i \xrightarrow[r]{} F + i * i \xrightarrow[r]{} i + i * i \end{aligned} \tag{2.4}$$

is neither right nor left. The three derivations are represented by the same tree.

This example actually illustrates an essential property of context-free grammars.

Property 2.41 Every sentence of a context-free grammar can be generated by a left (or right) derivation.

Therefore it does no harm to use just right (or left) derivations in the definition (p. 34) of the language generated by a grammar.

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

2.5.8 Parenthesis Languages

Many artificial languages include parenthesized or nested structures, made by matching pairs of *opening/closing marks*. Any such occurrence may contain other matching pairs.

The marks are abstract elements that have different concrete representations in distinct settings. Thus Pascal block structures are enclosed within ‘begin’ ... ‘end’, while in language C curly brackets are 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 `<title>...</title>` used to delimit the document title. Similarly, in the LaTeX notation, a mathematical formula is enclosed between the marks `\begin{equation}...` `\end{equation}`.

When a marked construct may contain another construct of the same kind, it is called *self-nested*. Self-nesting is potentially unbounded in artificial languages, whereas in natural languages its use is moderate, because it causes difficulty of comprehension by breaking the flow of discourse. Next comes an example of a complex German sentence²⁰ with three nested relative clauses:

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

Abstracting from concrete representation and content, this paradigm is known as a *Dyck language*. The terminal alphabet contains one or more pairs of opening/closing marks. An example is alphabet $\Sigma = \{', ', '(', ')', '[,]\}$ and sentence `[](([]))`.

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

$$[] \Rightarrow \varepsilon \quad () \Rightarrow \varepsilon$$

thus obtaining another string. Repeat until the transformation no longer applies; the original string is correct if, and only if, the last string is empty.

Example 2.42 (Dyck language) To aid the eye, we encode left parentheses as a, b, \dots and right parentheses as a', b', \dots . With alphabet $\Sigma = \{a, a', b, b'\}$, the Dyck language is generated by grammar:

$$S \rightarrow aSa'S \mid bSb'S \mid \varepsilon$$

Notice we need nonlinear rules (the first two) to generate this language. To see that, compare with language L_1 of Example 2.29 on p. 29. The latter, recoding its alphabet $\{b, e\}$ as $\{a, a'\}$, is strictly included in the Dyck language, since L_1 disallows any string with two or more nests, e.g.:

$$a a \underbrace{aa'}_{\text{nest}} a' a a \underbrace{aa'}_{\text{nest}} a' a' a'$$

Such sentences have a branching syntax tree that requires nonlinear rules for its derivation.

²⁰The man who loves the woman (who sees the child (who feeds the cat)) sleeps.

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

Definition 2.43 (Parenthesized grammar) Let $G = (V, \Sigma, P, S)$ be a grammar with an alphabet Σ not containing parentheses. The parenthesized grammar G_p has alphabet $\Sigma \cup \{(')', '\}\}$ and rules:

$$A \rightarrow (\alpha) \quad \text{where } A \rightarrow \alpha \text{ is a rule of } G$$

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

$$A \rightarrow ({}_A\alpha){}_A \quad B \rightarrow ({}_B\alpha){}_B$$

where $({}_A$ and $)_A$ are parentheses subscripted with the nonterminal name.

Clearly each sentence produced by such grammars exhibits parenthesized structure.

Example 2.44 (Parenthesis grammar) The parenthesized version of the grammar for lists of palindromes (p. 30) is

$$\begin{array}{ll} list \rightarrow (pal, list) & pal \rightarrow () \\ list \rightarrow (pal) & pal \rightarrow (a pal a) \\ & pal \rightarrow (b pal b) \end{array}$$

The original sentence aa becomes the parenthesized sentence $((a()a))$.

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, 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 the grammars defining languages L_1 and L_2 . We need the unrestrictive hypothesis that nonterminal sets are disjoint, $V_1 \cap V_2 = \emptyset$. Moreover, we stipulate that symbol S , to be used as axiom of the grammar under construction, is not used by either grammar, $S \notin (V_1 \cup V_2)$.

Union: The union $L_1 \cup L_2$ is defined by the grammar containing the rules of both grammars, plus the initial rules $S \rightarrow S_1 \mid S_2$. In formulas, the grammar is

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

Concatenation: The concatenation $L_1 L_2$ is defined by the grammar containing the rules of both grammars, plus the initial rule $S \rightarrow S_1 S_2$. The grammar is

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

Star: The grammar G of the starred language $(L_1)^*$ includes the rules of G_1 and rules $S \rightarrow SS_1 \mid \varepsilon$.

Cross: From the identity $L^+ = L \cdot L^*$, the grammar of the cross language could be written 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 the rules of G_1 and rules $S \rightarrow SS_1 \mid S_1$.

From all this we have the following.

Property 2.45 The family CF of context-free languages is closed by union, concatenation, star, and cross.

Example 2.46 (Union of languages) The language

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

contains sentences of the form $a^i b^j c^k$ with $i = j \vee j = k$, such as

$$a^5 b^5 c^2, a^5 b^5 c^5, b^5 c^5$$

The rules for the component languages are straightforward:

G_1	G_2
$S_1 \rightarrow XC$	$S_2 \rightarrow AY$
$X \rightarrow aXb \mid \varepsilon$	$Y \rightarrow bYc \mid \varepsilon$
$C \rightarrow cC \mid \varepsilon$	$A \rightarrow aA \mid \varepsilon$

We just add alternatives $S \rightarrow S_1 \mid S_2$, to trigger derivation with either grammar.

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 it, replace grammar G_2 with the trivially equivalent grammar: G'' :

$$S'' \rightarrow AX \quad X \rightarrow bXc \mid \varepsilon \quad A \rightarrow aA \mid \varepsilon$$

Then the putative grammar of the union, $\{S \rightarrow S_1 \mid S''\} \cup P_1 \cup P''$, would also allow hybrid derivations using rules from both grammars, thus generating for instance $abcbc$, which is not in the union language.

Notably, Property 2.45 holds for both families REG and CF , but only the former is closed by intersection and complement, to be seen later.

Grammar of Mirror Language Examining the effect of string reversal on the sentences of a *CF* language, one immediately sees the family is closed with respect to reversal (the same as family *REG*). Given a grammar, the rules generating the mirror language are obtained reversing every right part of a rule.

2.5.10 Ambiguity

The common linguistic phenomenon of ambiguity in natural language 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 *half baked chicken*, having different meanings depending on the structure assigned:

[[half baked] chicken] or [half [baked chicken]].

Although ambiguity may cause misunderstanding in human communication, negative consequences are counteracted by availability of non-linguistic 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 2.47 Consider again the language of arithmetic expressions of Example 2.39, p. 38, but define it with a different grammar G' equivalent to the previous one:

$$E \rightarrow E + E \mid E * E \mid (E) \mid i$$

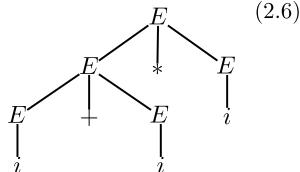
The left derivations:

$$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow i + E * E \Rightarrow i + i * E \Rightarrow i + i * i \quad (2.5)$$

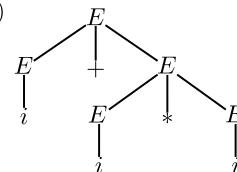
$$E \Rightarrow E + E \Rightarrow i + E \Rightarrow i + E * E \Rightarrow i + i * E \Rightarrow i + i * i \quad (2.6)$$

generate the same sentence, with different trees:

(2.5)



(2.6)



The sentence $i + i * i$ is therefore ambiguous.

Pay attention now to the meaning of the two readings of the same expression. The left tree interprets the sentence as $(i + i) * i$, the right tree assigns the interpretation

$i + (i * i)$. The latter is likely to be preferable, because it agrees with the traditional precedence of operators. Another ambiguous sentence is $i + i + i$, having two trees that differ in the order of association of subexpressions: from left to right, or the other way. As a consequence grammar G' is ambiguous.

A major defect of this grammar is that it does not force the expected precedence of product over sum.

On the other hand, for grammar G of Example 2.39 on p. 38 each sentence has only one left derivation, therefore all sentences are unambiguous, and the grammar 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, their conciseness with respect to equivalent unambiguous ones. In special situations, when one wants the simplest possible grammar 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 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 such degree may be unbounded.

Example 2.48 (Example 2.47 continued) The degree of ambiguity is 2 for sentence $i + i + i$; it is 5 for $i + i * i + i$, as one sees from the skeleton trees:

$$\underbrace{i + \underbrace{i * i + i}_{\text{one}}}_{\text{two}}, \underbrace{i + \underbrace{i * i + i}_{\text{one}}}_{\text{two}}$$

It is easy to see that longer sentences cause the degree of ambiguity 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.²¹ 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 non-ambiguity by applying some form of inductive reasoning.

In practice this is not necessary, and two approaches usually suffice. First, a small number of rather short sentences are tested, by constructing their syntax trees and checking that they are unique. If the test is passed, one has 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 to ensure non-ambiguity. Even better is to prevent the problem when a grammar is designed, by avoiding some common pitfalls to be explained next.

²¹A proof can be found in [8].

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 the cases of ambiguity are abundant in natural language, clarity of communication is not seriously impaired because sentences are uttered or written in a living context (gestures, intonation, presuppositions, etc.) that helps in selecting the interpretation intended by the author. On the contrary, in artificial languages ambiguity cannot be tolerated because machines are not as good as humans in making use of context, with the negative consequence of unpredictable behavior of the interpreter or compiler.

Now we classify the most common types of ambiguity 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 recursive if it is both left and right-recursive (i.e., it offers derivations $A \xrightarrow{+} A\gamma$ and $A \xrightarrow{+} \beta A$). We distinguish the case the two derivations are produced by the same or by different rules.

Example 2.49 (Bilateral recursion from the same rule) The grammar G_1 :

$$E \rightarrow E + E \mid i$$

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 order of generation of the string, from the left or from the right. Looking at the intended meaning as arithmetic formulas, this grammar does not specify the order of application of operations.

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

Example 2.50 (Left and right recursions in different rules) A second case of bilateral recursive ambiguity is grammar G_2 :

$$A \rightarrow aA \mid Ab \mid c$$

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

$$S \rightarrow AcB \quad A \rightarrow aA \mid \epsilon \quad B \rightarrow bB \mid \epsilon$$

An alternative remedy is to decide the first list should be generated before the second one (or conversely):

$$S \rightarrow aS \mid X \quad X \rightarrow Xb \mid c$$

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

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

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

2.5.11.2 Ambiguity from Union

If languages $L_1 = L(G_1)$ and $L_2 = L(G_2)$ share some sentences, that is, their intersection is not empty, the grammar G of the united languages, constructed as explained on p. 43, is ambiguous. (No need to repeat the two component grammars should have disjoint nonterminal sets.)

Take a sentence $x \in L_1 \cap L_2$. It is obviously produced by two distinct derivations, one using rules of G_1 , the other using rules of G_2 . The sentence is ambiguous for grammar G that contains all the rules. Notice that a sentence x belonging to the first but not the second language, $x \in L_1 \setminus L_2$, is derived by rules of G_1 only, hence is not ambiguous (if the first grammar is so).

Example 2.51 (Union of overlapping languages) In language and compiler design there are various causes for overlap.

- When one wants to single out a special pattern requiring special processing, within a general class of phrases. Consider additive arithmetic expressions with constants C and variables i . A grammar is

$$E \rightarrow E + C \mid E + i \mid C|i \quad C \rightarrow 0 \mid 1D \mid \dots \mid 9D \quad D \rightarrow 0D \mid \dots \mid 9D \mid \varepsilon$$

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, using increment instead of addition. To this end we add the rules:

$$E \rightarrow i + 1 \mid 1 + i$$

Unfortunately, the new grammar is ambiguous, since a sentence like $1 + i$ is generated by the original rules too.

- When the same operator is overloaded, i.e. used with different meanings in different constructs. In the language Pascal the sign ‘+’ denotes both addition in

$$E \rightarrow E + T \mid T \quad T \rightarrow V \quad V \rightarrow \dots$$

and set union in

$$E_{\text{set}} \rightarrow E_{\text{set}} + T_{\text{set}} \mid T_{\text{set}} \quad T_{\text{set}} \rightarrow V$$

Such ambiguities need severe grammar surgery to be eliminated: either the two ambiguous constructs are made disjoint or they are fused together. Disjunction of constructs is not feasible in the previous examples, because the string ‘1’ cannot be removed from the set of integer constants derived from nonterminal C . To enforce a special treatment of the value ‘1’, if one accepts a syntactic change to the language, it suffices to add operator inc (for increment by 1) and replace rule $E \rightarrow i + 1 \mid 1 + i$ with rule $E \rightarrow \text{inc } i$.

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

In the following examples removal of overlapping constructs actually succeeds.

Example 2.52 (McNaughton)

1. Grammar G :

$$S \rightarrow bS \mid cS \mid D \quad D \rightarrow bD \mid cD \mid \varepsilon$$

is ambiguous since $L(G) = \{b, c\}^* = L_D(G)$. The derivations:

$$S \xrightarrow{\pm} bbcD \Rightarrow bbc \quad S \Rightarrow D \xrightarrow{\pm} bbcD \Rightarrow bbc$$

produce the same result. Deleting the rules of D , which are redundant, we have
 $S \rightarrow bS \mid cS \mid \varepsilon$.

2. Grammar:

$$S \rightarrow B \mid D \quad B \rightarrow bBc \mid \varepsilon \quad D \rightarrow dDe \mid \varepsilon$$

where B generates $b^n c^n, n \geq 0$, and D generates $d^n e^n, n \geq 0$, has just one ambiguous sentence: ε . *Remedy*: generate it directly from the axiom:

$$S \rightarrow B \mid D \mid \varepsilon \quad B \rightarrow bBc \mid bc \quad D \rightarrow dDe \mid de$$

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.

Remember the grammar G of concatenation $L_1 L_2$ (p. 44) contains rule $S \rightarrow S_1 S_2$ in addition to the rules of G_1 and G_2 (by hypothesis not ambiguous). Ambiguity arises in G if the following sentences exist in the languages:

$$u' \in L_1 \quad u'v \in L_1 \quad v z'' \in L_2 \quad z'' \in L_2$$

Then string $u'vz''$ of language $L_1.L_2$ is ambiguous, via the derivations:

$$S \Rightarrow S_1 S_2 \stackrel{+}{\Rightarrow} u' S_2 \stackrel{+}{\Rightarrow} u' v z'' \quad S \Rightarrow S_1 S_2 \stackrel{+}{\Rightarrow} u' v S_2 \stackrel{+}{\Rightarrow} u' v z''$$

Example 2.53 (Concatenation of Dyck languages) For the concatenation $L = L_1 L_2$ of the Dyck languages (p. 42) L_1 and L_2 over alphabets (in the order given) $\{a, a', b, b'\}$ and $\{b, b', c, c'\}$, a sentence is $aa'bb'cc'$. The standard grammar of L is

$$S \rightarrow S_1 S_2 \quad S_1 \rightarrow a S_1 a' S_1 \mid b S_1 b' S_1 \mid \varepsilon \quad S_2 \rightarrow b S_2 b' S_2 \mid c S_2 c' S_2 \mid \varepsilon$$

The sentence is derived in two ways:

$$\overbrace{aa'}^{S_1} \overbrace{bb'}^{S_2} \overbrace{cc'}^{S_1} \quad \overbrace{aa'}^{S_1} \overbrace{bb'cc'}^{S_2}$$

To remove this ambiguity one should block the movement of 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 remedy is to interpose a new terminal as separator between the two languages. In our example, with \sharp as separator, the language $L_1 \sharp L_2$ is easily defined without ambiguity by a grammar with initial rule $S \rightarrow S_1 \sharp S_2$.

Otherwise, a more complex unambiguous solution would be to write a grammar, with the property that any string not containing c , such as bb' , is in language L_1 but not in L_2 . Notice that string $bcc'b'$ is, on the other hand, assigned to language two.

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 producing a message, i.e., a sequence of symbols from a finite set $\Gamma = \{A, B, \dots, Z\}$. Each such symbol is then encoded into a string over a terminal alphabet Σ (typically binary); a coding function maps each symbol into a short terminal string, termed its code.

Consider for instance the following source symbols and their mapping into binary codes ($\Sigma = \{0, 1\}$):

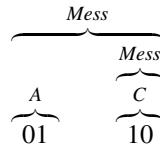
$$\Gamma = \{ \overbrace{A}^{01}, \overbrace{C}^{10}, \overbrace{E}^{11}, \overbrace{R}^{001} \}$$

Message *ARRECA* is encoded as 01 001 001 11 1001; by decoding this string the original text is the only one to be obtained. Message coding is expressed by grammar G_1 :

$$Mess \rightarrow A \text{ } Mess \mid C \text{ } Mess \mid E \text{ } Mess \mid R \text{ } Mess \mid A \mid C \mid E \mid R$$

$$A \rightarrow 01 \quad C \rightarrow 10 \quad E \rightarrow 11 \quad R \rightarrow 001$$

The grammar generates a message such as AC by concatenating the corresponding codes, as displayed in the syntax tree:



As the grammar 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:

$$\begin{aligned} \text{Mess} &\rightarrow A \text{ Mess} \mid C \text{ Mess} \mid E \text{ Mess} \mid R \text{ Mess} \mid A \mid C \mid E \mid R \\ A &\rightarrow 00 \quad C \rightarrow 01 \quad E \rightarrow 10 \quad R \rightarrow 010 \end{aligned}$$

Consequently, message 00010010100100 can be deciphered in two ways, as ARRECA or as ACAEECA.

The defect has two causes: the identity:

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

holds for two pairs of concatenated codes; and the first codes, 01 and 010, are one prefix of the other.

Code theory studies these and similar conditions that make a set of codes uniquely decipherable.

2.5.11.5 Other Ambiguous Situations

The next case is similar to the ambiguity of regular expressions (p. 21).

Example 2.54 Consider the grammar:

$$S \rightarrow DcD \quad D \rightarrow bD \mid cD \mid \varepsilon$$

Rule one says a sentence contains at least one c ; the alternatives of D generate $\{b, c\}^*$. The same language structure would be defined by regular expression $\{b, c\}^* c \{b, c\}^*$, which is ambiguous: every sentence with two or more c is ambiguous since the distinguished occurrence of c is not fixed. This defect can be repaired imposing that the distinguished c is, say, the leftmost one:

$$S \rightarrow BcD \quad D \rightarrow bD \mid cD \mid \varepsilon \quad B \rightarrow bB \mid \varepsilon$$

Notice that B may not derive a string containing c .

Example 2.55 (Setting an order on rules) In grammar:

$$S \rightarrow bSc \mid bbSc \mid \varepsilon$$

the first two rules may be applied in one or the other order, producing the ambiguous derivations:

$$S \Rightarrow bbSc \Rightarrow bbbScc \Rightarrow bbbcc \quad S \Rightarrow bSc \Rightarrow bbbScc \Rightarrow bbbcc$$

Remedy: oblige rule one to precede rule two:

$$S \rightarrow bSc \mid D \quad D \rightarrow bbDc \mid \varepsilon$$

2.5.11.6 Ambiguity of Conditional Statements

A notorious case of ambiguity in programming languages with conditional instructions occurred in the first version of language Algol 60,²² a milestone for applications of *CF* grammars. Consider grammar:

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

where *b* stands for a boolean condition and *a* for any non-conditional instruction, both left undefined for brevity. The first alternative produces a *two legs* conditional instruction, the second a *one-leg* construct.

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

$$\overbrace{\text{if } b \text{ then} \overbrace{\text{if } b \text{ then } a \text{ else } a}^{\text{one-leg}} }^{\text{two-leg}} \quad \overbrace{\text{if } b \text{ then} \overbrace{\text{if } b \text{ then } a}^{\text{one-leg}} \text{ else } a}^{\text{two-leg}}$$

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 that binds the *else* to the immediately preceding *if*. The corresponding grammar is

$$\begin{aligned} S &\rightarrow S_E \mid S_T & S_E &\rightarrow \text{if } b \text{ then } S_E \text{ else } S_E \mid a \\ S_T &\rightarrow \text{if } b \text{ then } S_E \text{ else } S_T \mid \text{if } b \text{ then } S \end{aligned}$$

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

$$\text{if } b \text{ then } S_T \text{ else } S_T \quad \text{and} \quad \text{if } b \text{ then } S_T \text{ else } S_E$$

²²The following official version [10] removed the ambiguity.

The facts that only S_E may precede *else*, and that only S_T defines a one-leg conditional, disallow 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 mark *end-if*:

$$S \rightarrow \text{if } b \text{ then } S \text{ else } S \text{ end-if} \mid \text{if } b \text{ then } S \text{ end-if} \mid a$$

Indeed this is a sort of parenthesizing (p. 43) of the original grammar.

2.5.11.7 Inherent Ambiguity of Language

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

Example 2.56 (Unavoidable ambiguity from union) Recall Example 2.46 on p. 44:

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

The language can be equivalently defined by means of the union:

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

of two non-disjoint languages.

We intuitively argue that any grammar of this language is necessarily ambiguous. The grammar on p. 44 unites the rules of the component grammars, and is obviously ambiguous for every sentence $x \in \{\varepsilon, abc, \dots a^i b^i c^i \dots\}$, shared by both languages. Any such sentence is produced by G_1 using rules checking that $|x|_a = |x|_b$, a check only possible for a syntax structure of the type:

$$\overbrace{a \dots a}^{} \underbrace{ab}_{\substack{\swarrow \\ \searrow}} \overbrace{b \dots b}^{} \underbrace{cc \dots c}_{\substack{\swarrow \\ \searrow}}$$

Now the same string x , viewed as a sentence of L_2 , must be generated with a structure of type

$$\overbrace{a \dots a}^{} \overbrace{ab \dots b}^{} \underbrace{bc}_{\substack{\swarrow \\ \searrow}} \overbrace{c \dots c}^{}_{\substack{\swarrow \\ \searrow}}$$

in order to perform the equality check $|x|_b = |x|_c$.

No matter which variation of the grammar we make, the two equality checks on the exponents are unavoidable for such sentences and the grammar remains ambiguous.

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

2.5.12 Weak and Structural Equivalence

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

We ought to reexamine the notion of grammar in the light of structural adequacy. Recall the definition on p. 35: two grammars are equivalent if they define the same language, $L(G) = L(G')$. Such definition, to be qualified now as *weak equivalence*, poorly fits with the real possibility of substituting one grammar for the other in technical artifacts such as compilers. The reason is 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 $L(G) = L(G')$ and in addition G and G' assign to each sentence two syntax trees, which may be considered *structurally similar*.

The last condition 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. 40) are equal.

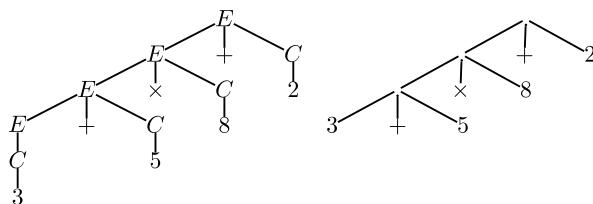
Strong equivalence implies weak equivalence, but the former is a decidable property, unlike the latter.²³

Example 2.57 (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 plus and times signs.

- First grammar G_1 :

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

The syntax tree of the previous sentence is



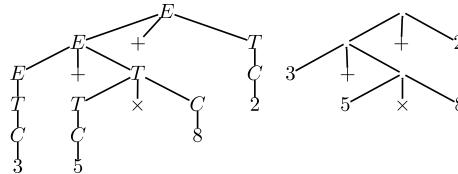
In the condensed skeleton (right), nonterminals and copy rules have been dropped.

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

- A second grammar G_2 for this language is

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

The two grammars are weakly equivalent. Observing now the syntax tree of the same sentence:



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

Is either one of the grammars preferable? Concerning ambiguity, both grammars are all right. But only grammar G_2 is structurally adequate, if one considers also meaning. In fact, the 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 specifying the order of evaluation $((3 + (5 \times 8)) + 2)$ can be mapped on the subtrees of the skeletal tree produced by G_2 .

On the contrary, grammar G_1 produces the parenthesizing $((3 + 5) \times 8) + 2$ which is not adequate, because by giving precedence to the first sum over the product, it returns the wrong semantic interpretation, 66. Incidentally, the second grammar 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 shall see in the last chapter on syntax-directed translations.

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

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

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

²⁴This grammar has more rules because it does not exploit copy rules to express inclusion of syntax classes. Categorization and ensuing taxonomies reduce the complexity of descriptions in any area of knowledge.

2.5.12.1 Generalized Structural Equivalence

Sometimes a looser criterion of similarity than strict identity of condensed skeleton trees is more suitable. This consists in requesting that the two corresponding trees should be easily mapped one on 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. For instance, the grammars:

$$\{S \rightarrow Sa \mid a\} \quad \text{and} \quad \{X \rightarrow aX \mid a\}$$

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

$$\underbrace{a \ a}_{\text{and}} \quad \underbrace{a \ a}_{\text{and}}$$

However, the two grammars may be considered structurally equivalent in a generalized sense because each left-linear tree of the first grammar corresponds to a right-linear tree of the second. The intuition that the two grammars are similar is satisfied because their trees are specularly identical, i.e., they become coincident by turning left-recursive rules into right-recursive (or conversely).

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 allowing any context-free language to be defined. Such forms are widely used in theoretical papers, to simplify statements and proofs of theorems. Otherwise, in applied work, grammars in normal form are rarely used because they are larger and less readable. However some normal forms have practical uses for language analysis in Chap. 4, in particular the non-left-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 simple ones.

Let grammar $G = (V, \Sigma, P, S)$ be given.

2.5.13.1 Nonterminal Expansion

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

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

$$A \rightarrow \alpha \beta_1 \gamma \mid \alpha \beta_2 \gamma \mid \dots \mid \alpha \beta_n \gamma$$

where $B \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$ 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., be a string over alphabet $(\Sigma \cup (V \setminus \{S\}))$. To this end, just introduce a new axiom S_0 and rule $S_0 \rightarrow S$.

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., $A \stackrel{*}{\Rightarrow} \varepsilon$.

Consider the set named $Null \subseteq V$ of nullable nonterminals. The set 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:

$$A \in Null \text{ if } A \rightarrow \varepsilon \in P$$

$$A \in Null \text{ if } (A \rightarrow A_1 A_2 \dots A_n \in P \text{ with } A_i \in V \setminus \{A\}) \wedge \forall A_i (A_i \in Null)$$

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

Example 2.58 (Computing nullable nonterminals)

$$S \rightarrow SAB \mid AC \quad A \rightarrow aA \mid \varepsilon \quad B \rightarrow bB \mid \varepsilon \quad C \rightarrow cC \mid c$$

Result: $Null = \{A, B\}$.

Notice that, if rule $S \rightarrow AB$ were added to the grammar, the result would include $S \in Null$.

The *normal form without nullable nonterminals*, for brevity *non-nullable*, is defined by the condition that no nonterminal other than the axiom is nullable. Moreover, the axiom is nullable if, and only if, the empty string is in the language.

To construct the non-nullable form, first compute the set $Null$, then do as follows:

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

If the grammar thus obtained is unclean or circular, it should be cleaned with the known algorithms (p. 35).

Example 2.59 (Example 2.58 continued) In Table 2.4, column one lists nullability. The other columns list the original rules, those produced by the transformation, and the final clean rules.

2.5.13.4 Copies or Subcategorization Rules and Their Elimination

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

Table 2.4 Elimination of copy rules

Nullable	G original	G' to be cleaned	G' non-nullables normal
F	$S \rightarrow SAB \mid AC$	$S \rightarrow SAB \mid SA \mid SB \mid S \mid AC \mid C$	$S \rightarrow SAB \mid SA \mid SB \mid AC \mid C$
V	$A \rightarrow aA \mid \varepsilon$	$A \rightarrow aA \mid a$	$A \rightarrow aA \mid a$
V	$B \rightarrow bB \mid \varepsilon$	$B \rightarrow bB \mid b$	$B \rightarrow bB \mid b$
F	$C \rightarrow cC \mid c$	$C \rightarrow cC \mid c$	$C \rightarrow cC \mid c$

For a concrete example, the rules

$$\text{iterative_phrase} \rightarrow \text{while_phrase} \mid \text{for_phrase} \mid \text{repeat_phrase}$$

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

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

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

$$\text{Copy}(A) = \{B \in V \mid \text{there is a derivation } A \xrightarrow{*} B\}$$

Note: if nonterminal C is nullable, the derivation may take the form:

$$A \xrightarrow{+} BC \Rightarrow B$$

For simplicity we assume the grammar is non-nullables and the axiom does not occur in a right part.

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

$$A \in \text{Copy}(A) \quad -- \text{ initialization}$$

$$C \in \text{Copy}(A) \text{ if } (B \in \text{Copy}(A)) \wedge (B \rightarrow C \in P)$$

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

$$P' := P \setminus \{A \rightarrow B \mid A, B \in V\} \quad -- \text{ copy cancelation}$$

$$P' := P' \cup \{A \rightarrow \alpha \mid \alpha \in ((\Sigma \cup V)^* \setminus V)\} \quad \text{where } (B \rightarrow \alpha) \in P \wedge B \in \text{Copy}(A)$$

The effect is that the old grammar derivation $A \xrightarrow{+} B \Rightarrow \alpha$ shrinks to the immediate derivation $A \Rightarrow \alpha$.

Notice the transformation keeps all original non-copy rules. In Chap. 3 the same transformation will be applied to remove spontaneous moves from an automaton.

Example 2.60 (Copy-free rules for arithmetic expressions) Applying the algorithm to grammar G_2 :

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

we obtain

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

The equivalent copy-free grammar is

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

It is worth repeating that copy rules are very convenient for reusing certain blocks of rules, corresponding to syntactic subcategories; the grammar is concise and evidences the generalization and specialization of language constructs. For these reasons, reference manuals of technical languages cannot do without copy rules.

2.5.13.5 Repeated Right Parts and Invertible Grammars

In Example 2.60 the last grammar contains two rules with repeated right part: $E \rightarrow T \times C$ and $T \rightarrow T \times C$. On the other hand, a grammar, such as G_2 , that 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's in grammars.

A convenient use of RRP's is when two nonterminals C_1, C_2 include in their languages a common part R and also two idiosyncratic parts, respectively denoted A_1 and A_2 , schematically: $C_1 \rightarrow R \mid A_1$, $C_2 \rightarrow R \mid A_2$. Since R can be the root of arbitrarily complex constructs, the use of the RRP may save considerable duplication in the grammar rules. A concrete example occurs in many programming languages: two constructs for assignment statements exist, one of general use and the other restricted to variable initialization, say by assigning constant values. Clearly the latter offers a subset 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 show²⁵ how to convert every grammar $G = (V, \Sigma, P, S)$ into an equivalent invertible grammar $G' = (V', \Sigma, P', S')$. Without loss of generality we assume that grammar G is non-nullifiable, i.e., free from empty rules (except $S \rightarrow \varepsilon$ if needed).

²⁵We refer to the book by Harrison [7] for more details and a correctness proof.

The nonterminals of G and G' (apart the axioms S and S') are respectively denoted by letters B, B_1, \dots, B_m and A, A_1, \dots, A_n . Moreover, apart from S' , each nonterminal of G' is uniquely identified by a nonempty subset of V , i.e., $V' = \{S'\} \cup V''$, where V'' is the set of nonempty subsets of V ; to emphasize that subsets of V are used as identifiers for nonterminals of G' , we denote them enclosing the subset elements between square brackets. (A similar convention will be adopted in Sect. 3.7.1 in the construction of accessible subsets for the determinization of finite automata.) For grammar G' , we construct the rules specified by the next clauses (2.7) and (2.8).

$$\{S' \rightarrow A \mid A \text{ is a subset of } V \text{ that contains axiom } S\}. \quad (2.7)$$

For each rule of G of form $B \rightarrow x_0 B_1 x_1 \dots B_r x_r$

with $r \geq 0$, $B_1, \dots, B_r \in V$, $x_0, x_1 \dots, x_r \in \Sigma^*$,

for each $A_1 \dots A_r \in V''$,

$A \rightarrow x_0 A_1 x_1 \dots A_r x_r$,

where nonterminal A is identified by the set

$[C \mid C \rightarrow x_0 C_1 x_1 \dots C_r x_r \text{ is in } P \text{ for some } C_1, \dots, C_r \text{ with each } C_i \in A_i]$

$$(2.8)$$

Notice that this construction typically produces also useless rules and nonterminals, which can be later eliminated.

Example 2.61 (Elimination of RRP's) We apply the construction to the non-invertible grammar G :

$$S \rightarrow aB_1 \mid bB_2, \quad B_1 \rightarrow aB_1 \mid aS \mid bB_2, \quad B_2 \rightarrow a \mid b$$

We first use clause (2.7) and obtain the rules

$$S' \rightarrow \{S\} \mid \{B_1, S\} \mid \{B_2, S\} \mid \{B_1, B_2, S\}$$

Then, applying clause (2.8), we have

$$V'' = \{\{B_1\}, \{B_2\}, \{S\}, \{B_1, B_2\}, \{B_1, S\}, \{B_2, S\}, \{B_1, B_2, S\}\}$$

and we consider the groups of rules of G that have the same pattern of terminal symbols in their right part, yielding the rules on the following table.

Original rules	Rules created
$B_1 \rightarrow bB_2$ and $S \rightarrow bB_2$	$[B_1, S] \rightarrow b[B_2] \mid b[B_1, S] \mid b[B_2, S] \mid a[B_1, B_2, S]$
$B_1 \rightarrow aB_1 \mid aS$ and	$[B_1, S] \rightarrow a[B_1] \mid a[B_1, B_2] \mid a[B_1, S] \mid a[B_1, B_2, S]$
$S \rightarrow aB_1$	$[B_1] \rightarrow a[S] \mid a[B_2, S]$
$B_2 \rightarrow a$	$[B_2] \rightarrow a$
$B_2 \rightarrow b$	$[B_2] \rightarrow b$

After cleanup, P' reduces to

$$P' = \{S' \rightarrow [B_1, S], [B_1, S] \rightarrow a[B_1, S] | bB_2, [B_2] \rightarrow a | b\}$$

As an alternative implementation of the construction, it is quicker to obtain the equivalent invertible grammar G' starting from the terminal rules, and considering only the nonterminals (= subsets of V) that are created (as the other ones generate the empty language \emptyset); in this way we do not have to exhaustively examine all the subsets. We tabulate the nonterminals and rules created step by step:

Clause	Original rules	Rules created
(2.8)	$B_2 \rightarrow a b$	$[B_2] \rightarrow a b$
(2.8)	$B_1 \rightarrow bB_2$ and $S \rightarrow bB_2$	$[B_1, S] \rightarrow b[B_2]$
(2.8)	$B_1 \rightarrow aB_1 aS$ and $S \rightarrow aB_1$	$[B_1, S] \rightarrow a[B_1, S]$

Adopting the same criterion, of considering only the subsets already obtained as the other ones generate \emptyset , we apply clause (2.7) obtaining the following axiomatic rule:

$$S' \rightarrow [B_1, S]$$

Notice that the invertible grammar preserves the syntactic structure; however, in principle, it may contain many more nonterminals, although in practice this does not happen for technical grammars.

2.5.13.6 Chomsky and Binary Normal Form

In Chomsky's normal form there are two types of rules:

1. *homogeneous binary*: $A \rightarrow BC$, where $B, C \in V$
2. *terminal with singleton right part*: $A \rightarrow a$, where $a \in \Sigma$

Moreover, if the empty string is in the language, there is rule $S \rightarrow \varepsilon$ but the axiom is not allowed in any right part.

With such constraints any internal node of a syntax tree may have either two nonterminal siblings or one terminal son.

Given a grammar, by simplifying hypothesis without nullable nonterminals, we explain how to obtain a Chomsky normal form. Each rule $A_0 \rightarrow A_1A_2 \dots A_n$ of length $n > 2$ is converted to a length 2 rule, singling out the first symbol A_1 and the remaining suffix $A_2 \dots A_n$. Then a new ancillary nonterminal is created, named $\langle A_2 \dots A_n \rangle$, and the new rule:

$$\langle A_2 \dots A_n \rangle \rightarrow A_2 \dots A_n$$

Now the original rule is replaced by

$$A_0 \rightarrow A_1 \langle A_2 \dots A_n \rangle$$

If symbol A_1 is terminal, we write instead the rules:

$$A_0 \rightarrow \langle A_1 \rangle \langle A_2 \dots A_n \rangle \quad \langle A_1 \rangle \rightarrow A_1$$

where $\langle A_1 \rangle$ is a new ancillary nonterminal.

Continue applying the same series of transformations to the grammar thus obtained, until all rules are in the form requested.

Example 2.62 (Conversion to Chomsky normal form) The grammar:

$$S \rightarrow dA | cB \quad A \rightarrow dAA | cS | c \quad B \rightarrow cBB | dS | d$$

becomes:

$$\begin{aligned} S &\rightarrow \langle d \rangle A | \langle c \rangle B & A &\rightarrow \langle d \rangle \langle AA \rangle | \langle c \rangle S | c & B &\rightarrow \langle c \rangle \langle BB \rangle | \langle d \rangle S | d \\ \langle d \rangle &\rightarrow d & \langle c \rangle &\rightarrow c & \langle AA \rangle &\rightarrow AA & \langle BB \rangle &\rightarrow BB \end{aligned}$$

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 such that each right part contains at most two nonterminals.

2.5.13.7 Operator Grammars

A grammar is in *operator form* if at least one terminal character is interposed between any two nonterminal symbols occurring in the right part of every rule, in formula:

$$\text{for every rule } A \rightarrow \alpha : \alpha \cap ((\Sigma \cup V)^* V V (\Sigma \cup V)^*) = \emptyset.$$

Since the operator form of grammars is used in some efficient language parsing algorithms, we explain how to convert a grammar to such form. 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 expression is already in operator form:

$$E \rightarrow E + T \mid T, \quad T \rightarrow T \times F \mid F, \quad F \rightarrow a \mid (E)$$

whereas in the next grammar of *prefix* Boolean expressions, in the sense that every operator precedes its operand(s), the first rule is not in operator form:

$$E \rightarrow BEE \mid \neg E \mid a, \quad B \rightarrow \wedge \mid \vee$$

Consider a grammar G , which we assume for simplicity to be in a form, slightly more general than Chomsky normal form, such that every rule contains at most two nonterminals in its right part. We explain²⁶ how to obtain an equivalent grammar H in operator form.

1. For each existing nonterminal X and for each terminal a , we create a new nonterminal, denoted X_a , with the stipulation that X_a generates a string u if, and only if, X generates string ua .

²⁶Following [1] where a proof of correctness of the transformation can be found.

Therefore, for the languages respectively generated by nonterminals X and X_a using the rules of grammar G and H , the identity holds:

$$L_G(X) = L_H(X_a)a \cup L_H(X_b)b \cup \dots \cup \{\varepsilon\}$$

where the union spans all 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 X_b is useless.

2. For every rule $A \rightarrow \alpha$ of G , for every nonterminal X occurring in the right part α , for all terminals a, b, \dots , replace X by the set of strings:

$$X_a a, X_b b, \dots, \varepsilon \quad (2.9)$$

where string ε is only needed if X is nullable.

3. To finish the construction, for every rule $X \rightarrow \alpha a$ computed at (2.9), we create the rule $X_a \rightarrow \alpha$.

As in similar cases, the construction may produce also useless nonterminals.

Example 2.63 (From [1]) The grammar $G = \{S \rightarrow aSS \mid b\}$ is not in operator form.

The nonterminals created at step 1 are S_a, S_b . The rules created by (2.9) are

$$S \rightarrow aS_a aS_a a \mid aS_a aS_b b \mid aS_b bS_a a \mid aS_b bS_b b \mid b$$

Notice that the empty string is not in $L_G(S)$.

Then we obtain from step 3 the rules:

$$S_a \rightarrow aS_a aS_a \mid aS_b bS_a \quad \text{and} \quad S_b \rightarrow aS_a aS_b \mid aS_b bS_b \mid \varepsilon$$

At last, since S_a generates nothing, we delete rules $S_a \rightarrow aS_a aS_a \mid aS_b bS_a$ and $S_b \rightarrow aS_a aS_b$, thus obtaining grammar:

$$H = \{S \rightarrow aS_b bS_b b \mid b, S_b \rightarrow aS_b bS_b \mid \varepsilon\}.$$

The operator form of the rules, combined with a sort of precedence relation between terminal characters, yields the class of Operator Precedence grammars, to be studied in Chap. 4 for their nice application in parallel parsing algorithms.

2.5.13.8 Conversion of Left to Right Recursions

Another normal form termed *not left-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-recursions to right recursions.

Transformation of Immediate l-Recursions The more common and easier case is when the l-recursion to be eliminated is immediate. Consider the l-recursive alternatives of a nonterminal A :

$$A \rightarrow A\beta_1 \mid A\beta_2 \mid \dots \mid A\beta_h, \quad h \geq 1$$

where no β_i is empty, and let

$$A \rightarrow \gamma_1 | \gamma_2 | \dots | \gamma_k, \quad k \geq 1$$

be the remaining alternatives.

Create a new ancillary nonterminal A' and replace the previous rules with the next ones:

$$\begin{aligned} A &\rightarrow \gamma_1 A' | \gamma_2 A' | \dots | \gamma_k A' | \gamma_1 | \gamma_2 | \dots | \gamma_k \\ A' &\rightarrow \beta_1 A' | \beta_2 A' | \dots | \beta_h A' | \beta_1 | \beta_2 | \dots | \beta_h \end{aligned}$$

Now every original l-recursive derivation, as for instance:

$$A \Rightarrow A\beta_2 \Rightarrow A\beta_3\beta_2 \Rightarrow \gamma_1\beta_3\beta_2$$

is replaced with the equivalent right-recursive derivation:

$$A \Rightarrow \gamma_1 A' \Rightarrow \gamma_1\beta_3 A' \Rightarrow \gamma_1\beta_3\beta_2$$

Example 2.64 (Converting immediate l-recursions to right recursion) In the usual grammar of expressions:

$$E \rightarrow E + T \mid T \quad T \rightarrow T * F \mid F \quad F \rightarrow (E) \mid i$$

nonterminals E and T are immediately l-recursive. Applying the transformation, the right-recursive grammar is obtained:

$$\begin{aligned} E &\rightarrow TE' \mid T \quad E' \rightarrow +TE' \mid +T \\ T &\rightarrow FT' \mid F \quad T' \rightarrow *FT' \mid *F \quad F \rightarrow (E) \mid i \end{aligned}$$

Actually, in this case, but not always, a simpler solution is possible, to specularly reverse the l-recursive rules, obtaining

$$E \rightarrow T + E \mid T \quad T \rightarrow F * T \mid F \quad F \rightarrow (E) \mid i$$

Transformation of Non-immediate Left Recursions The next algorithm is used to transform non-immediate l-recursions. We present it under the simplifying assumptions that grammar G is homogeneous, non-nullable, with singleton terminal rules; in other words, the rules are like in Chomsky normal form, but more than two nonterminals are permitted in a right part.

There are two nested loops; the external loop employs nonterminal expansion to change non-immediate into immediate l-recursions, thus shortening the length of derivations. The internal loop converts immediate l-recursions to right recursions; in so doing it creates ancillary nonterminals.

Let $V = \{A_1, A_2, \dots, A_m\}$ be the nonterminal alphabet and A_1 the axiom. For orderly scanning, we view the nonterminals as an (arbitrarily) ordered set, from 1 to m .

Table 2.5 Turning recursion from left to right

<i>i</i>	<i>j</i>	Grammar
1		Eliminate immediate l-recursions of A_1 (none)
2 1		Replace $A_2 \rightarrow A_1d$ with the rules constructed expanding A_1 , obtaining
		Eliminate the immediate l-recursion, obtaining G'_3 :

Algorithm for Left Recursion Eliminationfor $i := 1$ to m do for $j := 1$ to $i - 1$ do replace every rule of type $A_i \rightarrow A_j\alpha$, where $i > j$, with the rules: $A_i \rightarrow \gamma_1\alpha | \gamma_2\alpha | \dots | \gamma_k\alpha$

(– possibly creating immediate l-recursions)

 where $A_j \rightarrow \gamma_1 | \gamma_2 | \dots | \gamma_k$ are the alternatives of nonterminal A_j

end do

eliminate, by means of the previous algorithm, any immediate l-recursion

 that may have arisen as alternative of A_i , creating the ancillary nonterminal A'_i

end do

The idea²⁷ is to modify the rules in such a way that, if the right part of a rule $A_i \rightarrow A_j \dots$ starts with a nonterminal A_j , then it is $j > i$, i.e., the latter nonterminal follows in the ordering.

Example 2.65 Applying the algorithm to grammar G_3 :

$$A_1 \rightarrow A_2a | b \quad A_2 \rightarrow A_2c | A_1d | e$$

which has the l-recursion $A_1 \Rightarrow A_2a \Rightarrow A_1da$, we list in Table 2.5 the steps producing grammar G'_3 , which has no l-recursion.

It would be straightforward to modify the algorithm to transform right recursions to left ones, 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:

$$A \rightarrow a\alpha \quad \text{where } a \in \Sigma, \alpha \in \{\Sigma \cup V\}^*$$

²⁷A proof of correctness may be found in [8] or in [3].

A special case of this is the *Greibach* normal form:

$$A \rightarrow a\alpha \quad \text{where } a \in \Sigma, \alpha \in V^*$$

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 parsing algorithm: at each step it reads and consumes a terminal character, thus the total number of steps equals the length of the string to be parsed. Assuming for simplicity the given grammar to be non-nullifiable, we explain how to proceed to obtain the above forms.

For the real-time form: first eliminate all left-recursions; then, by elementary transformations, expand any nonterminal that occurs in first position in a right part, until a terminal prefix is produced. Then 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 it.

Example 2.66 The grammar:

$$A_1 \rightarrow A_2a \quad A_2 \rightarrow A_1c | bA_1 | d$$

is converted to Greibach form by the following steps.

1. Eliminate l-recursions by the step:

$$A_1 \rightarrow A_2a \quad A_2 \rightarrow A_2ac | bA_1 | d$$

and then:

$$A_1 \rightarrow A_2a \quad A_2 \rightarrow bA_1A'_2 | dA'_2 | d | bA_1 \quad A'_2 \rightarrow acA'_2 | ac$$

2. Expand the nonterminals in first position until a terminal prefix is produced:

$$\begin{aligned} A_1 &\rightarrow bA_1A'_2a | dA'_2a | da | bA_1a \quad A_2 \rightarrow bA_1A'_2 | dA'_2 | d | bA_1 \\ A'_2 &\rightarrow acA'_2 | ac \end{aligned}$$

3. Substitute ancillary nonterminals for any terminal in a position other than one:

$$\begin{aligned} A_1 &\rightarrow bA_1A'_2\langle a \rangle | dA'_2\langle a \rangle | d\langle a \rangle | bA_1\langle a \rangle \quad A_2 \rightarrow bA_1A'_2 | dA'_2 | d | bA_1 \\ A'_2 &\rightarrow a\langle c \rangle A'_2 | a\langle c \rangle \\ \langle a \rangle &\rightarrow a \quad \langle c \rangle \rightarrow c \end{aligned}$$

Halting before the last step, the grammar would be in real-time but not in Greibach’s form.

Although not all preceding transformations, especially not the Chomsky and Greibach ones, will be used in this book, 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.

Table 2.6 From subexpressions to grammar rules

	Subexpression	Grammar rule
1	$r = r_1.r_2 \dots r_k$	$E \rightarrow E_1 E_2 \dots E_k$
2	$r = r_1 \cup r_2 \cup \dots \cup r_k$	$E \rightarrow E_1 \mid E_2 \mid \dots \mid E_k$
3	$r = (r_1)^*$	$E \rightarrow E E_1 \mid \varepsilon$ or $E \rightarrow E_1 E \mid \varepsilon$
4	$r = (r_1)^+$	$E \rightarrow E E_1 \mid E_1$ or $E \rightarrow E_1 E \mid E_1$
5	$r = b \in \Sigma$	$E \rightarrow b$
6	$r = \varepsilon$	$E \rightarrow \varepsilon$

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 shall also see that longer sentences present unavoidable repetitions, a property that can be exploited to prove that certain context-free languages are not regular. Other contrastive properties of the *REG* and *CF* families will emerge in Chaps. 3 and 4 from consideration of the amount of memory needed to check whether a string is in the language, which is finite for the former and unbounded for the latter family.

2.6.1 From Regular Expressions to Context-Free Grammars

Given an r.e. it is straightforward to write a grammar for the language, by analyzing the expression and mapping its subexpressions into grammar rules. At the heart of the construction, the iterative operators (star and cross) are replaced by unilaterally recursive rules.

Algorithm 2.67 (From r.e. to grammar) First we identify and number the subexpressions contained in the given r.e. r . From the very definition of the r.e., the possible cases and corresponding grammar rules (with uppercase nonterminals) are in Table 2.6. Notice we allow the empty string as term.

For shortening 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 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 as a distinguishing subscript to a nonterminal. The axiom is associated with the first step, i.e., to the whole r.e. An example should suffice to understand the procedure.

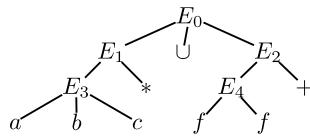
Example 2.68 (From r.e. to grammar) The expression:

$$E = (abc)^* \cup (ff)^+$$

Table 2.7 Mapping the r.e. of Example 2.68 on grammar rules

Mapping	Subexpression	Grammar rule
2	$E_1 \cup E_2$	$E_0 \rightarrow E_1 \mid E_2$
3	E_3^*	$E_1 \rightarrow E_1 E_3 \mid \varepsilon$
4	E_4^+	$E_2 \rightarrow E_2 E_4 \mid E_4$
1	abc	$E_3 \rightarrow abc$
1	ff	$E_4 \rightarrow ff$

is analyzed into the arbitrarily numbered subexpressions shown in the tree, which is a sort of syntax tree of the r.e. with added numbering:



We see that E_0 is union of subexpressions E_1 and E_2 , E_1 is star of subexpression E_3 , etc.

The mapping scheme of Table 2.6 yields the rules in Table 2.7. The axiom derives the sentential forms E_1 and E_2 ; nonterminal E_1 generates the string forms E_3^* , and from them $(abc)^*$. Similarly E_2 generates strings E_4^+ and finally $(ff)^+$.

Notice that if the r.e. is ambiguous (p. 21), the grammar is so (see Example 2.72 on p. 69).

We have thus seen how to map each operator of an r.e. on equivalent rules, to generate the same language. It follows that every regular language is context-free. Since we know of context-free languages which are not regular (e.g., palindromes and Dyck language), the following property holds.

Property 2.69 The family of regular languages REG is strictly included in the family of context-free languages CF , that is, $REG \subset CF$.

2.6.2 Linear Grammars

Algorithm 2.67 converts an r.e. to a grammar substantially preserving the structure of the r.e. But for a regular language it is possible to constrain the grammar to a very simple form of rules, called unilinear or of type 3. Such form gives evidence to some fundamental properties and leads to a straightforward construction of the automaton which recognizes the strings of a regular language.

We recall a grammar is *linear* if every rule has the form

$$A \rightarrow uBv \quad \text{where } u, v \in \Sigma^*, B \in (V \cup \varepsilon)$$

i.e., at most one nonterminal is in the right part.

Visualizing a corresponding syntax tree, we see it never branches into two subtrees but it has a linear structure made by a stem with leaves directly attached to it. Linear grammars are not powerful enough to generate all context-free languages (an example is Dyck language), but already exceed the power needed for regular languages. For instance, the following well-known subset of Dyck language is generated by a linear grammar but is not regular (to be proved on p. 77).

Example 2.70 (Non-regular linear language)

$$L_1 = \{b^n e^n \mid n \geq 1\} = \{be, bbee, \dots\}$$

Linear grammar: $S \rightarrow bSe \mid be$

Definition 2.71 (Left- and right-linearity) A rule of the following form is called *right-linear*:

$$A \rightarrow uB \quad \text{where } u \in \Sigma^*, \ B \in (V \cup \epsilon)$$

Symmetrically, a *left-linear* rule has the form

$$A \rightarrow Bu, \quad \text{with the same stipulations.}$$

Clearly both cases are linear and are obtained by deleting on either side a terminal string embracing nonterminal B .

A grammar such that all the rules are right-linear or all the rules are left-linear is termed *unilinear* or of type 3.²⁸

For a right-linear grammar every syntax tree has an oblique stem oriented towards the right (towards the left for a left-linear grammar). Moreover, if the grammar is recursive, it is necessarily right-recursive.

Example 2.72 The strings containing aa and ending with b are defined by the (ambiguous) r.e.:

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

The language is generated by the unilinear grammars:

1. Right-linear grammar G_r :

$$S \rightarrow aS \mid bS \mid aaA \quad A \rightarrow aA \mid bA \mid b$$

2. Left-linear grammar G_l :

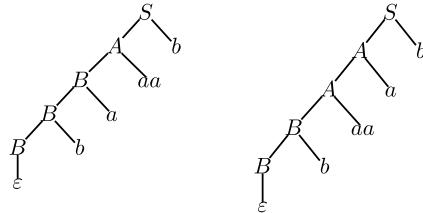
$$S \rightarrow Ab \quad A \rightarrow Aa \mid Ab \mid Baa \quad B \rightarrow Ba \mid Bb \mid \epsilon$$

²⁸Within Chomsky hierarchy (p. 86).

An equivalent non-unilinear grammar is constructed by the algorithm on p. 67:

$$E_1 \rightarrow E_2 aa E_2 b \quad E_2 \rightarrow E_2 a \mid E_2 b \mid \varepsilon$$

With grammar G_l the leftwards syntax trees of the ambiguous sentence $baaab$ are



Example 2.73 (Parenthesis-free arithmetic expressions) The language

$$L = \{a, a + a, a * a, a + a * a, \dots\}$$

is defined by the right-linear grammar G_r :

$$S \rightarrow a \mid a + S \mid a * S$$

or by the left-linear grammar G_l :

$$S \rightarrow a \mid S + a \mid S * a$$

By the way, neither grammar is adequate to impose the precedence of arithmetic operations.

Strictly Unilinear Grammars The unilinear rule form can be further constrained, with the aim of simplifying the coming discussion of theoretical properties and the construction of language recognizing automata. A grammar is *strictly unilinear* if every rule contains at most one terminal character, i.e., if it has the form

$$A \rightarrow aB \quad (\text{or } A \rightarrow Ba), \quad \text{where } a \in (\Sigma \cup \varepsilon), \quad B \in (V \cup \varepsilon)$$

A further simplification is possible: to impose that the only terminal rules are empty ones. In this case we may assume the grammar contains just the following rule types:

$$A \rightarrow aB \mid \varepsilon \quad \text{where } a \in \Sigma, \quad B \in V$$

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 2.74 (Example 2.73 continued) By adding ancillary nonterminals, the right-linear grammar G_r is transformed to the equivalent strictly right-linear gram-

mar G'_r :

$$S \rightarrow a \mid aA \quad A \rightarrow +S \mid *S$$

and also to the equivalent grammar with null terminal rules:

$$S \rightarrow aA \quad A \rightarrow +S \mid *S \mid \varepsilon$$

2.6.3 Linear Language Equations

Continuing the study of unilinear grammars, we show the languages they generate are regular. The proof consists of transforming the rules to a set of linear equations, having regular languages as their solution. In Chap. 3 we shall see that every regular language can be defined by a unilinear grammar, thus proving the identity of the languages defined by r.e. and by unilinear grammars.

For simplicity take a grammar $G = (V, \Sigma, P, S)$ in strictly right-linear form (the case of left-linear grammar is analogous) with null terminal rules.

Any such rule can be transcribed into a linear equation having as unknowns the languages generated from each nonterminal, that is, for nonterminal A :

$$L_A = \{x \in \Sigma^* \mid A \stackrel{+}{\Rightarrow} x\}$$

and in particular, $L(G) \equiv L_S$.

A string $x \in \Sigma^*$ is in language L_A if:

- x is the empty string and P contains rule $A \rightarrow \varepsilon$;
- x is the empty string, P contains rule $A \rightarrow B$ and $\varepsilon \in L_B$;
- $x = ay$ starts with character a , P contains rule $A \rightarrow aB$ and string $y \in \Sigma^*$ is in language L_B .

Let $n = |V|$ be the number of nonterminals. Each nonterminal A_i is defined by a set of alternatives:

$$A_i \rightarrow aA_1 \mid bA_1 \mid \dots \mid \dots \mid aA_n \mid bA_n \mid \dots \mid A_1 \mid \dots \mid A_n \mid \varepsilon$$

some possibly missing.²⁹ We write the corresponding equation:

$$L_{A_i} = aL_{A_1} \cup bL_{A_1} \cup \dots \cup aL_{A_n} \cup bL_{A_n} \cup \dots \cup L_{A_1} \cup \dots \cup L_{A_n} \cup \varepsilon$$

The last term disappears if the rule does not contain the alternative $A_i \rightarrow \varepsilon$.

This system of n simultaneous equations in n unknowns (the languages generated by the nonterminals) can be solved by the well-known method of Gaussian elimination, by applying the following formula to break recursion.

²⁹In particular, alternative $A_i \rightarrow A_i$ is never present since the grammar is noncircular.

Property 2.75 (Arden identity) The equation

$$X = KX \cup L \quad (2.10)$$

where K is a nonempty language and L any language, has one and only one solution:

$$X = K^*L \quad (2.11)$$

It is simple to see language K^*L is a solution of (2.10) because, substituting it for the unknown in both sides, the equation turns into the identity:

$$K^*L = (KK^*L) \cup L$$

It would also be possible to prove that equation (2.10) has no solution other than (2.11).

Example 2.76 (Language equations) The right-linear grammar:

$$S \rightarrow sS \mid eA \quad A \rightarrow sS \mid \varepsilon$$

defines a list of (possibly missing) elements e , divided by separator s . It is transcribed to the system:

$$\begin{cases} L_S = sL_S \cup eL_A \\ L_A = sL_S \cup \varepsilon \end{cases}$$

Substitute the second equation into the first:

$$\begin{cases} L_S = sL_S \cup e(sL_S \cup \varepsilon) \\ L_A = sL_S \cup \varepsilon \end{cases}$$

Then apply the distributive property of concatenation over union, to factorize variable L_S as a common suffix:

$$\begin{cases} L_S = (s \cup es)L_S \cup e \\ L_A = sL_S \cup \varepsilon \end{cases}$$

Apply the Arden identity to the first equation, obtaining

$$\begin{cases} L_S = (s \cup es)^*e \\ L_A = sL_S \cup \varepsilon \end{cases}$$

and then $L_A = s(s \cup es)^*e \cup \varepsilon$.

Notice that it is straightforward to write the equations also for unilinear grammars, which are not strictly unilinear. We have thus proved that every unilinearly generated language is regular.

An alternative method for computing the r.e. of a language defined by a finite automaton will be described in Chap. 3.

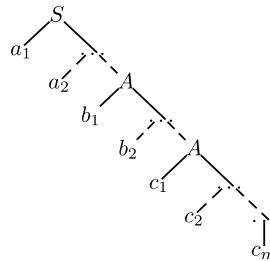
2.7 Comparison of Regular and Context-Free Languages

It is important to understand the scope of regular languages, in order to realize which constructs can be thus defined, and which require the full power of context-free grammars. To this end we present a structural property of regular languages. Recall first that, in order to generate an infinite language, a grammar must be recursive (Property 2.37, p. 37), because only recursive derivations, such as $A \xrightarrow{+} uAv$, can be iterated n (unbounded) times, producing the string u^nAv^n . This fact leads to the observation that any sufficiently large sentence necessarily includes a recursive derivation in its generation; therefore it contains certain substrings that can be unboundedly repeated, producing a longer sentence of the language.

This observation will be stated more precisely, first for unilateral, then for context-free grammars.

Property 2.77 (Pumping of strings) Take a unilinear grammar G . For any sufficiently long sentence x , meaning of length greater than some grammar dependent constant, it is possible to find a factorization $x = tuv$, where string u is not empty, such that, for every $n \geq 0$, the string $tu^n v$ is in the language. (It is customary to say the given sentence can be “pumped” by injecting arbitrarily many times the substring u .)

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 $t = a_1 a_2 \dots$, $u = b_1 b_2 \dots$, and $v = c_1 c_2 \dots c_m$. Therefore there is a recursive derivation:

$$S \xrightarrow{+} tA \xrightarrow{+} tuA \xrightarrow{+} tuv$$

that can be repeated to generate the strings $tuuv$, $tu\dots uv$ and tv . \square

This property is next exploited to demonstrate that a language is not regular.

Example 2.78 (Language with two equal powers) Consider the familiar context-free language:

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

and assume by contradiction it is regular. Take a sentence $x = b^k e^k$, with k large enough, and decompose it into three substrings, $x = tuv$, with u not empty. Depending on the positions of the two divisions, the strings t, u , and v are as in the following scheme:

$$\begin{array}{c} \overbrace{b \dots b}^t \overbrace{b \dots b}^u \overbrace{b \dots e}^v \\ \overbrace{b \dots b}^t \overbrace{b \dots e}^u \overbrace{e \dots e}^v \\ \overbrace{b \dots e}^t \overbrace{e \dots e}^u \overbrace{e \dots e}^v \end{array}$$

Pumping the middle string will lead to contradiction in all cases. For row one, if u is repeated twice, the number of b exceeds the number of e , causing the pumped string not to be in the language. For row two, repeating twice u , the string $tuuv$ contains a pair of substrings be and does not conform to the language structure. Finally for row three, repeating u , the number of e exceeds the number of b . In all cases the pumped strings are not valid and Property 2.77 is contradicted. This completes the proof that the language is not regular.

This example and the known inclusion of the families $REG \subseteq CF$ justify the following statement.

Property 2.79 Every regular language is context-free and there exist context-free languages which are not regular.

The reader should be convinced by this example that the regular family is too narrow for modeling some typical simple constructs of technical languages. Yet it would be foolish to discard regular expressions, because they are perfectly fit for modeling some most common parts of technical languages: on one hand there are the substrings that make the so-called lexicon (for instance, numerical constants and identifiers), on the other hand, many constructs that are variations over the list paradigm (e.g., lists of procedure parameters or of instructions).

Role of Self-nested Derivations Having ascertained that regular languages are a smaller family than context-free ones, it is interesting to focus on what makes some typical languages (as the two powers language, Dyck or palindromes) not regular. Careful observation reveals that their grammars have a common feature: they all use some recursive derivation which is neither left nor right, but is called *self-nested*:

$$A \stackrel{+}{\Rightarrow} uAv \quad u \neq \varepsilon \text{ and } v \neq \varepsilon$$

On the contrary, such derivations cannot be obtained with unilinear grammars which permit only unilateral recursions.

Now, it is the absence of self-nesting recursion that permitted us to solve linear equations by Arden identity. The higher generative capacity of context-free grammars essentially comes from such derivations, as next stated.

Property 2.80 Any context-free grammar not producing self-nesting derivations generates a regular language.

Example 2.81 (Not self-nesting grammar) The grammar G :

$$S \rightarrow AS \mid bA \quad A \rightarrow aA \mid \varepsilon$$

though not unilinear, does not permit self-nested derivations. Therefore $L(G)$ is regular, as we can see by solving the language equations.

$$\begin{cases} L_S = L_A L_S \cup bL_A \\ L_A = aL_A \cup \varepsilon \end{cases}$$

$$\begin{cases} L_S = L_A L_S \cup bL_A \\ L_A = a^* \end{cases}$$

$$L_S = a^* L_S \cup ba^*$$

$$L_S = (a^*)^* ba^*$$

Context-Free Languages of Unary Alphabet The converse of Property 2.80 is not true in general: in some cases self-nesting derivations do not cause the language to be non-regular. On the way to illustrate this fact, we take the opportunity to mention a curious property of context-free languages having a one-letter alphabet.

Property 2.82 Every language defined by a context-free grammar over a one-letter (or unary) alphabet, $|\Sigma| = 1$, is regular.

Observe that the sentences x with unary alphabet are in bijective correspondence with integer numbers, via the mapping $x \Leftrightarrow n$, if and only if $|x| = n$.

Example 2.83 The grammar:

$$G = \{S \rightarrow aSa \mid \varepsilon\}$$

has the self-nesting derivation $S \Rightarrow aSa$, but $L(G) = (aa)^*$ is regular. A right-linear, equivalent grammar is easily obtained, by shifting to suffix the nonterminal that is, positioned in the middle of the first rule:

$$\{S \rightarrow aaS \mid \varepsilon\}$$

2.7.1 Limits of Context-Free Languages

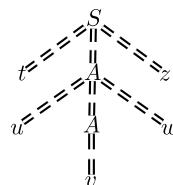
In order to understand what cannot be done with context-free grammars, we study the unavoidable repetitions which are found in longer sentences of such languages, much as we did for regular languages. We shall see that longer sentences necessarily contain two substrings, which can be repeated the same unbounded number of times by applying a self-nested derivation. This property will be exploited to prove that context-free grammars cannot generate certain languages where three or more parts are repeated the same number of times.

Property 2.84 (Language with three equal powers) The language

$$L = \{a^n b^n c^n \mid n \geq 1\}$$

is not context-free.

Proof By contradiction, assume grammar G of L exists and imagine the syntax tree of sentence $x = a^n b^n c^n$. Focus now on the paths from the root (axiom S) to the leaves. At least one path must have a length that increases with the length of sentence x , and since n can be arbitrarily large, such path necessarily traverses two nodes with identical nonterminal label, say A . The situation is depicted in the scheme:



where t, u, v, w, z are terminal strings. This scheme denotes the derivation

$$S \xrightarrow{+} t A z \xrightarrow{+} t u A w z \xrightarrow{+} t u v w z$$

This contains a recursive subderivation from A to A , which can be repeated any number j of times, producing strings of type

$$y = t \overbrace{u \dots u}^j v \overbrace{w \dots w}^j z$$

Now, examine all possible cases for strings u, w :

- Both strings contain just one and the same character, say a ; therefore, as j increases, string y will have more a than b , hence cannot be in the language.
- String u contains two or more different characters, for instance $u = \dots a \dots b \dots a \dots b \dots$. Then, by repeating the recursive part of the derivation, we obtain $uu = \dots a \dots b \dots a \dots b \dots$, where characters a and b are mixed up, hence y is not in the language. We do not discuss the analogous case when string w contains two different characters.

- String u contains only one character, say a , and string w only one different character, say b . When j increases, string y contains a number of a greater than the number of c , hence it is not valid. \square

This reasoning exploits the possibility of pumping the sentences by repeating a recursive derivation. It is a useful conceptual tool for proving that certain languages are not in the *CF* family.

Although the language with three equal powers has no 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 the *replica*, to be found in many technical contexts, whenever two lists contain elements that must be identical or more generally must agree with each other. A concrete case is provided by procedure declaration/invocation: the correspondence between the formal parameter list and the actual parameter list. An example inspired by English is: cats, vipers, crickets, and lions are respectively mammals, snakes, insects, and mammals.

In the most abstract form the two lists are made with the same alphabet and the replica is the language

$$L_{\text{replica}} = \{uu \mid u \in \Sigma^+\}$$

Let $\Sigma = \{a, b\}$. A sentence $x = abbbabbb = uu$ is in some respect analogous to a palindrome $y = abbbbbba = uu^R$, but string u is copied in the former language, specularly reversed in the latter. We may say the symmetry of sentences L_{replica} is translational, not specular. Strange enough, whereas palindromes are a most simple context-free language, the language of replicas is not context-free. This comes from the fact that the two forms of symmetry require quite different control mechanisms: a LIFO (last in first out) push-down stack for specular, and a FIFO (first in first out) queue for translational symmetry. We shall see in Chap. 4 that the algorithms (or automata) recognizing 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 to render it similar to the three equal powers language.

We focus on the following subset of L_{replica} , obtained by means of intersection with a regular language

$$L_{abab} = \{a^m b^n a^m b^n \mid m, n \geq 1\} = L_{\text{replica}} \cap a^+ b^+ a^+ b^+$$

We state (anticipating the proof on p. 153) 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 context-free, we may conclude the same for L_{replica} .

For brevity, we omit the analysis of the possible cases of the strings to be pumped, since it closely resembles the discussion in the previous proof (p. 76).

Table 2.8 Closure properties of REG and CF

Reflection	Star	Union or concatenation	Complement	Intersection
$R^R \in REG$	$R^* \in REG$	$R_1 \oplus R_2 \in REG$	$\neg R \in REG$	$R_1 \cap R_2 \in REG$
$L^R \in CF$	$L^* \in CF$	$L_1 \oplus L_2 \in CF$	$\neg L \notin CF$	$L_1 \cap L_2 \notin CF$ $L \cap R \in CF$

2.7.2 Closure Properties of REG and CF

We know language operations are used to combine languages into new ones with the aim of extending, filtering, or modifying a given language. But not all operations preserve the class or family the given languages belong to. When the result of the operation exits from the starting family, it can no longer be generated with the same type of grammar.

Continuing the comparison between regular and context-free languages, we resume in Table 2.8 the closure properties with respect to language operations: some are already known, others are immediate, and a few need to await the results of automata theory for their proofs. We denote by L and R a generic context-free language and regular language, respectively.

Comments and examples follow.

- A nonmembership (such as $\neg L \notin CF$) means that the left term does not always belong to the family; but this does not exclude, for instance, that the complement of some context-free language is context-free.
- The mirror language of $L(G)$ is generated by the *mirror grammar*, the one obtained reversing the right parts of the rules. Clearly, if grammar G is right-linear the mirror grammar is left-linear and defines a regular language.
- We know the star, union, and concatenation of context-free languages are context-free. Let G_1 and G_2 be the grammars of L_1 and L_2 , let S_1 and S_2 be their axioms, and suppose that the nonterminal sets are disjoint, $V_1 \cap V_2 = \emptyset$. To obtain the new grammar in the three cases, add to the rules of G_1 and G_2 the following initial rules:

$$\begin{array}{ll} \text{Star:} & S \rightarrow SS_1 \mid \epsilon \\ \text{Union:} & S \rightarrow S_1 \mid S_2 \\ \text{Concatenation:} & S \rightarrow S_1S_2 \end{array}$$

In the case of union, if the grammars are right-linear, so is the new grammar. On the contrary, the new rules introduced for concatenation and star are not right-linear but we know that an equivalent right-linear grammar for the resulting languages exists, because they are regular (Property 2.23, p. 23) and family REG is closed under such operations.

- The proof that the complement of a regular language is regular is in Chap. 3 (p. 132).

- The intersection of two context-free languages is not context-free (in general), as witnessed by the known language with three equal powers (Example 2.84 on p. 76):

$$\{a^n b^n c^n \mid n \geq 1\} = \{a^n b^n c^+ \mid n \geq 1\} \cap \{a^+ b^n c^n \mid n \geq 1\}$$

where the two components are easily defined by context-free grammars.

- As a consequence of De Morgan identity, the complement of a context-free language is not context-free (in general): since $L_1 \cap L_2 = \neg(\neg L_1 \cup \neg L_2)$, 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 and a regular language is context-free. The proof will be given on p. 153.

The last property can be applied, in order to make a grammar more discriminatory, by filtering the language with a regular language which forces some constraints on the original sentences.

Example 2.85 (Regular filters on Dyck language (p. 42)) It is instructive to see how the freely parenthesized sentences of a Dyck language L_D of alphabet $\Sigma = \{a, a'\}$ can be filtered, by intersecting with the regular languages:

$$L_1 = L_D \cap \neg(\Sigma^* a' a \Sigma^*) = (aa')^*$$

$$L_2 = L_D \cap \neg(\Sigma^* a' a \Sigma^*) = \{a^n (a')^n \mid n \geq 0\}$$

The first intersection preserves the sentences that do not contain substring $a' a'$, i.e., it eliminates all the sentences with nested parentheses. The second filter preserves the sentences having exactly one nest of parentheses. Both results are context-free languages, but the former is also regular.

2.7.3 Alphabetic Transformations

It is a common experience to find conceptually similar languages that differ by the concrete syntax, i.e., by the choice of terminals. For instance, multiplication may be represented by sign \times , by an asterisk, or by a dot in different languages. The term *transliteration* or *alphabetic homomorphism* refers to the linguistic operation that replaces individual characters by other ones.

Definition 2.86 (Transliteration (or alphabetic homomorphism³⁰)) Consider two alphabets: *source* Σ and *target* Δ . An alphabetic transliteration is a function:

$$h : \Sigma \rightarrow \Delta \cup \{\varepsilon\}$$

³⁰This is a simple case of the translation functions to be studied in Chap. 5.

The transliteration or image of character $c \in \Sigma$ is $h(c)$, an element of the target alphabet. If $h(c) = \varepsilon$, character c is erased. A transliteration is *nonerasing* if, for no source character c , it is $h(c) = \varepsilon$.

The image of a source string $a_1a_2\dots a_n, a_i \in \Sigma$ is the string $h(a_1)h(a_2)\dots h(a_n)$ obtained concatenating the images of individual characters. Notice the image of the empty string is itself.

Such transformation is compositional: the image of the concatenation of two strings v and w is the concatenation of the images of the strings:

$$h(v.w) = h(v).h(w)$$

Example 2.87 (Printer) An obsolete printer cannot print Greek characters and instead prints the special character \square . Moreover, the test sent to the printer may contain control characters (such as `start-text`, `end-text`) that are not printed. The text transformation (disregarding uppercase letters) is described by the transliteration:

$$\begin{aligned} h(c) &= c && \text{if } c \in \{a, b, \dots, z, 0, 1, \dots, 9\}; \\ h(c) &= c && \text{if } c \text{ is a punctuation mark or a blank space;} \\ h(c) &= \square && \text{if } c \in \{\alpha, \beta, \dots, \omega\}; \\ h(\text{start-text}) &= h(\text{end-text}) = \varepsilon. \end{aligned}$$

An example of transliteration is

$$h(\underbrace{\text{start-text}}_{\text{source string}} \underbrace{\text{the const. } \pi \text{ has value 3.14}}_{\text{const. string}} \underbrace{\text{end-text}}_{\text{target string}}) = \underbrace{\text{the const. }}_{\text{target string}} \underbrace{\square}_{\text{target string}} \text{ has value 3.14}$$

An interesting special case of erasing homomorphism is the *projection*: it is a function that erases some source characters and leaves the others unchanged.

Transliteration to Words The preceding qualification of transliteration as alphabetic means the image of a character is a character (or the empty string), not a longer string. Otherwise the transliteration or homomorphism may no longer be qualified as alphabetic. An example is the conversion of an assignment statement $a \leftarrow b + c$ to the form $a := b + c$ by means of a (non-alphabetical) transliteration:

$$h(\leftarrow) = ' := ' \quad h(c) = c \quad \text{for any other } c \in \Sigma$$

This case is also called transliteration *to words*.

Another example: vowels with umlaut of the German alphabet have as image a string of two characters:

$$h(\ddot{a}) = ae, \quad h(\ddot{o}) = oe, \quad h(\ddot{u}) = ue$$

Language Substitution A further generalization leads us to a language transformation termed *substitution* (already introduced in the discussion of linguistic abstraction on p. 26). Now a source character can be replaced by any string of a specified language. Substitution is very useful in early language design phases, in order to leave a construct unspecified in the working grammar. The construct is denoted by a symbol (for instance `(identifier)`). As the project progresses, the symbol will be substituted with the definition of the corresponding language (for instance with $(a \dots z)(a \dots z \mid 0 \dots 9)^*$).

Formally, given a source alphabet $\Sigma = \{a, b, \dots\}$, a substitution h associates each character with a language $h(a) = L_a, h(b) = L_b, \dots$ of target alphabet Δ . Applying substitution h to a source string $a_1a_2 \dots a_n, a_i \in \Sigma$ we obtain a set of strings:

$$h(a_1a_2 \dots a_n) = \{y_1y_2 \dots y_n \mid y_i \in L_{a_i}\}$$

We may say a transliteration to words is a substitution such that each image language contains one string only; if the string has length one or zero, the transliteration is alphabetic.

2.7.3.1 Closure Under Alphabetic Transformation

Let L be a source language, context-free or regular, and h a substitution such that for every source character, its image is a language in the same family as the source language. Then the substitution maps the set of source sentences (i.e., L) on a set of image strings, called the *image* or *target* language, $L' = h(L)$. Is the target language a member of the same family as the source language? The answer is yes, 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 2.88 The family CF is closed under the operation of substitution with languages of the same family (therefore also under the operation of transliteration).

Proof Let G be the grammar of L and h a substitution such that, for every $c \in \Sigma$, L_c is context-free. Let this language be defined by grammar G_c with axiom S_c . Moreover, we assume the nonterminal sets of grammars G, G_a, G_b, \dots are pairwise disjoint (otherwise it suffices to rename the overlapping nonterminals).

Next we construct the grammar G' of language $h(L)$ by transliterating the rules of G with the following mapping f :

$$\begin{aligned} f(c) &= S_c, && \text{for every terminal } c \in \Sigma; \\ f(A) &= A, && \text{for every nonterminal symbol } A \text{ of } G. \end{aligned}$$

The rules of grammar G' are constructed next:

- to every rule $A \rightarrow \alpha$ of G apply transliteration f , to the effect of replacing each terminal character with the axiom of the corresponding target grammar;
- add the rules of grammars G_a, G_b, \dots

It should be clear that the new grammar generates language $h(L(G))$. □

In the simple case where the substitution h is a transliteration, the construction of grammar G' is more direct: replace in G any terminal character $c \in \Sigma$ with its image $h(c)$.

For regular languages an analogous result holds.

Property 2.89 The family REG is closed under substitution (therefore also under transliteration) with regular languages.

Essentially the same construction of the proof of Property 2.88 can be applied to the r.e. of the source language, to compute the r.e. of the target language.

Example 2.90 (Grammar transliterated) The source language $i(; i)^*$, defined by rules:

$$S \rightarrow i; S \mid i$$

schematizes a program including a list of instructions i separated by semicolon. Imagine that instructions have to be now defined as assignments. Then the following transliteration to words is appropriate:

$$g(i) = v \leftarrow e$$

where v is a variable and e an arithmetic expression. This produces the grammar:

$$S \rightarrow A; S \mid A \quad A \rightarrow v \leftarrow e$$

As next refinement, the definition of expressions can be plugged in by means of a substitution $h(e) = L_E$, where the image language is the well-known one. Suppose it is defined by a grammar with axiom E . The grammar of the language after expression expansion is

$$S \rightarrow A; S \mid A \quad A \rightarrow v \leftarrow E \quad E \rightarrow \dots \quad \text{--- usual rules for arith. expr.}$$

As a last refinement, 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 an r.e. 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 recursive rules are indispensable for parenthesis structures, the idea arises to combine r.e. and grammar rules in a notation, called *extended context-free grammar*, or *EBNF*³¹ that takes the best of each formalism: simply

³¹Extended BNF.

enough we allow the right part of a rule to be an r.e. Such grammars have a nice graphical representation, the syntax charts to be shown in Chap. 4 on p. 169, which represents the blueprint of the flowchart of the syntax analyzer.

First observe that, since family CF is closed by all the operations of r.e., the family of languages defined by EBNF grammars coincide with family CF .

In order to appreciate the clarity of extended rules with respect to basic ones, we examine a few typical constructs of programming languages.

Example 2.91 (EBNF grammar of a block language: declarations) Consider a list of variable declarations:

char text1, text2; *real* temp, result; *int* alpha, beta2, gamma;

to be found with syntactic variations in most programming languages.

The alphabet is $\Sigma = \{c, i, r, v, ', ',', ;\}$, where c, i, r stand for *char*, *int*, *real* and v for a variable name. The language of lists of declarations is defined by r.e. D :

$$((c \mid i \mid r)v(_, v)^*;\)^+$$

The iteration operators used to generate lists are dispensable: remember any regular language can be defined by a grammar, even a unilinear one.

The lists can be defined by the basic grammar:

$$D \rightarrow DE \mid E \quad E \rightarrow AN; \quad A \rightarrow c \mid i \mid r \quad N \rightarrow v, N \mid v$$

with two recursive rules (for D and N). The grammar is a bit longer than the r.e. and subjectively less perspicuous in giving evidence to the two-level hierarchical structure of declarations, which is evident in the r.e. Moreover, the choice of metasymbols A, E, N is to some extent mnemonic but arbitrary and may cause confusion, when several individuals jointly design a grammar.

Definition 2.92 An *extended context-free* or *EBNF grammar* $G = (V, \Sigma, P, S)$ contains exactly $|V|$ rules, each one in the form $A \rightarrow \alpha$, where A is a nonterminal and α is an r.e. of alphabet $V \cup \Sigma$.

For better legibility and concision, other derived operators (cross, power, option) too may be permitted in the r.e.

We continue the preceding example, adding typical block structures to the language.

Example 2.93 (Algol-like language) A block B embraces an optional declarative part D followed by the imperative part I , between the marks b (begin) and e (end):

$$B \rightarrow b[D]Ie$$

The declarative part D is taken from the preceding example:

$$D \rightarrow ((c \mid i \mid r)v(_, v)^*;\)^+$$

The imperative part I is a list of phrases F separated by semicolon:

$$I \rightarrow F(; F)^*$$

Last, a phrase F can be an assignment a or a block B :

$$F \rightarrow a \mid B$$

As an exercise, yet worsening legibility, we eliminate as many nonterminals as possible by applying nonterminal expansion to D :

$$B \rightarrow b[(c \mid i \mid r)v(v, v)^*;]^+ I e$$

A further expansion of I leads to

$$B \rightarrow b((c \mid i \mid r)v(v, v)^*;)^* F(F)^* e$$

Last F can be eliminated obtaining a one-rule grammar G' :

$$B \rightarrow b((c \mid i \mid r)v(v, v)^*;)^* (a \mid B) (; (a \mid B))^* e$$

This cannot be reduced to an r.e. because nonterminal B cannot be eliminated, as it is needed to generate nested blocks ($bb\dots ee$) by a self-nesting derivation (in agreement with Property 2.80, p. 75).

Usually, language reference manuals specify grammars by *EBNF* rules, but beware that 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 shall see in Chap. 5.

2.7.4.1 Derivations and Trees in Extended Context-Free Grammars

The right part α of an extended rule $A \rightarrow \alpha$ of grammar G is an r.e., which in general derives an infinite set of strings: each one can be viewed as the right part of a nonextended rule having unboundedly many alternatives.

For instance, $A \rightarrow (aB)^+$ stands for a set of rules:

$$A \rightarrow aB \mid aBaB \mid \dots$$

The notion of derivation can be defined for extended grammars too, via the notion of derivation for r.e. introduced on p. 19.

Shortly, for an *EBNF* grammar G consider a rule $A \rightarrow \alpha$, where α is an r.e. possibly containing choice operators (star, cross, union, and option); let α' be a string deriving from α (using the definition of the derivation of an r.e.), not containing any choice operator. For any (possibly empty) strings δ and η there is a one-step derivation:

$$\delta A \eta \xrightarrow{G} \delta \alpha' \eta$$

Then one can define multi-step derivations starting from the axiom and producing terminal strings, and finally the language generated by an *EBNF* grammar, in the same manner as for basic grammars. Exemplification should be enough.

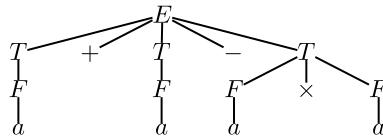
Example 2.94 (Derivation for extended grammar of expressions) The grammar G :

$$E \rightarrow [+ | -]T((+ | -)T)^* \quad T \rightarrow F((\times | /)F)^* \quad F \rightarrow (a | ' (E)')$$

generates arithmetic expressions with the four infix operators, and the prefix operators \pm , parentheses, and terminal a standing for a numeric argument. Square brackets denote option. The left derivation:

$$\begin{aligned} E &\Rightarrow T + T - T \Rightarrow F + T - T \Rightarrow a + T - T \Rightarrow a + F - T \Rightarrow a + a - T \Rightarrow \\ &\Rightarrow a + a - F \times F \Rightarrow a + a - a \times F \Rightarrow a + a - a \times a \end{aligned}$$

produces the syntax tree:



Observe that the degree of a node can be unbounded, with the consequence that the breadth of the tree increases and the height decreases, in comparison with the tree of an equivalent basic grammar.

Ambiguity in Extended Grammars It is obvious that an ambiguous basic grammar remain such also when viewed as an *EBNF* grammar. On the other hand, a different form of ambiguity may arise in an *EBNF* grammar, caused by the ambiguity of the r.e. present in the rules. Recall that an r.e. is ambiguous (p. 21) if it derives a string with two different left derivations.

For instance, the r.e. $a^*b \mid ab^*$, numbered $a_1^*b_2 \mid a_3b_4^*$, is ambiguous because the string ab can be derived as a_1b_2 or as a_3b_4 . As a consequence the extended grammar:

$$S \rightarrow a^*bS \mid ab^*S \mid c$$

is ambiguous.

2.8 More General Grammars and Language Families

We have seen context-free grammars cover the main constructs occurring in technical languages, namely, parentheses structures and hierarchical lists, but fail with other syntactic structures even simple, such as the replica or the three equal powers language on p. 76.

Such shortcomings have motivated, from the early days of language theory, much research on more powerful formal grammars. It is fair to say that none of the formal models have been successful; the more powerful are too obscure and difficult to use, and the models marginally superior to the context-free do not offer significant advantages. In practice, application of such grammars to compilation³² has been episodic and was quickly abandoned. Since the basis of all subsequent developments is the classification of grammars due to the linguist Noam Chomsky, it is appropriate to briefly present it for reference, before moving on to more applied aspects in the coming chapters.

2.8.1 Chomsky Classification

The historical classification of phrase structure grammars based on the form of the rewriting rules is in Table 2.9. Surprisingly enough, very small difference in the rule form, determine substantial changes in the properties of the corresponding family of languages, both in terms of decidability and algorithmic complexity.

A rewriting rule has a left part and a right part, both strings on the terminal alphabet Σ and nonterminal set V . The left part is replaced by the right part. The four types are characterized as follows:

- a rule of *type 0* can replace an arbitrary not empty string over terminals and nonterminals, with another arbitrary string;
- a rule of *type 1* adds a constraint to the form of type 0: 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 we have studied.

For completeness Table 2.9 lists the names of the automata (abstract string recognition algorithms) corresponding to each type, although the notion of automata will not be introduced until next chapter.

The language families are strictly included one into the next from bottom to top, which 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, the others need unbounded memory.

Other properties are worth mentioning, without any claim to completeness. All four language families are closed under union, concatenation, star, reflection, and intersection with a regular language. But for other operators, properties differ: for instance families 1 and 3, but not 0 and 2, are closed under complement.

Concerning 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 grammar. For

³²In computational linguistics some types of grammars more powerful than the context-free have gained acceptance; we mention the best known example, Joshi's Tree Adjoining Grammar (TAG) [9].

Table 2.9 Chomsky classification of grammars and corresponding languages and machines

Grammar	Form of rules	Language family	Type of recognizer
Type 0	$\beta \rightarrow \alpha$ where $\alpha, \beta \in (\Sigma \cup V)^+$ and $\beta \neq \varepsilon$	Recursively enumerable	Turing machine
Type 1 context dependent (or context sensitive)	$\beta \rightarrow \alpha$ where $\alpha, \beta \in (\Sigma \cup V)^+$ and $ \beta \leq \alpha $	Contextual or context-dependent	Turing machine with space complexity limited by the length of the source string
Type 2 context-free or BNF	$A \rightarrow \alpha$ where A is a nonterminal and $\alpha \in (\Sigma \cup V)^*$	Context-free CF or algebraic	Push-down automaton
Type 3 unilinear (right-linear or left-linear)	Right-linear: $A \rightarrow uB$ Left linear: $A \rightarrow Bu$, where A is a nonterminal, $u \in \Sigma^*$ and $B \in (V \cup \varepsilon)$	Regular REG or rational or finite-state	Finite automaton

type 1 grammars the problem is decidable, though its time complexity is not polynomial. Last, only for type 3 the equivalence problem for two grammars is decidable.

We finish with two examples of type 1 grammars and languages.

Example 2.95 (Type 1 grammar of the three equal powers language) The language, proved on p. 76 to be not *CF*, is

$$L = \{a^n b^n c^n \mid n \geq l\}$$

It is generated by the context-sensitive grammar:

1. $S \rightarrow aSBC$
2. $S \rightarrow abC$
3. $CB \rightarrow BC$
4. $bB \rightarrow bb$
5. $bC \rightarrow bc$
6. $cC \rightarrow cc$

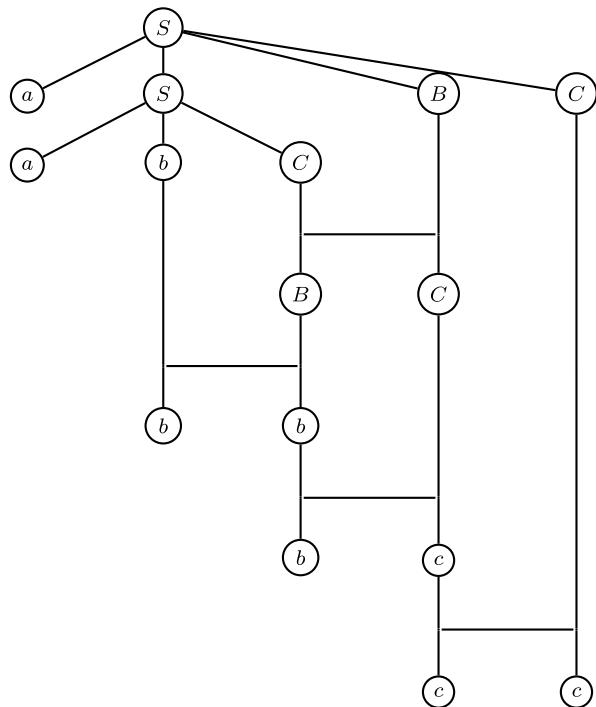
For type 0 and 1 grammars a derivation cannot be represented as a tree, because the left part of a rule typically contains more than one symbol. However, the derivation can be visualized as a graph, where the application of a rule such as $BA \rightarrow AB$ is displayed by a bundle of arcs (hyper-edge) connecting 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 sentences by left derivations. But if we try to generate sentence *aabbcc* proceeding from left to right:

$$S \Rightarrow aSBC \Rightarrow aabCBC \Rightarrow aabcBC \Rightarrow \text{block!}$$

surprisingly, the derivation is stuck, before all nonterminal symbols are eliminated. In order to generate this string we need a not leftmost derivation, shown in Fig. 2.2.

Fig. 2.2 Graph representation of a context-sensitive derivation (Example 2.95)



Intuitively the derivation produces the requested number of 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. In order to produce the valid string, all C 's must be shifted to the right, using rule (3) $CB \rightarrow BC$, until they reach the suffix position.

The first application of (3) reorders the sentential form, to the aim that B becomes adjacent to b . This enables derivation (4) $bB \Rightarrow bb$. Then the derivation continues in the same manner, alternating steps (3) and (4), until the sentential form is left with just C as nonterminal. The occurrences of C are transformed to terminals c , by means of rule $bC \rightarrow bc$, followed by repeated applications of rule $cC \rightarrow cc$.

We stress the finding that the language generated by type 1 grammars may not coincide with the sentences generated by left derivations, unlike for type 2 grammars. This is a cause of difficulty in string recognition algorithms.

Type 1 grammars have the power to generate the language of replicas, or lists with agreement between elements, a construct that we have singled out as practically relevant and exceeding the power of context-free grammars.

Example 2.96 (Type 1 grammar of replica with center) Language $L = \{ycy \mid y \in \{a, b\}^+\}$ contains sentences such as $aabcaab$, where a prefix and a suffix, divided by the central separator c , must be equal.

To simplify the grammar, we assume sentences are terminated to the right by the end-of-text character or *terminator*, \dashv . Grammar:

$$\begin{array}{lllll}
 S \rightarrow X \dashv & XA \rightarrow XA' & A'A \rightarrow AA' & A' \dashv \rightarrow a & B'a \rightarrow ba \\
 X \rightarrow aXA & XB \rightarrow XB' & A'B \rightarrow BA' & B' \dashv \rightarrow b & B'b \rightarrow bb \\
 X \rightarrow bXB & & B'A \rightarrow AB' & A'a \rightarrow aa & Xa \rightarrow ca \\
 & & B'B \rightarrow BB' & A'b \rightarrow ab & Xb \rightarrow cb
 \end{array}$$

To generate a sentence, the grammar follows this strategy: it first generates a palindrome, say $aabXBA$, where X marks the center of the sentence and the second half is in uppercase. Then the second half, modified as $B'AA$, is reflected and converted in several steps to $A'A'B'$. Last, the primed uppercase symbols are rewritten as aab and the center symbol X is converted to c . We illustrate the derivation of sentence $aabcaab$. For legibility, at each step we underline the left part of the rule being applied:

$$\begin{aligned}
 \underline{S} \Rightarrow \underline{X} \dashv \Rightarrow a\underline{XA} \dashv \Rightarrow aa\underline{XAA} \dashv \Rightarrow aab\underline{XBAA} \dashv \Rightarrow \\
 aab\underline{XB'AA} \dashv \Rightarrow aab\underline{XAB'A} \dashv \Rightarrow aab\underline{XA'B'A} \dashv \Rightarrow \\
 aab\underline{XA'AB'} \dashv \Rightarrow aab\underline{XAA'B'} \dashv \Rightarrow aab\underline{XA'A'B'} \dashv \Rightarrow \\
 aab\underline{XA'A'b} \Rightarrow aab\underline{XA'ab} \Rightarrow aab\underline{Xaab} \Rightarrow aabcaab
 \end{aligned}$$

We observe that the same strategy used in generation, if applied in reverse order, would allow to 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 procedure is essentially the computation of a Turing machine that never goes out of the portion of tape containing the original string, but may overprint its symbols.

Such examples should have persuaded the reader of the difficulty to design and apply context-sensitive rules even for very simple languages. The fact is that interaction of grammar rules is hard to understand and control.

In other words, type 1 and 0 grammars can be viewed as a particular notation for writing algorithms. All sorts of problems could be programmed in this way, even mathematical ones, by using the very simple mechanism of string rewriting.³³ It is not surprising that using such an elementary mechanism as only data and control structure makes the algorithm description very entangled.

Undoubtedly the development of language theory towards models of higher computational capacity has mathematical and speculative interests but is almost irrelevant for language engineering and compilation.³⁴

³³An extreme case is a type 1 grammar presented in [12] to generate prime numbers encoded in unary base, i.e., the language $\{a^n \mid n \text{ prime number}\}$.

³⁴For historical honesty, we mention that context-sensitive grammars have been occasionally considered by language and compiler designers. The language Algol 68 has been defined with a special class of type 1 grammars termed 2-level grammars, also known as VW-grammars from Van Wijngarten [14]; see also Cleaveland and Uzgalis [2].

In conclusion, we have to admit that the state of the art of formal language theory does not entirely satisfy the need of 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, coming from general-purpose programming methods. They will be introduced in Chap. 5.

References

1. J. Autebert, J. Berstel, L. Boasson, Context-free languages and pushdown automata, in *Handbook of Formal Languages, vol. 1: Word, Language, Grammar*, ed. by G. Rozenberg, A. Salomaa (Springer, New York, 1997), pp. 111–174
2. J. Cleaveland, R. Uzgalis, *Grammars for Programming Languages* (North-Holland, Amsterdam, 1977)
3. S. Crespi Reghizzi, P. Della Vigna, C. Ghezzi, *Linguaggi Formali e Compilatori* (ISEDI, Milano, 1976)
4. S. Crespi Reghizzi, M. Pradella, Tile rewriting grammars and picture languages. *Theor. Comput. Sci.* **340**(2), 257–272 (2005)
5. F. Gecseg, M. Steinby, Tree languages, in *Handbook of Formal Languages, vol. 3: Beyond Words*, ed. by G. Rozenberg, A. Salomaa (Springer, New York, 1997), pp. 1–68
6. D. Giammarresi, A. Restivo, Two-dimensional languages, in *Handbook of Formal Languages, vol. 3: Beyond Words*, ed. by G. Rozenberg, A. Salomaa (Springer, New York, 1997), pp. 215–267
7. M. Harrison, *Introduction to Formal Language Theory* (Addison-Wesley, Reading, 1978)
8. J. Hopcroft, J. Ullman, *Formal Languages and Their Relation to Automata* (Addison-Wesley, Reading, 1969)
9. D. Jurafsky, J.H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition, and Computational Linguistics* (Prentice-Hall, Englewood Cliffs, 2009)
10. P. Naur, Revised report on the algorithmic language ALGOL 60. *Commun. ACM* **6**, 1–33 (1963)
11. D. Perrin, J.E. Pin, *Infinite Words* (Elsevier, New York, 2004)
12. A. Salomaa, *Formal Languages* (Academic Press, New York, 1973)
13. W. Thomas, Languages, automata, and logic, in *Handbook of Formal Languages, vol. 3: Beyond Words*, ed. by G. Rozenberg, A. Salomaa (Springer, New York, 1997), pp. 389–455
14. A. Van Wijngarten, Report on the algorithmic language ALGOL 68. *Numer. Math.* **22**, 79–218 (1969)

3.1 Introduction

Regular expressions and grammars are widely used in technical specifications of languages but the actual design and implementation of a compiler needs a way to describe the algorithms, termed *recognizers* or *acceptors*, which examine a string and decide if it is a valid sentence of the language. In this chapter we study the recognizers for regular languages and in the next two chapters those of context-free languages.

The need to recognize if 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 words, then verifies syntax; and a graphic user interface must check that the data are correctly entered. Such control is done by a , which can be conveniently specified using minimalist models, termed or . The advantages are that automata do not depend on programming languages and techniques, i.e., on implementation, and that their properties (such as time and memory complexity) are in this way more clearly related with the family of the source language.

In this chapter we briefly introduce more general automata, then focus on those having a finite memory that match the regular language family and have countless applications in computer science and beyond. First we consider the deterministic machines and describe useful methods for cleaning and minimizing. Then we motivate and introduce nondeterministic models, and we show they correspond to the unilinear grammars of the previous chapter. Conversion back to deterministic models follows.

A central section deals with transformations back and forth from regular expressions to automata. On the way to proving their convertibility, we introduce the subfamily of local regular languages, which have a simpler recognition procedure making use of a sliding window.

Then we return to the operations of complement and intersection on regular languages from the standpoint of their recognizers, and present the composition of au-

tomata by Cartesian product. The chapter ends with a summary of the interrelation between finite automata, regular expressions, and grammars.

In compilation finite automata have several uses to be described especially in Chap. 4: in lexical analysis to extract the shortest meaningful strings from a text, for making simple translations, and for modeling and analyzing program properties in static flow analysis and optimization.

3.2 Recognition Algorithms and Automata

To check if a string is valid for a specified language, we need a , a type of algorithm producing a answer, commonly referred to in computational complexity studies as a . For instance, a famous problem is to decide if 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 of alphabet Σ . The application of a recognition algorithm α to a given string x is denoted as $\alpha(x)$. We say string x is or if $\alpha(x) = \text{yes}$, otherwise it is . The language recognized, $L(\alpha)$, is the set of accepted strings:

$$L(\alpha) = \{x \in \Sigma^* \mid \alpha(x) = \text{yes}\}$$

The 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 not defined, hence x is not a valid sentence of language $L(\alpha)$. In such case we say that the membership problem for L is , or also that L is recursively enumerable.

In principle, if for an artificial language the membership problem is semidecidable, we cannot exclude that the compiler falls into an endless loop, for some input strings. In practice we do not have to worry about such decidability issues, central as they are for the theory of computation,¹ because in language processing the only language families of concern are decidable, and efficiently so.

Incidentally, even within the family of context-free languages, some decision problems, different from string membership, are not decidable: we have mentioned in Chap. 2 the problem of deciding if two grammars are weakly equivalent, and 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 shall see the translation from a language to another: clearly the codomain of a translation algorithm is much more complex than a pure yes/no, since it is itself a set of strings, the target language.

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

¹Many books cover the subject, e.g., Bovet and Crescenzi [3], Floyd and Beigel [4], Hopcroft and Ullman [7], Kozen [8], Mandrioli and Ghezzi [9], and McNaughton [10]).

consider time rather than space complexity. With rare exceptions, all the problems of interest for compilation have 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 an important attribute for portable programmed devices.

A tenet of the theory of computation is that the complexity of an algorithm should be measured by the number of steps, rather than by 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 computational steps: 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 premature commitment and 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.

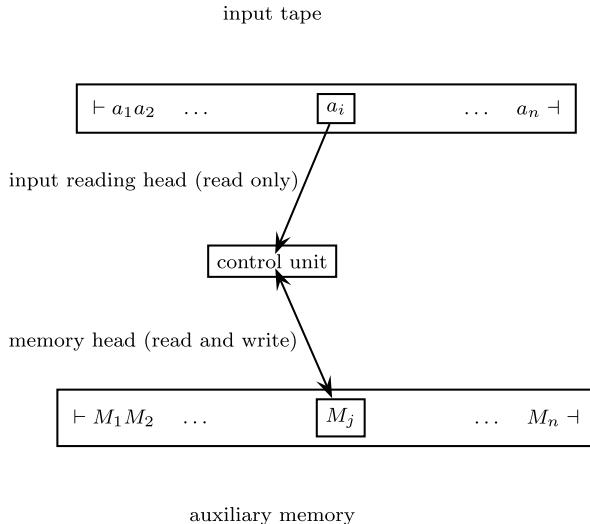
3.2.1 A General Automaton

An automaton or abstract machine is an ideal computer featuring a very small set of simple instructions. Starting from the Turing machine of the 1930s, research on abstract machines has spawned many models, but only a few are important for language processing.² In its more general form a recognizer is schematized in Fig. 3.1. It comprises three parts: input tape, control unit, and (auxiliary) memory.

The control unit has a limited store, to be represented as a finite set of states; the auxiliary memory, on the other hand, has unbounded capacity. The upper tape contains the given input or *source* string, which can be read but not changed. Each case of the tape contains a terminal character; the cases to the left and right of the input contain two delimiters, the start of text mark \vdash and the end of text mark or terminator \dashv . A peculiarity of automata is that the auxiliary memory is also a tape (instead of the random access memory used in other computational models), containing symbols of another alphabet.

The automaton performs at discrete time instants the following actions: read the *current character* a_i from input, shift the reading head, read from memory the *current symbol* M_j and replace it with another symbol, move the memory head, and change the *current state* of the control unit to the *next one*.

²Other books have a broader and deeper coverage of automata theory, such as Salomaa [14], Hopcroft and Ullman [6, 7], Harrison [5], and the handbook [12]; for finite automata a specific reference is Sakarovitch [13].

Fig. 3.1 General automaton

The automaton examines 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 some of the following effects:

- shift the input head left or right by one position;
- overwrite the current memory symbol with another one, and shift the memory head left or right by one position;
- change 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 text processing by a single scan. For 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*: the suffix of the input string still to be read, that is, laying to the right of the head; the contents of the memory tape and the position of the head; the state of the control unit. The *initial configuration* has: the input head positioned on character a_1 (i.e., right of the start of text mark), the control unit in an initial state, and the memory containing a specific 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 situation may explore alternative paths.

A *configuration* is *final* if the control is in a state specified as final, and the input head is on the terminator. Sometimes, instead of, 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 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 leading to a final configuration (a nondeterministic automaton can possibly reach a final configuration by different computations). The *language accepted or recognized* by the machine is the set of accepted strings.

Notice a computation terminates either when the machine has entered a final configuration or when in the current configuration no move can be applied. In the latter case the source string is not accepted by that computation (but perhaps accepted by another computation if the machine is indeterministic).

Two automata accepting the same language are called *equivalent*. Of course two machines, though equivalent, may belong to different models or have different computational complexities.

Turing Machine The preceding description substantially reproduces the automaton 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 accepting it and halting for every input string. The family of decidable languages is smaller than the one of recursively enumerable languages.

Such machine is the recognizer of two of the language families of Chomsky classification (p. 86). The languages generated by type 0 grammars are exactly the recursively enumerable. 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 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 are a significant term of comparison for the efficient machine models used to recognize and transform technical languages. Such models can be viewed as Turing machines with severe limitation on memory. 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 LIFO (last in first out) store, the machine is termed a pushdown automaton and is the recognizer of context-free languages.

The memory limitations have a profound effect on the properties and in particular the performance of the 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 the space bounded Turing machine may take a non-polynomial time to recognize a context-sensitive language, another reason making it unpractical to use. Context-free language recognizers are somehow 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 is able to support innumerable applications in diverse areas, from digital circuit design to system modeling and communication protocols, to mention just a few. Our presentation will be focused on the aspects that are important for language and compiler design, but in order to make the presentation self-contained we briefly introduce some essential definitions and theoretical results.

Conforming to the general scheme, a finite automaton comprises: the input tape with the source string $x \in \Sigma^*$; the control unit; the reading head, initially placed on the first character of x , scanning the string until its end, unless an error occurs before. Upon reading a character, the automaton updates the state of the control unit and advances the reading head. Upon reading the last character, the automaton accepts x if and only if the state is an accepting one.

A well-known representation of an automaton is by a *state-transition diagram* or *graph*. This is a directed graph whose nodes are the states of the control unit. Each arc is labeled with a terminal and represents the change of state or *transition* caused by reading the terminal.

Example 3.1 (Decimal constants) The set L of decimal constants has the alphabet $\Sigma = \Delta \cup \{0, \bullet\}$ where $\Delta = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ is a digit other than zero. The r.e. is

$$L = (0 \cup \Delta(0 \cup \Delta)^*) \bullet (0 \cup \Delta)^+$$

The recognizer is specified by the *state-transition diagram* or by the equivalent *state-transition table* in Fig. 3.2.

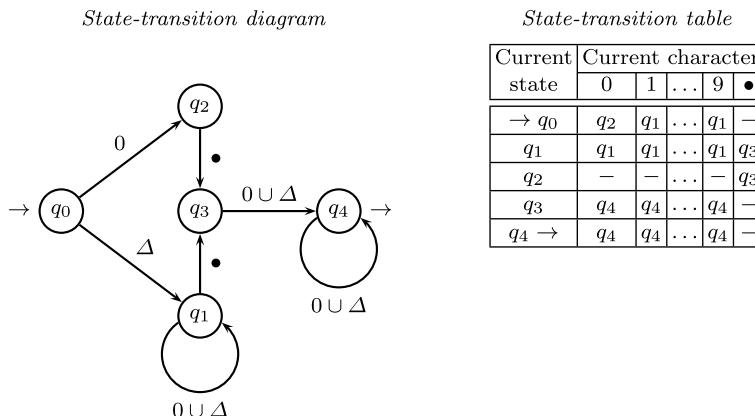


Fig. 3.2 State-transition diagram or graph (left) and table (right) for decimal constants (Example 3.1)

Notice that if several arcs span the same two nodes, only one is drawn: e.g., arc $(q_0 \rightarrow q_1)$ represents a bundle of 9 arcs with labels 1, 2, ..., 9. The initial state q_0 is marked by an ingoing arrow; the final state q_4 by an outgoing arrow. The transition table is the incidence matrix of the graph: for each pair *(current state, current character)* the entry contains the next state. Notice the arrows giving evidence to initial/final states.

Given the source string $0\bullet 2\bullet$, the automaton transits through the states q_0, q_2, q_3, q_4 . In the last state, since no transition allows reading a \bullet character, the computation stops before the entire input has been consumed. As a consequence, the source string is not accepted. On the other hand, input $0\bullet 21$ would be accepted.

If we prefer to modify the language so that constants such as $305\bullet$ ending with a decimal point are legal, the state q_3 too must be marked as final.

We have seen an automaton may have several 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.

A *finite deterministic automaton* M comprises five items:

1. Q , the *state set* (finite and not empty)
2. Σ , the *input or terminal alphabet*
3. the *transition function* $\delta : (Q \times \Sigma) \rightarrow Q$
4. $q_0 \in Q$, the *initial state*
5. $F \subseteq Q$, the *set of final states*.

Function δ specifies the moves: the meaning of $\delta(q, a) = r$ is that machine M in the current state q reads a and moves to next state r . If $\delta(q, a)$ is undefined, the automaton stops and we can 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 symbol a , as $q \xrightarrow{a} r$, which intuitively recalls the corresponding arc on the state-transition diagram.

The automaton processes a not empty string x by a series of moves. Take $x = ab$; reading the first character, the first step $\delta(q_0, a) = q_1$ leads to state q_1 , then to q_2 by the second step $\delta(q_1, b) = q_2$.

In short, instead of writing $\delta(\delta(q_0, a), b) = q_2$, we combine the two steps into $\delta(q_0, ab) = q_2$, to say that reading string ab the machine moves to state q_2 . Notice that the second argument of function delta is now a string.

A special case is the empty string, for which we assume no change of state:

$$\forall q \in Q : \delta(q, \varepsilon) = q$$

Following these stipulations, the domain of the transition function is $(Q \times \Sigma^*)$ and the definition is

$$\delta(q, ya) = \delta(\delta(q, y), a), \quad \text{where } a \in \Sigma \text{ and } y \in \Sigma^*$$

Looking at the graph, if $\delta(q, y) = q'$, then, and only then, there exists a path from node q to node q' , such that the concatenated labels of the arrows make string y . We say y is the *label* of the path and the path represents a *computation*.

A string x is *recognized* or accepted by the automaton if it is the label of a path from the initial state to a final state, $\delta(q_0, x) \in F$.

Notice the empty string is recognized if, and only if, the initial state is also final, $q_0 \in F$.

The *language recognized* or accepted by automaton M is

$$L(M) = \{x \in \Sigma^* \mid x \text{ is recognized by } M\}$$

The languages accepted by such automata are termed *finite-state recognizable*.

Two automata are *equivalent* if they accept the same language.

Example 3.2 (Example 3.1 continued) The automaton M of p. 96 is defined by

$$\begin{aligned} Q &= \{q_0, q_1, q_2, q_3, q_4\} \\ \Sigma &= \{1, 2, 3, 4, 5, 6, 7, 8, 9, 0, \bullet\} \\ q_0 &= q_0 \\ F &= \{q_4\} \end{aligned}$$

Examples of transitions:

$$\delta(q_0, 3 \bullet 1) = \delta(\delta(q_0, 3 \bullet), 1) = \delta(\delta(\delta(q_0, 3), \bullet), 1) = \delta(\delta(q_1, \bullet), 1) = \delta(q_3, 1) = q_4$$

Since $q_4 \in F$, string $3 \bullet 1$ is accepted. On the contrary, since $\delta(q_0, 3 \bullet) = q_3$ is not final, string $3 \bullet$ is rejected, as well as string 02 because function $\delta(q_0, 02) = \delta(\delta(q_0, 0), 2) = \delta(q_2, 2)$ is undefined.

Observing that for each input character the automaton executes one step, the total number of steps is exactly equal to the length of the input string. Therefore such machines are very efficient as they can recognize strings in real time by a single left-to-right scan.

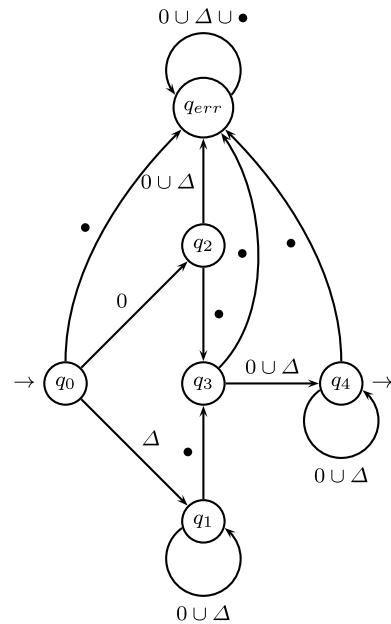
3.4.1 Error State and Total Automata

If the move is not defined in state q when reading character a , we say that the automaton falls into the *error state* q_{err} . The error state is such that for any character the automaton remains in it, thus justifying its other name of *sink* or *trap state*. Obviously the error state is not final.

The state-transition function can be made *total* by adding the error state and the transitions from/to it:

$$\begin{array}{ll} \forall \text{ state } q \in Q \text{ and } \forall a \in \Sigma, \text{ if } \delta(q, a) \text{ is undefined set } \delta(q, a) = q_{\text{err}} & \\ \forall a \in \Sigma & \text{set } \delta(q_{\text{err}}, a) = q_{\text{err}} \end{array}$$

Fig. 3.3 Recognizer of decimal constants completed with sink (error or trap) state



The recognizer of decimal constants, completed with error state, is shown in Fig. 3.3.

Clearly any computation reaching the error state gets trapped in it and cannot reach a final state. As a consequence, the total automaton accepts the same language as the original one. It is customary to leave the error state implicit, neither drawing a node nor specifying the transitions for it.

3.4.2 Clean Automata

An automaton may contain useless parts not contributing to any accepting computation, which are best eliminated. Notice the following concepts hold as well for nondeterministic finite automata, and in general for any kind of abstract machine.

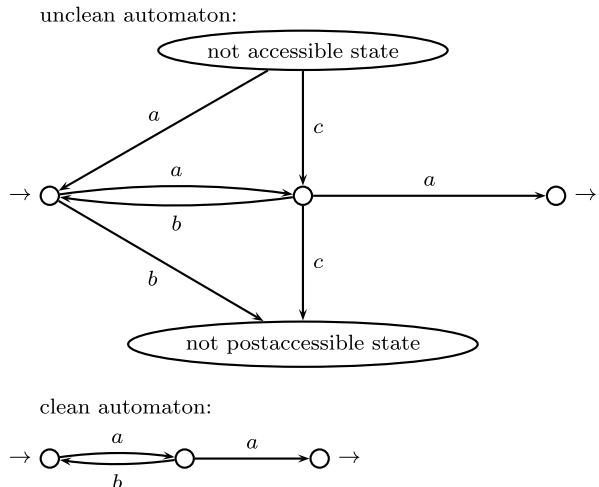
A state q is *reachable* from state p if a computation exists going from p to q . A state is *accessible* if it can be reached from the initial state; it is *postaccessible* if a final state can be reached from it. A state is called *useful* if it is accessible and postaccessible; otherwise it is *useless*. 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 3.3 For every finite automaton there exists an equivalent clean automaton.

The construction of the clean machine consists of identifying and deleting useless states, together with adjoining transitions.

Fig. 3.4 Useless states (top)
and their elimination
obtaining a clean automaton
(bottom)



Example 3.4 (Elimination of useless states) Figure 3.4 shows a machine with useless states and the corresponding clean machine.

Notice the error state is never postaccessible, hence always useless.

3.4.3 Minimal Automata

We focus on properties of the automata recognizing the same language. Their state sets and transition functions are in general different but their computations are equivalent. A central property is that, among such equivalent machines, there exists only one which is the smallest in the following sense.

Property 3.5 For every finite-state language, the (deterministic) finite recognizer minimal with respect to the number of states is unique (apart from renaming of states).

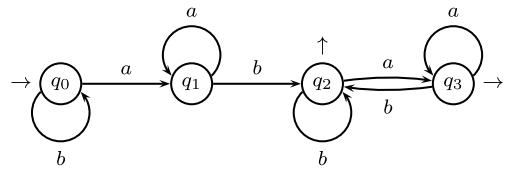
Conceptually the statement is fundamental as it permits to represent a collection of equivalent machines by a standard one which, moreover, is minimal. In practice this is not so important for compiler applications, nevertheless we describe for self-consistency the standard minimization algorithm.

First we need to introduce an algebraic relation between equivalent states, to be computed by the algorithm.³

Although we assume the given automaton is clean, some of its states may be redundant, in the sense that they can be fused together at no consequence for strings accepted or rejected. Any two such states are termed *indistinguishable* from each

³Other more efficient and subtle algorithms have been invented. We refer the reader to the survey in [16].

Fig. 3.5 Automaton M to be minimized (Example 3.7)



other, and the corresponding binary relation is termed *indistinguishability* (or of Nerode type).

Definition 3.6 The states p and q are *indistinguishable* if, and only if, for every string $x \in \Sigma^*$, either both 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 indistinguishable if, starting from them and scanning the same arbitrarily chosen input string x , it never happens that a computation reaches a final state and the other does not.

Notice that:

1. the sink state q_{err} is distinguishable from every state p , since for any state there exists a string x such that $\delta(p, x) \in F$, while for every string x it is $\delta(q_{\text{err}}, x) = q_{\text{err}}$;
2. p and q are distinguishable if p is final and q is not, because $\delta(p, \varepsilon) \in F$ and $\delta(q, \varepsilon) \notin F$;
3. p and q are distinguishable if, for some character a , the next states $\delta(p, a)$ and $\delta(q, a)$ are distinguishable.

In particular, p is distinguishable from q if the set of labels attached to the outgoing arrows from p and the similar set from q are different: in that case there exists a character a such that the move from state p reaches state p' , while the move from q is not defined (i.e., it reaches the sink); from condition (3), the two states are distinguishable.

Indistinguishability as a relation is symmetric, reflexive, and transitive, i.e., it is an equivalence relation, whose classes are computed by a straightforward procedure to be described next by means of an example.

Example 3.7 (Equivalence classes of indistinguishable states) For automaton M of Fig. 3.5, we construct a table of size $|Q| \times |Q|$, representing the indistinguishability relation; but, since the relation is symmetric, we do not have to fill the cases on and above the principal diagonal.

The procedure will mark with X the case at position (p, q) , when it discovers that the states p and q are distinguishable. We initialize with X the cases such that one state only is final:

	q_1	-	-
q_1		X	X
q_2	X	X	-
q_3	X	X	
	q_0	q_1	q_2

Then we examine every case (p, q) still unmarked and, for every character a , we consider the next states $r = \delta(p, a)$ and $s = \delta(q, a)$. If case (r, s) already contains X , meaning that r is distinguishable from s , then also p and q are so, therefore case (p, q) is marked X . Otherwise, if (r, s) is not marked X , the 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 to the left. The result of this step is the table to the right. Notice that case $(1, 0)$ is marked because the pair $(0, 2) \equiv (2, 0)$ was already marked:

q_1	$(1, 1), (0, 2)$	$-$	$-$	q_1	X	$-$	$-$
q_2	X	X	$-$	q_2	X	X	$-$
q_3	X	X	$(3, 3), (2, 2)$	q_3	X	X	$(3, 3), (2, 2)$
	q_0	q_1	q_2	q_0	q_1		q_2

Now all cases (under the principal diagonal) are filled and the algorithm terminates. The cases not marked with X identify the indistinguishable pairs, in our example only the pair (q_2, q_3) .

An equivalence class contains all the pairwise indistinguishable states. In the example the equivalence classes are $[q_0], [q_1], [q_2, q_3]$.

It is worthwhile analyzing another example where the transition function is not total. To this end, we modify automaton M of Fig. 3.5, erasing the self-loop $\delta(q_3, a) = q_3$, i.e., redefining the function as $\delta(q_3, a) = q_{\text{err}}$. As a consequence, states q_2 and q_3 become distinguishable, because $\delta(q_2, a) = q_3$ and $\delta(q_3, a) = q_{\text{err}}$. Now every equivalence class is a singleton, meaning the automaton is already minimal.

3.4.3.1 Construction of Minimal Automaton

The minimal automaton M' , equivalent to the given M , has for states the equivalence classes of the indistinguishability relation. It is simple to construct its transition function: M' contains the arc:

$$\overbrace{[\dots, p_r, \dots]}^{C_1} \xrightarrow{b} \overbrace{[\dots, q_s, \dots]}^{C_2}$$

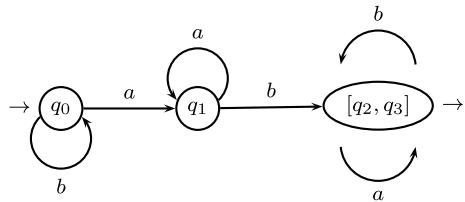
between equivalence classes C_1 and C_2 , if, and only if, M contains an arc:

$$p_r \xrightarrow{b} q_s$$

between two states, respectively, belonging to the two classes. Notice that the same arc of M' may derive from several arcs of M .

Example 3.8 (Example 3.7 continued) The minimal automaton M' is in Fig. 3.6. This has the least number of states of all equivalent machines. In fact, we could easily check that merging any two states, the resulting machine would not be equivalent, but it would accept a larger language than the original.

Fig. 3.6 Result M' of minimization of automaton M of Fig. 3.5



The example has constructively illustrated Property 3.5: that the minimal automaton is unique. From this it is a straightforward test to check whether two given machines are equivalent. First minimize both machines; then compare their state-transition graphs to see if they are identical (i.e., isomorphic and with corresponding initial and final states), apart from a change of name of the states.⁴

In practical use, obvious economy reasons 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 of the recognizer should be avoided: when the automaton is enriched with actions computing an output function, to be seen in Chap. 5, two states that are indistinguishable for the recognizer, can be associated with different output actions. Then merging the states would spoil the intended translation.

Finally we anticipate that the uniqueness property of the minimal automaton does not hold for the nondeterministic machines to be introduced next.

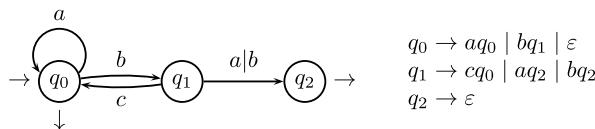
3.4.4 From Automata to Grammars

Without much effort we are going to realize that finite automata and unilinear (or type 3) grammars of p. 69 are alternative but equivalent notations defining the same language family.

First we show how to construct a right-linear grammar equivalent to an automaton. Grammar G has as nonterminal set the states Q of the automaton, and the axiom is the initial state. For each move $q \xrightarrow{a} r$ the grammar has the rule $q \rightarrow ar$. If state q is final, it has also the terminal rule $q \rightarrow \varepsilon$.

It is evident that there exists a bijective correspondence between the computations of the automaton and the derivations of the grammar: a string x is accepted by the automaton if, and only if, it is generated by a derivation $q_0 \xrightarrow{+} x$.

Example 3.9 (From the automaton to the right-linear grammar) The automaton and corresponding grammar are



⁴Other more efficient equivalence tests do without first minimizing the automata.

Sentence bca is recognized in 3 steps by the automaton and it derives from the axiom in $3 + 1$ steps:

$$q_0 \Rightarrow bq_1 \Rightarrow bcq_0 \Rightarrow bcaq_0 \Rightarrow bca\epsilon$$

Observe the grammar contains empty rules but of course it can be turned into the non-nullable normal form by means of the transformation on p. 57.

For the example, first we find the set of nullable nonterminals $Null = \{q_0, q_2\}$; then we construct the equivalent rules:

$$q_0 \rightarrow aq_0 \mid bq_1 \mid a \mid \epsilon \quad q_1 \rightarrow cq_0 \mid aq_2 \mid bq_2 \mid a \mid b \mid c$$

Now q_2 is eliminated because its only rule is empty. Finally, grammar cleaning produces the rules:

$$q_0 \rightarrow aq_0 \mid bq_1 \mid a \mid \epsilon \quad q_1 \rightarrow cq_0 \mid a \mid b \mid c$$

Of course the empty rule $q_0 \rightarrow \epsilon$ must stay there because q_0 is the axiom, which causes the empty string to be a sentence.

In this normal form of the grammar it is evident that a move entering a final state r :

$$q \xrightarrow{a} r \rightarrow$$

may correspond to two rules $q \rightarrow ar \mid a$, one producing the nonterminal r , the other of the terminal type.

The conversion from automaton to grammar has been straightforward, but to make the reverse transformation from grammar to automaton, we need to modify the machine definition by permitting nondeterministic behavior.

3.5 Nondeterministic Automata

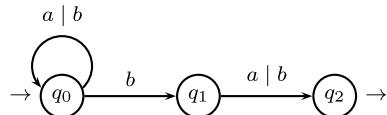
A right-linear grammar may contain two alternative rules:

$$A \rightarrow aB \mid aC \quad \text{where } a \in \Sigma, A, B, C \in V \quad \begin{array}{c} (C) \leftarrow a \\ \quad (A) \xrightarrow{a} (B) \end{array}$$

starting with the same character a . In this case, converting the rules to machine transitions, two arrows with identical label would 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 can choose which one of the next states to enter: its behavior is not deterministic. Formally the transition function takes two values, $\delta(A, a) = \{B, C\}$.

Similarly a copy rule:

$$A \rightarrow B \quad \text{where } B \in V \quad \begin{array}{c} (A) \xrightarrow{\epsilon} (B) \end{array}$$



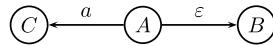
$$L_2 = (a \mid b)^* b (a \mid b)$$

$$N_2 :$$

Fig. 3.7 R.e. and corresponding nondeterministic machine N_2 checking the penultimate character is b

would be converted into an unusual machine transition from state A to state B , which does not read any terminal character (it would be odd to say it reads the empty string).

A machine move that does not read an input character is termed *spontaneous* or an *epsilon move*. Spontaneous moves too cause the machine to be nondeterministic, as in the following situation:



where in state A the automaton can choose to move spontaneously (i.e., without reading) to B , or to read a character, and if it is an a , to move to C .

3.5.1 Motivation of Nondeterminism

The mapping from grammar rules to transitions pushed us to introduce some transitions with multiple destinations and spontaneous moves, which are the main forms of indeterminism. Since this might appear as a useless theoretical concept, we hasten to list several advantages.

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 3.10 (Penultimate character) A sentence, like *abaabb*, is characterized by the presence of letter b in the penultimate position. The r.e. and nondeterministic recognizer are in Fig. 3.7.

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 characters; if it succeeds, the string is accepted. Thus string *baba* is recognized with the computation:

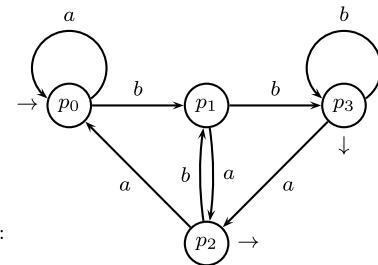
$$q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2$$

Notice 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.

Fig. 3.8 Deterministic recognizer M_2 of strings having b next to last



The same language is accepted by the deterministic automaton M_2 in Fig. 3.8. Clearly this machine is not just more complicated than the nondeterministic type in Fig. 3.7, but the latter makes it more 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 b . With little thought we see the nondeterministic automaton would have $k+1$ states, while one could prove that the number of states of the minimal deterministic machine is an exponential function of k .

In conclusion nondeterminism sometimes allows much shorter definitions.

3.5.1.2 Left Right Interchange and Language Reflection

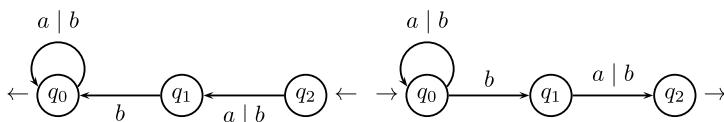
Nondeterminism also arises in string reversal when a given deterministic machine is transformed to recognize the reflection L^R of language L . This is sometimes required when for some reason strings must be scanned from right to left.

The new machine is straightforward to derive: interchange initial and final states⁵ and reverse all the arrows. Clearly this may give birth to nondeterministic transitions.

Example 3.11 The language having b as penultimate character (L_2 of Fig. 3.7) is the reflection of the language having the second character equal to b :

$$L' = \{x \in \{a, b\}^* \mid b \text{ is the second character of } x\} \quad L_2 = (L')^R$$

L' is recognized by the deterministic automaton (left):



⁵If the machine has several final states, multiple initial states result thus causing another form of indeterminism to be dealt with later.

Transforming the deterministic automaton for L' as explained above, we obtain the nondeterministic machine for L_2 (right), which by the way is identical to the one in Fig. 3.7.

As a further reason, we anticipate that nondeterministic machines are the intermediate product of 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 the *initial* states and the set F of *final* states
- the transition relation δ , a subset of the Cartesian product $Q \times \Sigma \times Q$

Remark This machine may have several initial states. In the graphic representation a transition $q_1 \xrightarrow{a} q_2$ is an arc labeled a , from the first to the second state.

As before, a *computation* is a series of transitions such that the origin of each one coincides with the destination of the preceding one:

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \cdots \xrightarrow{a_n} q_n$$

The computation *origin* is q_0 , the *termination* is q_n , and the *length* is the number n of transitions or moves. A computation of length 1 is just a transition. The computation *label* is the concatenation $a_1a_2\dots a_n$ of the characters read by each transition. In brief, the computation is also written $q_0 \xrightarrow{a_1a_2\dots a_n} q_n$.

A string x is *recognized or accepted* by the automaton, if it is the label of a computation originating in some initial state, terminating in some final state, and having label x .

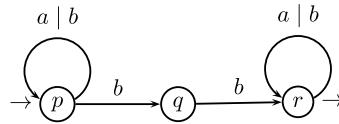
Let us focus on the empty string. We stipulate that every state is the origin and termination of a computation of length 0, having the empty string ε as label. It follows that the empty string is accepted by an automaton if, and only if, there exists an initial state which is also final.

The *language* $L(N)$ *recognized* by automaton N is the set of accepted strings:

$$L(N) = \{x \in \Sigma^* \mid q \xrightarrow{x} r \text{ with } q \in I, r \in F\}$$

Example 3.12 (Searching a text for a word) Given a string or word y and a text x , does x contain y as substring? The following machine recognizes the texts which contain one or more occurrences of y , that is, the language $(a \mid b)^* y (a \mid b)^*$. We

illustrate with the word $y = bb$:



String $abbb$ is the label of several computations originating in the initial state:

$$\begin{array}{ll} p \xrightarrow{a} p \xrightarrow{b} p \xrightarrow{b} p \xrightarrow{b} p & p \xrightarrow{a} p \xrightarrow{b} p \xrightarrow{b} p \xrightarrow{b} q \\ p \xrightarrow{a} p \xrightarrow{b} p \xrightarrow{b} q \xrightarrow{b} r & p \xrightarrow{a} p \xrightarrow{b} q \xrightarrow{b} r \xrightarrow{b} r \end{array}$$

The first two computations do not find the word looked for. The last two find the word, respectively, at position $ab \underbrace{bb}_{b}$ and $a \underbrace{bb}_{b} b$.

3.5.2.1 Transition Function

The moves of a nondeterministic automaton can still be considered as a finite function, but 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:

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \wp(Q)$$

where symbol $\wp(Q)$ indicates the powerset of set Q , i.e., the set of all the subsets of Q . Now the meaning of $\delta(q, a) = [p_1, p_2, \dots, p_k]$ is that the machine reading a in the current state q , can arbitrarily move to any of the states p_1, \dots, p_k . As we did for deterministic machines, we extend the function to any string y including the empty one:

$$\forall q \in Q : \delta(q, \varepsilon) = [q]$$

$$\forall q \in Q, y \in \Sigma^* : \delta(q, y) = [p \mid q \xrightarrow{y} p]$$

In other words, it is $p \in \delta(q, y)$ if there exists a computation labeled y from q to p .

The previous definitions allow a reformulation of the language accepted by automaton N :

$$L(N) = \{x \in \Sigma^* \mid \exists q \in I : \delta(q, x) \cap F \neq \emptyset\}$$

i.e., the set computed by function delta must contain a final state, for a string to be recognized.

Example 3.13 (Example 3.12 continued)

$$\delta(p, a) = [p], \quad \delta(p, ab) = [p, q], \quad \delta(p, abb) = [p, q, r]$$

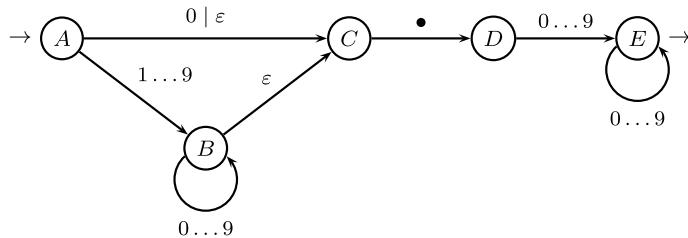


Fig. 3.9 Decimal constant definition with spontaneous moves (Example 3.14)

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*, to be depicted in the state-transition diagram as an arc labeled ε (named ε -arc). Such arcs will prove to be expedient for assembling the automata recognizing a regular composition of finite-state languages. The next example illustrates the case for union and concatenation.

Example 3.14 (Compositional definition of decimal constants) This language includes constants such as $90 \bullet 01$. The part preceding the decimal point may be missing (as in $\bullet 02$); but 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.:

$$L = (0 \mid \varepsilon \mid N) \bullet (0 \dots 9)^+ \quad \text{where } N = (1 \dots 9)(0 \dots 9)^*$$

The automaton in Fig. 3.9 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 computation originating in an initial state and terminating in a final state.

Observe that taking the spontaneous move from A to C , the integer part N vanishes. The string $34 \bullet 5$ is accepted with the computation:

$$A \xrightarrow{3} B \xrightarrow{4} B \xrightarrow{\varepsilon} C \xrightarrow{\bullet} D \xrightarrow{5} E$$

But the number of steps of the computation (i.e., the length of the path in the graph) 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 time complexity remains linear, because it is possible to assume that there are no cycles 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 Initial State

The official definition of nondeterministic machine allows two or more initial states, but 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 the ϵ -arcs going from it to the former initial states of the automaton. Clearly any computation on the new automaton accepts a string if, and only if, the old automaton does so.

The transformation has changed one form of indeterminism, linked with multiple initial states, to another form related to spontaneous moves. We shall see on p. 114 that such moves can be eliminated.

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 that it witnesses the two models are essentially notational variants.

Consider a right-linear grammar $G = (V, \Sigma, P, S)$ and a nondeterministic automaton $N = (Q, \Sigma, \delta, q_0, F)$, which we may assume from the preceding discussion to have a single initial state. First assume the grammar rules are strictly unilinear (p. 70). The states Q of the automaton match the nonterminals V of the grammar. The initial state corresponds to the axiom. Notice (row 3) that the pair of alternatives $p \rightarrow aq \mid ar$ correspond to two 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 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 left-linear grammars, since we know from Chap. 2 they have the same capacity as right-linear grammars.

If the grammar contains non-empty terminal rules of type $p \rightarrow a$ where $a \in \Sigma$, the automaton is modified, to include a new final state f , different from those produced by row 5 of Table 3.1, and the move $(p) \xrightarrow{a} (f) \rightarrow$.

Table 3.1 Correspondence between finite nondeterministic automata and right-linear grammars

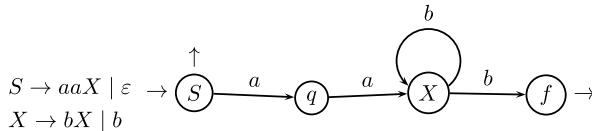
	Right-linear grammar	Finite automaton
1	Nonterminal set V	Set of states $Q = V$
2	Axiom $S = q_0$	Initial state $q_0 = S$
3	$p \rightarrow aq$, where $a \in \Sigma$ and $p, q \in V$	$(p) \xrightarrow{a} (q) \rightarrow$
4	$p \rightarrow q$, where $p, q \in V$	$(p) \xrightarrow{\epsilon} (q) \rightarrow$
5	$p \rightarrow \epsilon$	Final state $(p) \rightarrow$

Example 3.16 (Right-linear grammar and nondeterministic automaton) The grammar matching the automaton of decimal constants (Fig. 3.9 on p. 109) is

$$\begin{array}{ll} A \rightarrow 0C \mid C \mid 1B \mid \dots \mid 9B & B \rightarrow 0B \mid \dots \mid 9B \mid C \\ C \rightarrow \bullet D & D \rightarrow 0E \mid \dots \mid 9E \\ E \rightarrow 0E \mid \dots \mid 9E \mid \varepsilon & \end{array}$$

where A is the axiom.

Next we drop the assumption of unilinearity in the strict sense. Observe the right-linear grammar (left) and matching automaton (right) below:

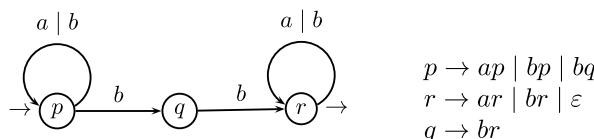


The not strictly right-linear rule $S \rightarrow aaX$ is simply converted to a cascade of two arcs, with an intermediate state q after the first a . Moreover, the new final state f has been added to mirror the last step of a derivation using rule $X \rightarrow b$.

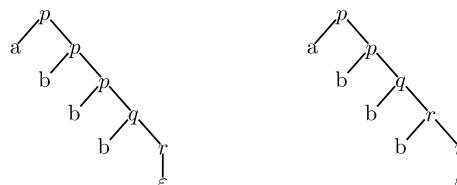
3.5.5 Ambiguity of Automata

An *automaton is ambiguous* if it accepts a string with two different computations. Clearly it follows from the definition that a deterministic automaton is never ambiguous. It is interesting to link the notion of ambiguity for automata and for unilinear grammars. Knowing that there is a 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 it generates a sentence with two distinct syntax trees.

Example 3.17 (Ambiguity of automaton and grammar) The automaton of Example 3.12 on p. 107, reproduced below:



recognizes string $abbb$ in two ways. The equivalent grammar (right) generates the same string with the trees:



3.5.6 Left-Linear Grammars and Automata

Remember that the *REG* family can also be defined using left-linear grammars. By interchanging left with right, it is simple to discover the mapping between such grammars and automata. Observe the forms of left-linear rules:

$$A \rightarrow Ba, \quad A \rightarrow B, \quad A \rightarrow \varepsilon$$

Consider such a grammar G and the specular language $L^R = (L(G))^R$: this language is generated by the reflected grammar, denoted G_R , obtained (p. 78) transforming the rules of the first form to $A \rightarrow aB$, while the remaining two forms do not change. Since the reflected grammar G_R is right-linear, we know how to construct a finite automaton N_R for L^R . In order to obtain the automaton of the original language L , we modify N_R , reversing the arrows of transitions and interchanging initial and final states.

Example 3.18 (From left-linear grammar to automaton) Given grammar G :

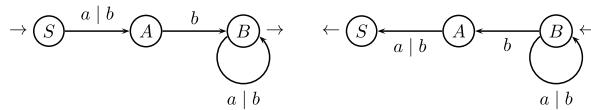
$$S \rightarrow Aa \mid Ab \quad A \rightarrow Bb \quad B \rightarrow Ba \mid Bb \mid \varepsilon$$

the reflected grammar G_R is

$$S \rightarrow aA \mid bA \quad A \rightarrow bB \quad B \rightarrow aB \mid bB \mid \varepsilon$$

The corresponding automaton N^R recognizing the mirror language and the recognizer N for language $L(G)$ are

Recognizer N_R of $(L(G))^R$ *Recognizer N of $L(G)$*



Incidentally, the language has a b in the position before the last.

3.6 Directly from Automata to Regular Expressions: BMC Method

In applied work one has sometimes to compute an r.e. for the language defined by a machine. We already know a rather indirect method: since an automaton is easily converted to an equivalent right-linear grammar, the r.e. of the language can be computed solving the linear simultaneous equations, seen on p. 71. The next direct *elimination method* **BMC** is named after Brzozowski and McCluskey.

Suppose for simplicity the initial state i is unique and no arc enters in it; similarly the final state t is unique and without outgoing arcs. Otherwise, just add a new initial

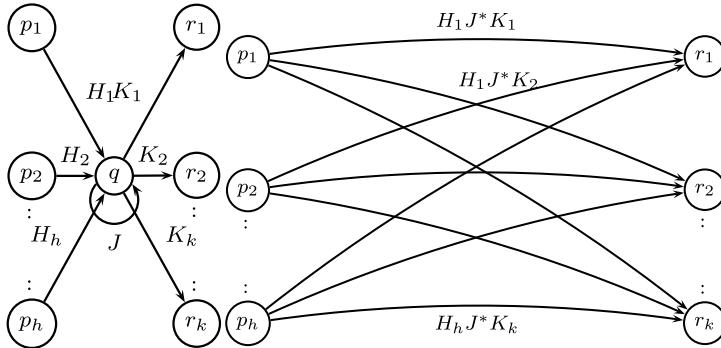
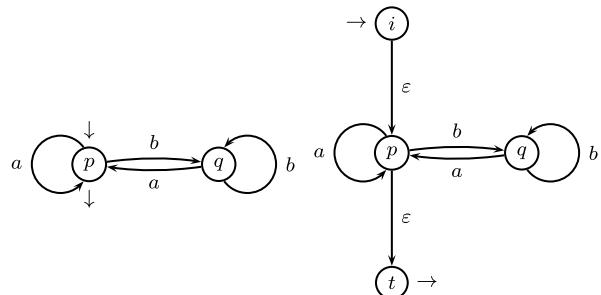


Fig. 3.10 Deleting a node and compensating with new generalized arcs

Fig. 3.11 Automaton before (left) and after (right) normalization (Example 3.19)



state i connected by spontaneous moves to the ex-initial states; similarly introduce a new unique final state t . Every state other than i and t is called *internal*.

We construct an equivalent automaton, termed *generalized*, which is more flexible as it allows 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 one by one the internal states, 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.

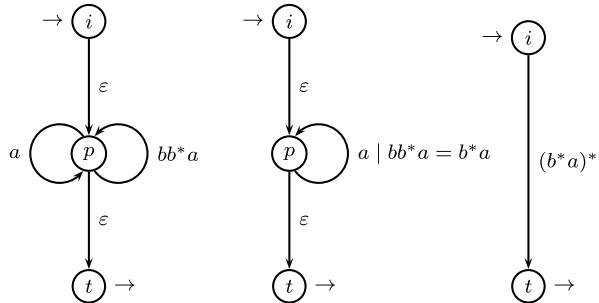
Observe in Fig. 3.10 to the left an internal state q with all adjoining arcs; to the right, the same fragment of machine, after eliminating state q and with compensatory arcs labeled with the same strings produced when traversing state q .

To avoid too many superpositions some labels are not printed, but of course, for every pair of states p_i and r_j there should be an arc $p_i \xrightarrow{H_i J^* K_j} r_j$. Notice that in the scheme some states p_i, r_j may coincide. It is evident that the set of strings which may be read when the original automaton moves from state p_i to state r_j , coincide with the language labeling the arc $p_i \rightarrow r_j$ 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 eliminating each time an internal state.

Example 3.19 (From Sakarovitch) The automaton is shown in Fig. 3.11 before and after normalization.

Fig. 3.12 Left to right: the automaton of Fig. 3.11 after elimination of node q , simplification of r.e., and elimination of node p



We trace in Fig. 3.12 the execution of the *BMC* algorithm, eliminating the states in the order q, p .

The elimination order does not affect the result, but, as in the solution of simultaneous equations by elimination, it may yield more or less complex yet equivalent solutions. For instance, the order p, q would produce the r.e. $(a^*b)^+a^+ \mid a^*$.

3.7 Elimination of Nondeterminism

We have argued for the use of nondeterministic machines in 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 to recognize strings: this happens for instance in a text editor, when implementing a searching algorithm by means of an automaton to be used only one time to find a string in a text.

Next we describe a procedure for turning a nondeterministic automaton into an equivalent deterministic one; as a corollary every unilinear grammar can be made non-ambiguous. The next transformation operates in two phases:

1. Elimination of spontaneous moves, obtaining a machine that in general is non-deterministic. We recall that, if the machine has multiple initial states, an equivalent machine can be obtained with one initial state, which is connected to the former initial states via spontaneous moves.
2. Replacing several not deterministic transitions with one transition entering a new state: this phase is called *powerset construction*, because the new states correspond to the subsets of the set of states.

We anticipate that a different algorithm combining the two phases into one, the Berry–Sethi method, is described in Sect. 3.8.2.4 (p. 130).

Elimination of Spontaneous Moves Because 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. 57). But first we show how to directly eliminate ε -arcs on the graph of the automaton.

For the automaton graph, we call ε -path a path made 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-2 path made by an ε -arc followed by a non- ε -arc, also to be called a *scanning move*, the algorithm adds the arc that scans the same terminal character. Finally, any state linked to a final state by a ε -arc is made final, all spontaneous arcs are deleted, as well as all useless states.

Algorithm 3.20 (Direct elimination of spontaneous moves) Let δ be the original transition graph, and $F \subseteq Q$ be the final states.

1. Transitive closure of ε -paths:

```
repeat
    add to  $\delta$  the arc  $p \xrightarrow{\varepsilon} r$  if  $\delta$  contains a path  $p \xrightarrow{\varepsilon} q \xrightarrow{\varepsilon} r$ , where  $p, q, r$  are distinct states;
    until no more arcs have been added in the last iteration.
```

2. Backwards propagation of scanning moves over ε -moves:

```
repeat add to  $\delta$  the arc  $p \xrightarrow{b} r$  if  $\delta$  contains a path  $p \xrightarrow{\varepsilon} q \xrightarrow{b} r$ , where  $p$  and  $q$  are distinct states;
    until no more arcs have been added in the last iteration.
```

3. New final states:

$$F := F \cup \{q \mid q \xrightarrow{\varepsilon} f \text{ is in } \delta \text{ and } f \in F\}.$$

4. Clean-up: delete all ε -arcs and the states that are not accessible from the initial state.

Example 3.21 For the automaton in Fig. 3.13 (top left), the graph resulting after transitive closure of $\xrightarrow{\varepsilon}$ is shown to the right, and the final automaton, after steps (2) and (3) of Algorithm 3.20, and the elimination of ε -arcs is below. Notice that state B is useless.

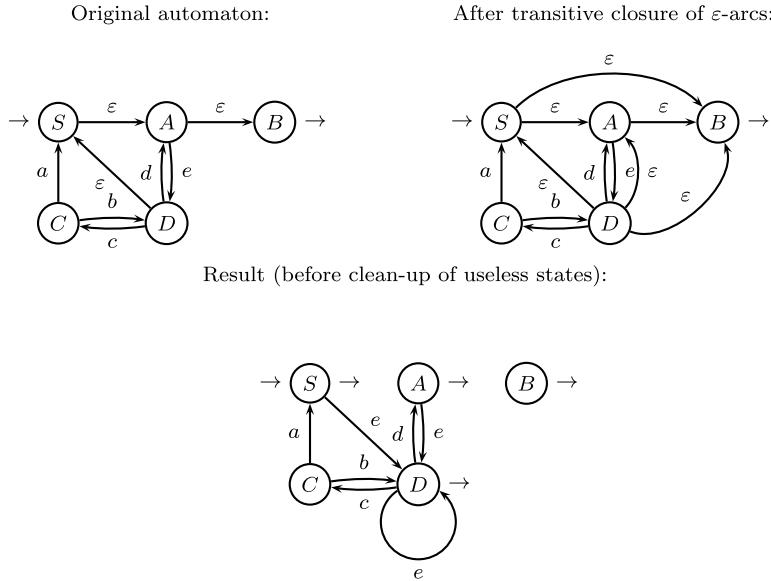
As said, an alternative way to solve the problem transforms the automaton into a right-linear grammar, from which the copy rules are then eliminated as explained on p. 57.⁶ The bottom part of Fig. 3.13 shows the details.

After this machine transformation, if the result is not deterministic, the next transformation must be applied.

3.7.1 Construction of Accessible Subsets

Given N , a nondeterministic automaton without spontaneous moves, we explain how to construct an equivalent deterministic machine M' . The main idea is that, if

⁶The copy rule elimination algorithm works for right-linear grammars although the grammar contains empty rules.



Automaton transformation via elimination of grammar copy rules.

Original left-linear grammar:

$$\begin{array}{lll} S \rightarrow A & A \rightarrow B \mid eD & B \rightarrow \varepsilon \\ C \rightarrow aS \mid bD & D \rightarrow S \mid cC \mid dA & \end{array}$$

Result:

$$\begin{array}{lll} S \rightarrow \varepsilon \mid eD & A \rightarrow \varepsilon \mid eD & B \rightarrow \varepsilon \\ C \rightarrow aS \mid bD & D \rightarrow \varepsilon \mid eD \mid cC \mid dA & \end{array}$$

Fig. 3.13 Automaton with spontaneous moves (*top left*), with arcs added by transitive closure (*top right*), and after steps (2) and (3) of Algorithm 3.20 (*middle*). Equivalent grammar before and after copy rule elimination (*bottom*)

N contains the moves:

$$p \xrightarrow{a} p_1, \quad p \xrightarrow{a} p_2, \quad \dots, \quad p \xrightarrow{a} p_k$$

after reading a , machine N can be in any one of the next states p_1, p_2, \dots, p_k , i.e., it is in a state of uncertainty. Then we create in M' a new collective state named:

$$[p_1, p_2, \dots, p_k]$$

that will simulate the uncertainty.

To connect the new state to the others, we construct the outgoing arcs according to the following rule. If a collective state contains the states p_1, p_2, \dots, p_k , for each

one we consider in N the outgoing arcs labeled with the same letter a :

$$p_1 \xrightarrow{a} [q_1, q_2, \dots], \quad p_2 \xrightarrow{a} [r_1, r_2, \dots], \quad \text{etc.}$$

and we merge together the next states:

$$[q_1, q_2, \dots] \cup [r_1, r_2, \dots] \cup \dots$$

thus obtaining the collective state reached by transition:

$$[p_1, p_2, \dots, p_k] \xrightarrow{a} [q_1, q_2, \dots, r_1, r_2, \dots]$$

If such state does not exist in M' , it is added to the current state set.

Algorithm 3.22 (Powerset construction) The deterministic automaton M' equivalent to N is defined by

1. state set Q' is the powerset of Q , i.e., $Q' = \wp(Q)$
2. final states $F' = \{p' \in Q' \mid p' \cap F \neq \emptyset\}$, which are the states containing a final state of N
3. initial state⁷ $[q_0]$
4. transition function δ' :
for all $p' \in Q'$ and for all $a \in \Sigma$

$$p' \xrightarrow{a} [s \mid q \in p' \wedge (\text{arc } q \xrightarrow{a} s \text{ is in } N)]$$

In step (4), if an arc $q \xrightarrow{a} q_{\text{err}}$ leads to the error state, it is not added to the collective state: in fact, any computation entering the sink never recognizes any string and can be ignored.

Because the states of M' are the subsets of Q , the cardinality of Q' is in the worst case exponentially larger than the cardinality of Q . This confirms previous findings that deterministic machines may be larger: remember the exponential explosion of the number of states in the language having a specific character k positions before the last (p. 105).

The algorithm can be improved: 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 which can be reached from the initial state.

Example 3.23 The nondeterministic automaton in Fig. 3.14 (top) is transformed to the deterministic one below.

Explanations: from $\delta(A, b) = \{A, B\}$ we draw the collective state $[A, B]$; from this the transitions:

⁷If the given automaton N has several initial states, the initial state of M' is the set of all initial states.

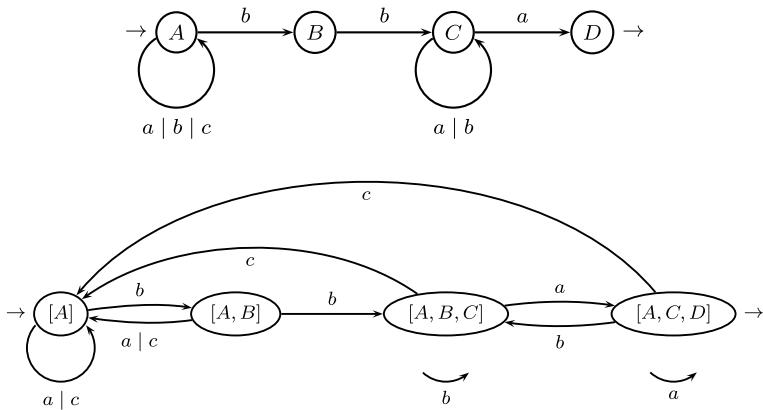


Fig. 3.14 From nondeterministic N (top) to deterministic M' (bottom)

$$[A, B] \xrightarrow{a} (\delta(A, a) \cup \delta(B, a)) = [A]$$

$$[A, B] \xrightarrow{b} (\delta(A, b) \cup \delta(B, b)) = [A, B, C]$$

Then we create the transitions originating from the new collective state $[A, B, C]$:

$$[A, B, C] \xrightarrow{a} (\delta(A, a) \cup \delta(B, a) \cup \delta(C, a)) = [A, C, D], \quad \text{final collective state}$$

$$[A, B, C] \xrightarrow{b} (\delta(A, b) \cup \delta(B, b) \cup \delta(C, b)) = [A, B, C], \quad \text{self-loop}$$

etc. The algorithm ends when step (4), applied to the current set Q' of states, does not generate any new state. Notice that not all subsets of Q correspond to an accessible state: e.g., the subset and state $[A, C]$ would be useless.

To justify the correctness of the algorithm, we show that a string x is recognized by M' if, and only if, it is accepted by N .

If a computation of N accepts x , there exists a labeled path 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 $[\dots, q_f, \dots]$ containing q_f .

Conversely, if x is the label of a valid computation of M' , from $[q_0]$ to a final state $p \in F'$, then by definition p contains at least one final state q_f of N . By construction there exists in N a labeled path x from q_0 to q_f .

We summarize with a fundamental statement.

Property 3.24 Every finite-state language can be recognized by a deterministic automaton.

This property ensures that the recognition algorithm of finite-state languages works in real time, i.e., completes the job within a number of transitions equal to the length of the input string (fewer if an error occurs before the string has been entirely scanned).

As a corollary, for every language recognized by a finite automaton, there exists an unambiguous unilinear grammar, the one naturally (p. 103) corresponding to the deterministic automaton. This also says that for any regular language we have a procedure to eliminate ambiguity from the grammar, in other words a regular language cannot be inherently ambiguous (p. 53).

3.8 From Regular Expression to Recognizer

When a language is specified with a regular expression, instead of a unilinear grammar, we do not know how to construct its automaton. Since this requirement is quite common in applications such as compilation and document processing, 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*, because it analyzes the expression into smaller and smaller subexpressions until the atomic ones. Then the recognizers of subexpressions are constructed and interconnected into a graph that implements the language operations (union, concatenation, and star) present in the expression. The result is in general nondeterministic with spontaneous moves.

The second method, named after Berry and Sethi, builds a deterministic machine, though possibly non-minimal in the number of states.

The Thompson method can be combined with previous determinization algorithms, to the effect of directly producing a deterministic automaton.

Eventually we will be able to transform language specifications back and forth from automata, grammars and regular expressions, thus proving the three models are equivalent.

3.8.1 Thompson Structural Method

Given an r.e. we analyze it into simple parts, we produce 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 for the *BMC* algorithm of p. 112.

The Thompson algorithm⁸ incorporates the mapping rule between simple r.e. and automata schematized in Table 3.2.

Observing the machines in Table 3.2 one sees many nondeterministic bifurcations, with outgoing ϵ -arcs.

⁸Originally presented in [15]. It forms the base of the popular tool *lex* (or GNU *flex*) used for building scanners.

Table 3.2 From component subexpression to component subautomata. A rectangle depicts a component subautomaton with its unique initial (*left*) and final (*right*) states

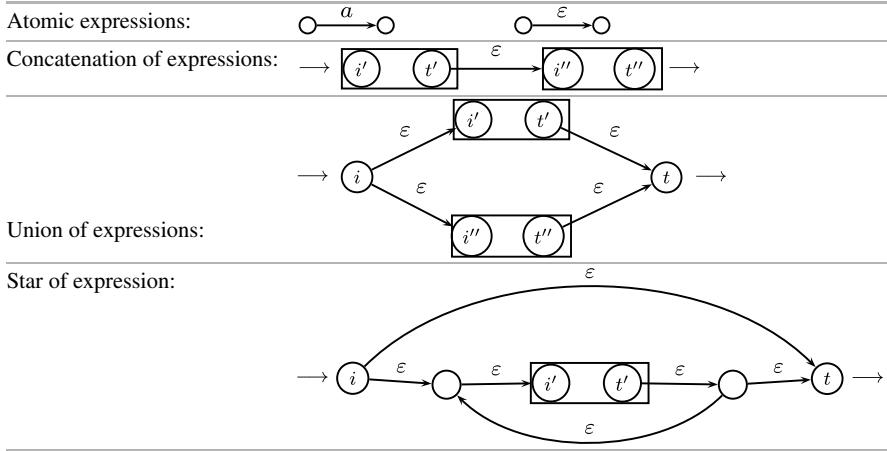
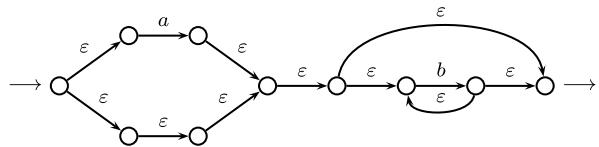


Fig. 3.15 The automaton obtained with the structural (Thompson) method from r.e. $(a \cup \epsilon).b^*$



The validity of Thompson's method comes from it being an operational reformulation of the closure properties of regular languages under concatenation, union, and star (stated in the preceding chapter, p. 23).

Example 3.25 Take the r.e. $(a \cup \epsilon).b^*$ and decompose it into the subexpressions:

$$\overbrace{\quad\quad\quad}^{E_{0,11}} \\
 \overbrace{\quad\quad\quad}^{E_{1,6}} \quad \overbrace{\quad\quad\quad}^{E_{7,10}} \\
 \overbrace{\quad\quad\quad}^{E_{2,3}} \quad \overbrace{\quad\quad\quad}^{E_{4,5}} \quad \overbrace{\quad\quad\quad}^{E_{8,9}} \\
 (\underbrace{a}_{E_{2,3}} \cup \underbrace{\epsilon}_{E_{4,5}}) . \underbrace{b^*}_{E_{8,9}}$$

Then apply the mapping to each subexpression, producing the automaton of Fig. 3.15.

Notice that we have moderately simplified the constructions to avoid the proliferation of states. Of course, several states are redundant and could be coalesced.

Existing tools use improved versions of the algorithm to avoid constructing redundant states. Other versions combine the algorithm with the one for elimination of spontaneous moves.

It is interesting to look at the Thompson algorithm as a converter from the notation of r.e. to the state-transition graph of the machine. From this standpoint this is a typical case of syntax-directed analysis and translation, where the syntax is the one of r.e.; such approach to translator design is presented in Chap. 5.

3.8.2 Berry–Sethi Method

Another classical method, due to Berry and Sethi [1] derives a *deterministic* automaton recognizing the language of a given regular expression. To justify this construction, we preliminary introduce⁹ local languages, a simple subfamily of regular languages, and the related notions of local automata and linear regular expressions.

3.8.2.1 Locally Testable Languages and Local Automata

Some regular languages are extremely simple to recognize, because it suffices to test if certain short substrings are present. An example is the set of strings starting with b , ending with a or b , and containing ba or ab as substrings.

Definition 3.26 For a language L of alphabet Σ , the *set of initials* is

$$Ini(L) = \{a \in \Sigma \mid a \Sigma^* \cap L \neq \emptyset\}$$

i.e., the starting characters of the sentences of L .

The *set of finals* is

$$Fin(L) = \{a \in \Sigma \mid \Sigma^* a \cap L \neq \emptyset\}$$

i.e., the last characters of the sentences of L .

The *set of digrams* (or factors) is

$$Dig(L) = \{x \in \Sigma^2 \mid \Sigma^* x \Sigma^* \cap L \neq \emptyset\}$$

i.e., the substrings of length 2 present in the sentences of L . The complementary digrams are

$$\overline{Dig}(L) = \Sigma^2 \setminus Dig(L)$$

The three sets will be called *local*.

Example 3.27 (Locally testable language) The local sets for language $L_1 = (abc)^*$ are

$$Ini(L_1) = a \quad Fin(L_1) = c \quad Dig(L_1) = \{ab, bc, ca\}$$

and the complement of the digram set is

$$\overline{Dig(L_1)} = \{aa, ac, ba, bb, cb, cc\}$$

⁹Following the conceptual path of Berstel and Pin [2].

We observe that the non-empty sentences of this language are precisely defined by the three sets, in the sense of the following identity:

$$L_1 \setminus \{\varepsilon\} = \{x \mid \text{Ini}(x) \in \{a\} \wedge \text{Fin}(x) \in \{c\} \wedge \text{Dig}(x) \subseteq \{ab, bc, ca\}\}$$

Definition 3.28 A language L is called *local* (or locally testable) if it satisfies the following identity:

$$L \setminus \{\varepsilon\} = \{x \mid \text{Ini}(x) \in \text{Ini}(L) \wedge \text{Fin}(x) \in \text{Fin}(L) \wedge \text{Dig}(x) \subseteq \text{Dig}(L)\} \quad (3.1)$$

Notice that the above definition of local language only refers to its *non-empty* strings, hence the inclusion of the ε string in any language L is immaterial to L being local; for instance, language $L_2 = (abc)^+ = L_1 \setminus \{\varepsilon\}$ is also local.

Although not all languages are local, it should be clear that any language L satisfies a modified condition (3.1) where the equal sign is substituted by the inclusion. In fact, by definition every sentence starts (resp. ends) with a character of $\text{Ini}(L)$ (resp. of $\text{Fin}(L)$) and its digrams are included in the set $\text{Dig}(L)$. But such conditions may be satisfied by some strings that do not belong to the language: in this case the language is not local, because it does not include all strings that can be generated from Ini , Dig , and Fin .

Example 3.29 (Nonlocal regular language) For $L_2 = b(aa)^+b$ we have

$$\text{Ini}(L_2) = \text{Fin}(L_2) = \{b\} \quad \text{Dig}(L_2) = \{aa, ab, ba\} \quad \overline{\text{Dig}(L_2)} = \{bb\}$$

The sentences of L_2 have even length, but among the strings starting and terminating with b , which do not contain bb as digram, there are those of odd length, such as $baaab$, and those containing some b 's surrounded by a 's, such as $babaab$. Therefore the language defined by condition (3.1) strictly includes L_2 , which therefore is not local.

Automata Recognizing Local Languages Our present interest¹⁰ for local languages comes from the notable simplicity of their recognizers. To recognize a string, the machine scans it from left to right, it checks that the initial character is in Ini , it verifies that any pairs of adjacent characters are in Dig , and finally it checks that the last character is in Fin . These checks are easily performed by a finite automaton.

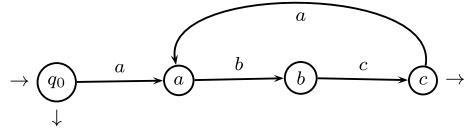
An automaton recognizing a local language $L \subseteq \Sigma^*$, specified by the local sets Ini , Fin , Dig , is therefore easily obtained, as next specified.

Construction 3.30 (Recognizer of a local language specified by its local sets)

- The states are $q_0 \cup \Sigma$, meaning that each one of the latter states is identified by a terminal character.

¹⁰In Chap. 5 local languages and automata are also used to model the control-flow graph of a program.

Fig. 3.16 Local automaton
recognizing language $(abc)^*$



- The final states include *Fin*, and, if the empty string is in the language, also q_0 ; no other state is final.
- The transitions are $q_0 \xrightarrow{a} a$ if $a \in \text{Ini}$; $a \xrightarrow{b} b$ if $ab \in \text{Dig}$.

The deterministic automata produced by this construction have certain features that characterize the family of *local automata*, to be next defined.

Definition 3.31 A deterministic automaton $A = (Q, \Sigma, \delta, q_0, F)$ is *local* if all transitions labeled by the same symbol $a \in \Sigma$ enter the same state, i.e., if

$$\forall a \in \Sigma \quad |\{\delta(q, a) \mid q \in Q\}| \leq 1$$

Moreover, notice that a local automaton obtained by Construction 3.30 enjoys some additional properties:

The non-initial states correspond one-to-one to the terminals (3.2)

(i.e., there are $|Q| = |\Sigma| + 1$ states)

No arc enters the initial state q_0 (3.3)

The local automata that additionally satisfy conditions (3.2) and (3.3) are termed *normalized*. Clearly a normalized local automaton recognizes a local language. Intuitively, such automaton is in the state b if, and only if, the last scanned character is b : we may think that the machine has a sliding window two characters wide, which triggers the transition from the previous state to the current one if the diagram is listed in *Dig*. Therefore the following equivalence holds.

Proposition 3.32 For a language L , the two conditions of being a local language and of being recognized by a normalized local automaton are equivalent.

Example 3.33 The normalized local automaton accepting the language $(abc)^*$ of Example 3.27 is shown in Fig. 3.16.

Composition of Local Languages with Disjoint Alphabets Before we apply the sliding window idea to a generic regular expression, another conceptual step is needed, based on the following observation: the basic language operations preserve the property of being a local language, provided the terminal alphabets of the languages to be combined are disjoint.

Property 3.34 Given local languages L' and L'' with disjoint alphabets, i.e., $\Sigma' \cap \Sigma'' = \emptyset$, the union $L' \cup L''$, concatenation $L'.L''$ and star L'^* (and cross L^+ too) are local languages.

Proof it is immediate to construct a normalized local automaton for the resulting language by combining the component machines (also to be called L' and L'' with a slight abuse) as next explained. Let q'_0, q''_0 be the respective initial states and F', F'' the sets of final states of the component machines. In general the normalized local automaton contains the states of the component machines, with some adjustments on the initial and final states and their arcs, as next described.

Construction 3.35 (Composition of normalized local automata over disjoint alphabets) For the union $L' \cup L''$:

- the initial state q_0 is obtained merging the initial states q'_0 and q''_0 ;
- the final states are those of the component machines, $F' \cup F''$; if the empty string belongs to either one or both languages, the new state q_0 is marked as final.

For the concatenation $L'.L''$:

- the initial state is q'_0 ;
- the arcs of the recognizer are:
 - the arcs of L' plus
 - the arcs of L'' , except those exiting from the initial state q''_0
 - the latter are substituted by the arcs $q' \xrightarrow{a} q''$, from every final state $q' \in F'$ to a state q'' such that there is in L'' an arc $q''_0 \xrightarrow{a} q''$;
- the final states are F'' , if $\varepsilon \notin L''$; otherwise the final states are $F' \cup F''$.

For the star L'^* :

- from each final state $q \in F'$ we draw the arc $q \xrightarrow{a} r$ if machine L' has arc $q'_0 \xrightarrow{a} r$ (exiting from the initial state);
- state q'_0 is added to the final states F' .

It is straightforward to see that the above steps correctly produce the recognizer of the union, concatenation, or star of a local language. Clearly such machines have by construction the characteristic properties of normalized local automata. \square

Example 3.36 We illustrate in Fig. 3.17 the Construction 3.35 of the normalized local recognizer for the r.e. $(ab \cup c)^*$, which is obtained, starting with the atomic subexpressions, by the steps:

concatenate a and b ; unite with c ; apply star to $(ab \cup c)$.

3.8.2.2 Linear Regular Expressions and Their Local Sets

In a generic r.e. a terminal character may of course be repeated. An r.e. is called *linear* if no terminal character is repeated. For instance, $(abc)^*$ is linear whereas $(ab)^*a$ is not, because character a is repeated.

Property 3.37 The language defined by a linear r.e. is local.

Proof From the hypothesis of linearity it follows that the subexpressions have disjoint alphabets. Since the r.e. is obtained by composition of the local languages of the subexpressions, Property 3.34 ensures that the language it defines is local. \square

<i>subexpression</i>	<i>component automata</i>
<i>elements</i> a, b, c	
<i>concatenation and union</i> $ab \cup c$	
<i>star</i> $(ab \cup c)^*$	

Fig. 3.17 Step-by-step composition of normalized local automata for the linear r.e. $(ab \cup c)^*$ of Example 3.36

Notice that linearity of a regular expression is a sufficient condition for its language to be local, but it is not necessary: the regular expression $(ab)^*a$, though not linear, defines a local language.

Having ascertained that the language of a linear r.e. is local, the problem of constructing its recognizer melts down to the computation of the sets *Ini*, *Fin*, and *Dig* of the language. We next explain how to orderly perform the job.

Computing Local Sets of Regular Languages We list in Table 3.3 the rules for computing the three local sets of any regular expression.

First we must check if an r.e. e is *nullable*, i.e., if $\varepsilon \in L(e)$. To this end we define the predicate *Null*(e) inductively on the structure of e through the rules in Table 3.3, top. To illustrate, we have

$$\begin{aligned} \text{Null}((a \cup b)^*ba) &= \text{Null}((a \cup b)^*) \wedge \text{Null}(ba) \\ &= \text{true} \wedge ((\text{Null}(b) \wedge \text{Null}(a))) = \text{true} \wedge (\text{false} \wedge \text{false}) = \text{false} \end{aligned}$$

We observe that the above rules are valid for any r.e., but we will apply them only to linear expressions.

Table 3.3 Rules for computing predicate Null and the local sets Ini , Fin , and Dig

$\text{Null}(\epsilon) = \text{true}$	$\text{Null}(\emptyset) = \text{false}$
$\text{Null}(a) = \text{false}$ for every terminal a	$\text{Null}(e \cup e') = \text{Null}(e) \vee \text{Null}(e')$
$\text{Null}(e.e') = \text{Null}(e) \wedge \text{Null}(e')$	$\text{Null}(e^*) = \text{true}$
$\text{Null}(e^+) = \text{Null}(e)$	
<hr/>	
Initials	Finals
$\text{Ini}(\emptyset) = \emptyset$	$\text{Fin}(\emptyset) = \emptyset$
$\text{Ini}(\epsilon) = \emptyset$	$\text{Fin}(\epsilon) = \emptyset$
$\text{Ini}(a) = \{a\}$ for every terminal a	$\text{Fin}(a) = \{a\}$ for every terminal a
$\text{Ini}(e \cup e') = \text{Ini}(e) \cup \text{Ini}(e')$	$\text{Fin}(e \cup e') = \text{Fin}(e) \cup \text{Fin}(e')$
$\text{Ini}(e.e') = \begin{cases} \text{Null}(e) \\ \text{then } \text{Ini}(e) \cup \text{Ini}(e') \\ \text{else } \text{Ini}(e) \end{cases}$	$\text{Fin}(e.e') = \begin{cases} \text{Null}(e') \\ \text{then } \text{Fin}(e) \cup \text{Fin}(e') \\ \text{else } \text{Fin}(e') \end{cases}$
$\text{Ini}(e^*) = \text{Ini}(e^+) = \text{Ini}(e)$	$\text{Fin}(e^*) = \text{Fin}(e^+) = \text{Fin}(e)$
<hr/>	
Digrams	
$\text{Dig}(\emptyset) = \emptyset$	
$\text{Dig}(\epsilon) = \emptyset$	
$\text{Dig}(a) = \emptyset$, for every terminal a	
$\text{Dig}(e \cup e') = \text{Dig}(e) \cup \text{Dig}(e')$	
$\text{Dig}(e.e') = \text{Dig}(e) \cup \text{Dig}(e') \cup \text{Fin}(e)\text{Ini}(e')$	
$\text{Dig}(e^*) = \text{Dig}(e^+) = \text{Dig}(e) \cup \text{Fin}(e)\text{Ini}(e)$	

Given a linear regular expression, the finite automaton recognizing its language can be constructed by computing its local sets and then applying Construction 3.30 (p. 122), which allows one to derive an automaton recognizing a local language given its local sets.

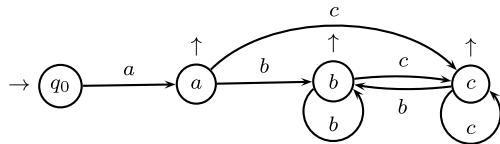
Example 3.38 (Recognizer of linear r.e.) For the linear regular expression $a(b \cup c)^*$ the local sets are

$$\text{Ini} = a \quad \text{Fin} = \{b, c\} \cup a = \{a, b, c\} \quad \text{Dig} = \{ab, ac\} \cup \{bb, bc, cb, cc\}$$

Applying Construction 3.30 of p. 122, we produce the local automaton in Fig. 3.18, where the property that all arcs labeled by the same symbol enter the same state is easily checked.

Numbered Regular Expressions Given any regular expression e over alphabet Σ , we obtain from it a linear regular expression e' as follows: we make all terminals distinct by progressively numbering them, say from left to right, with an integer subscript; the alphabet of e' is denoted as Σ_N . Thus, for instance, the regular expression $e = (ab)^*a$ becomes $e' = (a_1b_2)^*a_3$, and $\Sigma_N = \{a_1, b_2, a_3\}$.

Fig. 3.18 Local automaton of Example 3.38, recognizing the language $a(b \cup c)^*$



3.8.2.3 From Generic Regular Expressions to Deterministic Recognizers

We are ready to present a construction for obtaining a deterministic recognizer for any regular expression, through a sequence of steps based on previous results for local languages and linear expressions.

Construction 3.39 (Deterministic automaton recognizing the language of a regular expression)

1. From the original r.e. e over alphabet Σ derive the linear expression $e' \dashv$, where e' is the numbered version of e and \dashv is a string terminator symbol, with $\dashv \notin \Sigma$.
2. Build the local automaton recognizing the local language $L(e' \dashv)$: this automaton includes the initial state q_0 , one non-initial and non-final state for each element of Σ_N , and a unique final state \dashv .
3. Label each state of the automaton with the set of the symbols on its outgoing edges. The initial state q_0 is labeled with $Ini(e' \dashv)$, the final state \dashv is labeled with the empty set \emptyset . For each non-initial and non-final states c , $c \in \Sigma_N$, the set labeling that state is called the *set of followers* of symbol c , $Fol(c)$, in the expression $e' \dashv$; it is derived directly from the local set of Digrams as follows: $Fol(a_i) = \{b_j \mid a_i b_j \in Dig(e' \dashv)\}$. Fol is equivalent to the Dig local set and, together with the other two local sets Ini and Fin , characterizes a local language.
4. Merge any existing states of the automaton that are labeled by the same set. The obtained automaton is equivalent to the previous one: since the recognized language is local, states marked with equal sets of followers are indistinguishable (see Definition 3.6, p. 101).
5. Remove the numbering from the symbols that label the transitions of the automaton: the resulting automaton, which may be nondeterministic, accepts by construction the language $L(e \dashv)$.¹¹
6. Derive a deterministic, equivalent automaton by applying the construction of Accessible Subsets (Sect. 3.7.1, p. 115); label the sets resulting from the union of several states of the previous nondeterministic automaton with the union of the sets labeling the merged states. The resulting deterministic automaton recognizes $L(e \dashv)$.
7. Remove from the automaton the final state (labeled by \emptyset) and all arcs entering it (i.e., all arcs labeled by \dashv); define as final states of the resulting automaton those labeled by a set that includes the \dashv symbol; the resulting automaton is deterministic and recognizes $L(e)$.

¹¹This is an example of transliteration (homomorphism) defined on p. 79.

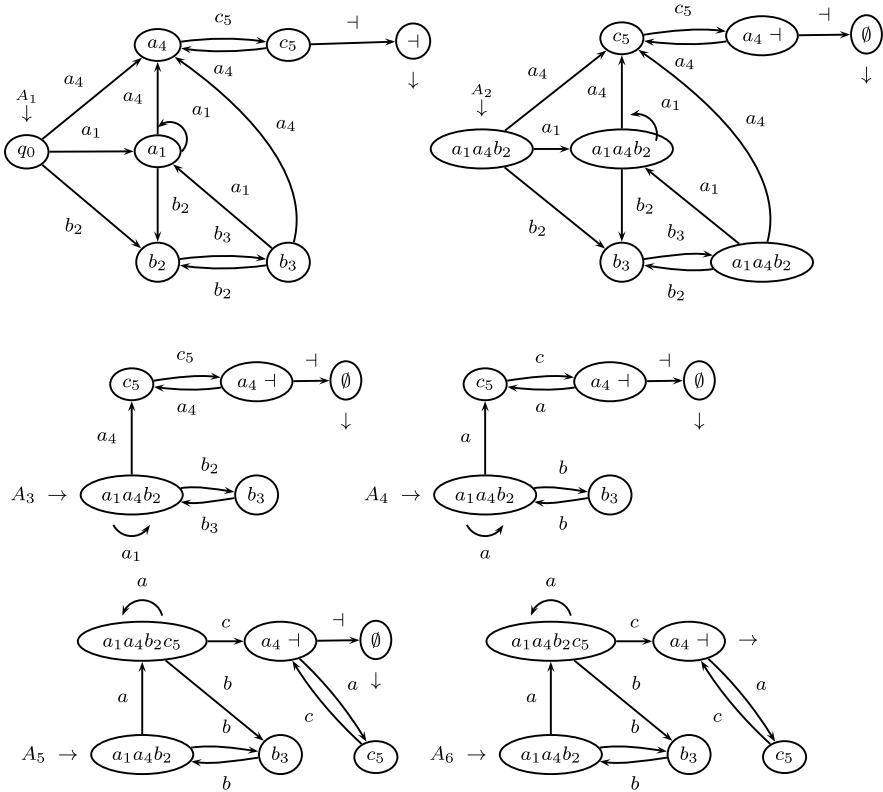


Fig. 3.19 Phases of Berry-Sethi Construction 3.39 of the recognizer for language $e = (a \cup bb)^*(ac)^+$ of Example 3.40

It is worth noticing that restricting the above construction to steps (1), (2), (5), and (7) yields another classical method called *GMY* (due to Glushkov and to McNaughton and Yamada) to obtain a machine recognizing the language of a generic r.e. Now each state corresponds to one letter of the numbered alphabet Σ_N of e' ; there are no spontaneous moves. However, in contrast to the result of Construction 3.39, the machine may be nondeterministic: for instance, from $a \cup ab$, numbered $a_1 \cup a_2b_3$, the arcs $q_0 \xrightarrow{a} a_1$ and $q_0 \xrightarrow{a} a_2$ are obtained.

Example 3.40 We apply Construction 3.39 to r.e. $e = (a \cup bb)^*(ac)^+$, obtaining the intermediate results shown in Fig. 3.19 and listed next.

1. The numbered version of e is $e' = (a_1 \mid b_2b_3)^*(a_4c_5)^+$ and its alphabet is $\Sigma_N = \{a_1, b_2, b_3, a_4, c_5\}$.
2. The local sets needed for building the local automaton are

$$\text{Ini}(e' \dashv) = \{a_1, b_2, a_4\}$$

$$Dig(e' \dashv) = \{a_1a_1, a_1b_2, b_2b_3, b_3a_1, a_1a_4, b_3a_4, a_4c_5, b_3b_2, c_5a_4, c_5 \dashv\}$$

$$Fin(e' \dashv) = \{\dashv\}$$

and the normalized local automaton is A_1 (Fig. 3.19).

3. The initial state q_0 is labeled with $Ini(e' \dashv) = \{a_1, b_2, a_4\}$. The automaton with states labeled with the sets of followers is A_2 of the same figure. Notice that three distinct states are labeled by the same set of symbols, because $Ini(e' \dashv) = Fol(a_1) = Fol(b_3) = \{a_1, a_4, b_2\}$.
4. The automaton obtained by merging identically labeled states is A_3 .
5. The automaton A_4 recognizes $L((a \mid bb)^*(ac)^+ \dashv)$ and is nondeterministic due to the two a -labeled arcs outgoing from the initial state.
6. The deterministic automaton A_5 recognizes $L(e \dashv)$.
7. The deterministic automaton recognizing $e = (a \mid bb)^*(ac)^+$ is A_6 ; the (unique) final state is the one labeled with set $\{a_4 \dashv\}$.

3.8.2.4 Deterministic Recognizer by Berry and Sethi Algorithm

The Construction 3.39 (p. 127) for building a deterministic recognizer of the language of a generic regular expression can be optimized thus obtaining the Berry and Sethi Algorithm.¹² For the reader's convenience we remind the reader that the numbered version of a r.e. e of alphabet Σ is denoted by e' and by $e' \dashv$ when followed by the end of text mark; by construction $e' \dashv$ is a linear r.e., whose alphabet is $\Sigma_N \cup \{\dashv\}$. For any symbol $a_i \in \Sigma_N$ the set of a_i 's followers $Fol(a_i) \subseteq (\Sigma_N \cup \{\dashv\})$ is defined as $Fol(a_i) = \{b_j \mid a_i b_j \in Dig(e' \dashv)\}$.

Instead of starting from the local automaton for the linear expression $e' \dashv$ and transforming it into an equivalent deterministic automaton, the optimized algorithm generates incrementally the states of the final deterministic machine, each one identified by the set of followers of the last letter that has been read. The construction thus avoids state duplication and performs on the fly the determinization by merging the states reached through distinctly subscripted versions $b_i, b_j \in \Sigma_N$ of the same letter $b \in \Sigma$.

Algorithm 3.41 (Algorithm BS (Berry and Sethi)) Each state is denoted by a subset of $(\Sigma_N \cup \{\dashv\})$. When a state is examined, it is marked as visited to prevent the algorithm from reexamining it. Final states are exactly those containing the element \dashv .

```

 $q_0 := Ini(e' \dashv)$ 
 $Q := \{q_0\}$ 
 $\delta := \emptyset$ 
while a not yet visited state  $q$  exists in  $Q$  do
    mark  $q$  as visited
    for every character  $b \in \Sigma$  do

```

¹²For a thorough justification of the method we refer to [1].

```


$$q' := \bigcup_{\forall b_i \in \Sigma_N \text{ s.t. } b_i \in q} Fol(b_i)$$


if  $q' \neq \emptyset$  then
  if  $q' \notin Q$  then
    set  $q'$  as a new unmarked state
     $Q := Q \cup \{q'\}$ 
  end if
   $\delta := \delta \cup \{q \xrightarrow{b} q'\}$ 
end if

end do
end do

```

Example 3.42 Apply the Berry and Sethi Algorithm to the r.e. $e = (a \mid bb)^*(ac)^+$ already considered in Example 3.40 at p. 128. The deterministic automaton is shown in Fig. 3.20.

For the numbered r.e. $e' \dashv$, the initials and followers are

x	Fol(x)
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

$$e' \dashv = (a_1 \mid b_2 b_3)^* (a_4 c_5)^+ \dashv$$

$$Ini(e' \dashv) = \{a_1, b_2, a_4\}$$

We mention that the automata thus produced may contain more states than necessary, but of course can be minimized by the usual method.

Use for Determinizing an Automaton Another use of algorithm BS is as an alternative to the powerset construction, for converting a nondeterministic machine N into a deterministic one M . This approach is actually more flexible since it ap-

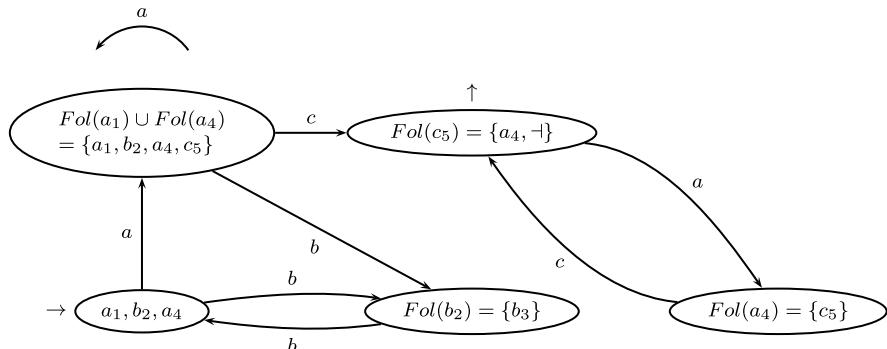
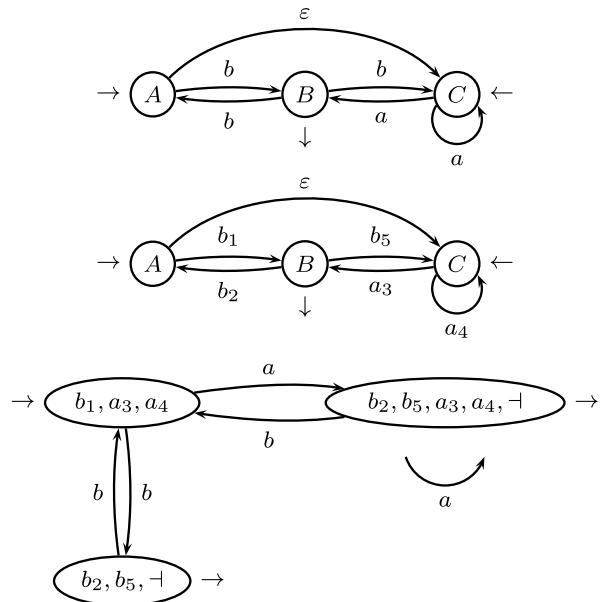


Fig. 3.20 Direct construction of the deterministic automaton for $(a \mid bb)^*(ac)^+$ (Example 3.42)

Fig. 3.21 Automaton N (top) with spontaneous moves and numbered version N' (middle). Deterministic machine constructed by the BS algorithm (bottom) (Example 3.44)



plies to all forms of nondeterminism, including multiple initial states and ε -arcs. We proceed as follows.

Construction 3.43 (Determinization of a finite automaton)

1. Distinctly number the labels of non- ε arcs of N , obtaining automaton N' .
2. Compute the local sets Ini , Fin , and Fol for the language $L(N')$. These can be easily derived from the transition graph, possibly exploiting the identity $\varepsilon a = a\varepsilon = a$.
3. Applying the BS construction to the sets Ini , Fin , and Fol , produce the deterministic automaton M .

The validity of this construction is justified by noting that the language recognized by automaton N' corresponds to all its successful computations (i.e., to all paths from an initial state to a final state), and the sets Ini , Fin , and Fol , derived from the transition graph, characterize exactly the labels of such successful computations. Therefore, the language accepted by N' coincides with the set of strings that can be obtained from the local sets in all possible ways, hence it satisfies condition (3.1) (p. 122) and is a local language.

It is sufficient to illustrate this application with an example.

Example 3.44 Given the nondeterministic automaton N of Fig. 3.21 (top), numbering the arc labels we obtain automaton N' (bottom).

Its language is local and not nullable. Then compute the initials:

$$Ini(L(N') \dashv) = \{b_1, a_3, a_4\}$$

and note that $\varepsilon a_4 = a_4$ and $\varepsilon a_3 = a_3$. Continue with the set of followers:

x	$Fol(x)$
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

Finally apply algorithm BS to construct the deterministic automaton M of Fig. 3.21, bottom.

3.9 Regular Expressions with Complement and Intersection

Before leaving regular languages, we complete the study of 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. Using the properties of finite automata, we can now state and prove a property anticipated in Chap. 1.

Property 3.45 (Closure of REG by complement and intersection) Let L and L' be regular languages. The complement $\neg L$ and the intersection $L \cap L'$ are regular languages.

First we show how to build the recognizer of the complement $\neg L = \Sigma^* \setminus L$. We assume the recognizer M of L is deterministic, with initial state q_0 , state set Q , final state set F , and transition function δ .

Algorithm 3.46 (Construction of deterministic recognizer \overline{M} of complement) First we complete the automaton M , in order to make its function total, by adding the error or sink state p and the arcs to and from it.

1. Create a new state $p \notin Q$, the *sink*; the states of \overline{M} are $Q \cup \{p\}$
2. the transition function $\overline{\delta}$ is
 - (a) $\overline{\delta}(q, a) = \delta(q, a)$, where $\delta(q, a) \in Q$;
 - (b) $\overline{\delta}(q, a) = p$, where $\delta(q, a)$ is not defined;
 - (c) $\overline{\delta}(p, a) = p$, for every character $a \in \Sigma$;
3. the final states are $\overline{F} = (Q \setminus F) \cup \{p\}$.

Notice the final and non-final states have been interchanged.

To justify the construction, first observe that, if a computation of M accepts a string $x \in L(M)$, then the corresponding computation of \overline{M} terminates in a non-final state, so that $x \notin L(\overline{M})$.

Second, if a computation of M does not accept 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 y is in $L(\overline{M})$.

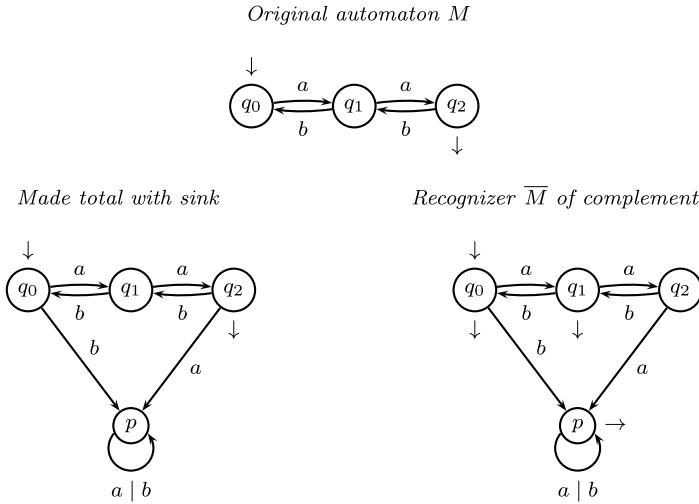


Fig. 3.22 Construction of recognizer of complement (Example 3.47)

Finally, in order to show that the intersection of two regular languages is regular, it suffices to quote the well-known De Morgan identity:

$$L \cap L' = \neg(\neg L \cup \neg L')$$

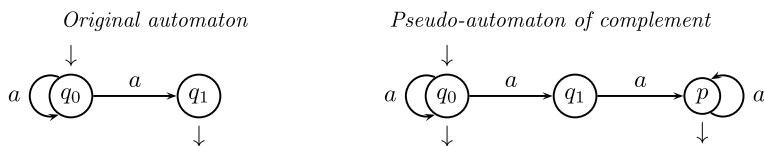
because, knowing that the languages $\neg L$ and $\neg L'$ are regular, their union too is regular as well as its complement.

As a corollary, the *set difference* of two regular languages is regular because of the identity:

$$L \setminus L' = L \cap \neg L'$$

Example 3.47 (Automaton of complement) Figure 3.22 shows three machines: the given one M , the intermediate completed with sink, and the recognizer \overline{M} of complement.

For the construction to work the original machine must be deterministic, otherwise the language accepted by the constructed machine may be nondisjoint from the original one, which would violate the characteristic property of complement, $L \cap \neg L = \emptyset$. See the following counterexample where the pseudo-complement machine mistakenly accepts string a , which is in the original language.



Finally we mention the construction may produce unclean or non-minimal machines.

3.9.1 Product of Automata

A frequently used technique consists in simulating two (or more) machines by a single one having as state set the Cartesian product of the two state sets. We present the technique for the case of the recognizer of the intersection of two regular languages.

Incidentally, the proof of the closure of family REG under intersection based on De Morgan identity (p. 133) already gives a procedure for recognizing the intersection: given two finite deterministic recognizers, first construct the recognizers of the complement languages, then the recognizer of their union (by the method of Thompson on p. 119). From the latter (after determinization 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 machines to be free from spontaneous moves, but not necessarily deterministic.

The *product machine* M has state set $Q' \times Q''$, the Cartesian product of the two state sets. This means each state is a pair $\langle q', q'' \rangle$, where the first (second) component is a state of the first (second) machine. For such pair or state we construct the outgoing arc:

$$\langle q', q'' \rangle \xrightarrow{a} \langle r', r'' \rangle$$

if, and only if, there exist 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 first (resp. second) component exists in M' (resp. in M'').

The initial states I of M are the product $I = I' \times I''$ of the initial states of the component machines; the final states are the product of the final states, i.e., $F = F' \times F''$.

To justify the correctness of the construction consider any string x in the intersection. Since x is accepted by a computation of M' as well as by a computation of M'' it is also accepted by the computation of M that traverses the pairs of states respectively traversed by the two computations.

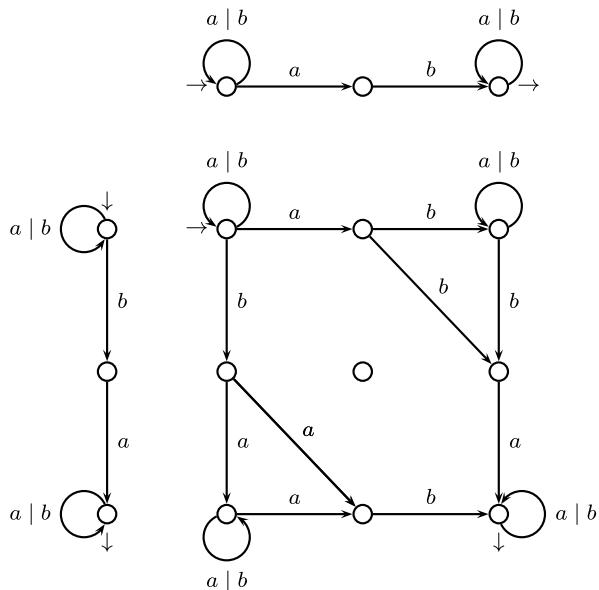
Conversely, if x is not in the intersection, at least one of the computations by M' or by M'' does not reach a final state, hence the computation of M does not reach a final state because the latter are pairs of final states.

Example 3.48 (Intersection and product machine (Sakarovitch)) The recognizer of the strings containing as substrings both ab and ba is quite naturally specified through the intersection of languages L', L'' :

$$L' = (a \mid b)^* ab(a \mid b)^* \quad L'' = (a \mid b)^* ba(a \mid b)^*$$

The Cartesian product of the recognizers of the component languages is shown in Fig. 3.23.

Fig. 3.23 Recognizers of component languages (top and left) and product machine recognizing their intersection (Example 3.48)



As usual the pairs of the Cartesian product, which are not accessible from the initial state, 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 of the recognizer of the union of two languages. But the size of the product machine may be much larger than the size of the machine obtained with the Thompson method: the number of states would be in the order of the product of the sizes of the machines, instead of the sum of the sizes.

3.9.1.1 Extended and Restricted Regular Expressions

A *regular expression* is *extended* if it uses other operators beyond the basic ones (i.e., union, concatenation, star, cross): complement, intersection, and set difference.

For instance, the strings containing one or multiple occurrences of aa as substring and not ending with bb are defined by the extended r.e.:

$$((a | b)^* aa(a | b)^*) \cap \neg((a | b)^* bb)$$

The next realistic example shows the practicality of using extended expressions for greater expressivity.

Example 3.49 (Identifiers) Suppose the valid identifiers may contain letters $a \dots z$, digits $0 \dots 9$ (not in first position) and the dash ' $-$ ' (neither in first nor in last position). Adjacent dashes are not permitted. A sentence of this language is *after-2nd-test*.

The next extended r.e. 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:

$$\begin{aligned} & \underbrace{(a \dots z)^+ (a \dots z \mid 0 \dots 9 \mid -)^*}_{\text{(i)}} \cap \\ & \quad \neg \left((a \dots z \mid 0 \dots 9 \mid -)^* - - (a \dots z \mid 0 \dots 9 \mid -)^* \right) \cap \\ & \quad \neg \left((a \dots z \mid 0 \dots 9 \mid -)^* - \right) \end{aligned}$$

(ii)

(iii)

When a language is specified by an extended r.e. we can of course construct its recognizer by applying the complement and Cartesian product methods. Then a non-extended r.e. can be obtained if desired.

3.9.1.2 Star-Free Languages

We know the addition of complement and intersection operators to r.e. does not enlarge the family of regular languages, because they can be eliminated and replaced by basic operators. On the other hand, removing the star from the permitted operators causes a loss of generative capacity: the family of languages shrinks into a subfamily of the *REG* family, variously named as *aperiodic* or *star-free* or *non-counting*. Since such 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 , we can write a so-called *star-free* r.e. using only these operators. Notice the presence of intersection and complement is essential to compensate somehow for the loss of the star (and the cross), otherwise just finite languages would be defined.

A language is called *star-free* if there exists a star-free r.e. that defines it.

First, observe the universal language of alphabet Σ is star-free, since it is the complement of the empty set, i.e., it is defined by the star-free r.e. $\Sigma^* = \neg \emptyset$.

Second, a useful subclass of star-free languages has already been studied without even knowing the term: the local languages on p. 121 can indeed be defined without stars or crosses. Recall that a local language is specified by three local sets: initials, finals, and permitted (or forbidden) digrams. Its specification can be directly mapped on the intersection of three star-free languages as next illustrated.

Example 3.50 (Star-free r.e. of local and nonlocal languages) The sentences of local language $(abc)^+$ (Example 3.27 on p. 121) start with a , end with c , and do not contain any digram from $\{aa \mid ac \mid ba \mid bb \mid cb \mid cc\}$. The language is therefore

specified by the star-free r.e.:

$$(a - \emptyset) \cap (\neg \emptyset c) \cap (\neg (\neg \emptyset(aa | ac | ba | bb | cb | cc) \neg \emptyset))$$

Second example. The language $L_2 = a^*ba^*$ is star-free because it can be converted to 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 because the local sets do not suffice to eliminate spurious strings. Amidst the strings that, as prescribed by L_2 , start and end with a or b and may contain the digrams $\{aa \mid ab \mid ba\}$, we find the string, say, $abab$, which does not belong to L_2 .

The family of star-free languages is strictly included in the family of regular languages. It excludes in particular the languages characterized by certain counting properties that justify the other name “non-counting” of the family. An example is the regular language:

$$\{x \in \{a \mid b\}^* \mid |x|_a \text{ is even}\}$$

which is accepted by a machine having in its graph a length-2 circuit, i.e., a modulo 2 counter (or flip-flop) of the letters a encountered. This language cannot be defined with an r.e not 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 if ever needed. In other words, the classification of strings based on the classes of congruences modulo some integer is usually not correlated with their being valid sentences or not. For reasons which may have to do with the organization of the human mind or perhaps with 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. But on one side star-free r.e. are often less readable than basic ones. On the other side, in different fields of computer science, counting is of 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.¹³

¹³For the theory of star-free languages we refer to McNaughton and Papert [11].

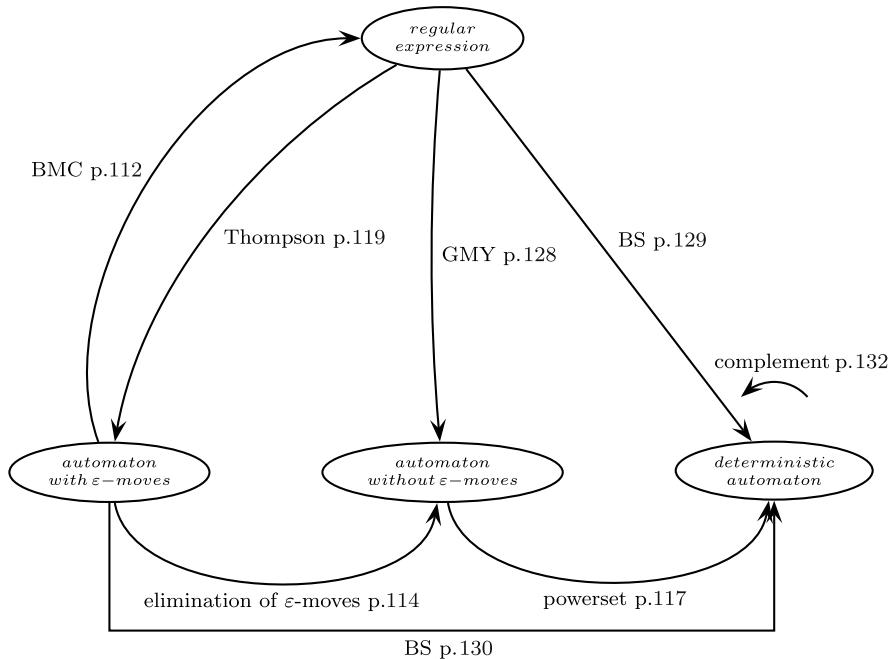
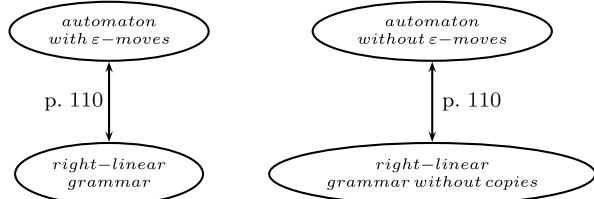


Fig. 3.24 Conversion methods between r.e. and finite automata deterministic and not

Fig. 3.25 Correspondence between right-linear grammars and finite automata



3.10 Summary of Relations Between Regular Languages, Grammars, and Automata

As we leave 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 family of languages. Figure 3.24 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 converts an r.e. to an automaton devoid of spontaneous moves.

Fig. 3.26 Correspondence between regular expression and grammars

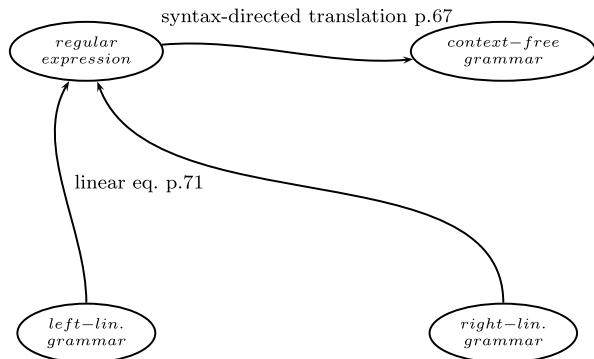


Figure 3.25 represents the direct correspondence between right-linear grammars and finite automata.

The relations between automata and left-linear grammars are not listed, because they are essentially identical, thanks to the left/right duality of grammar rules and the arrow reversing transformation of automaton moves.

Finally, Fig. 3.26 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, finite automata, and unilinear grammars.

Finally we recall that regular languages are a very restricted subset of the context-free, which are indispensable for defining artificial languages.

References

1. G. Berry, R. Sethi, From regular expressions to deterministic automata. *Theor. Comput. Sci.* **48**(1), 117–126 (1986)
2. J. Berstel, J.E. Pin, Local languages and the Berry–Sethi algorithm. *Theor. Comput. Sci.* **155**(2), 439–446 (1996)
3. D. Bovet, P. Crescenzi, *Introduction to the Theory of Complexity* (Prentice-Hall, Englewood Cliffs, 1994)
4. R. Floyd, R. Beigel, *The Language of Machines: An Introduction to Computability and Formal Languages* (Computer Science Press, New York, 1994)
5. M. Harrison, *Introduction to Formal Language Theory* (Addison Wesley, Reading, 1978)
6. J. Hopcroft, J. Ullman, *Formal Languages and Their Relation to Automata* (Addison-Wesley, Reading, 1969)
7. J. Hopcroft, J. Ullman, *Introduction to Automata Theory, Languages, and Computation* (Addison-Wesley, Reading, 1979)
8. D. Kozen, *Theory of Computation* (Springer, London, 2007)
9. D. Mandrioli, C. Ghezzi, *Theoretical Foundations of Computer Science* (Wiley, New York, 1987)
10. R. McNaughton, *Elementary Computability, Formal Languages and Automata* (Prentice-Hall, Englewood Cliffs, 1982), p. 400
11. R. McNaughton, S. Papert, *Counter-Free Automata* (The MIT Press, Cambridge, 1971)
12. G. Rozenberg, A. Salomaa (eds.), *Handbook of Formal Languages, vol. 1: Word, Language, Grammar* (Springer, New York, 1997).

13. J. Sakarovitch, *Elements of Automata Theory* (Cambridge University Press, Cambridge, 2009)
14. A. Salomaa, *Formal Languages* (Academic Press, New York, 1973)
15. K. Thompson, Regular expression search algorithm. Commun. ACM **11**(6), 419–422 (1968)
16. B. Watson, A taxonomy of finite automata minimization algorithms. Report, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands (1994)

4.1 Introduction

The algorithms for recognizing if a string is a legal sentence require more memory and time resources for context-free than for regular languages. This chapter presents several algorithms, first as abstract automata with a pushdown memory stack, then as parsing¹ (or syntax analysis) procedures that produce the syntax tree of a sentence. Incidentally, parsing has little interest for regular languages because their syntax structure is usually predeterminate (left- or right-linear), whereas for context-free languages any tree structure is possible.

Similarly to unilinear grammars, also context-free grammar rules can be made to correspond to the moves of an automaton, which is no longer a pure finite-state machine but also has a pushdown stack memory. In contrast to the finite-state case, a deterministic pushdown machine exists only for a subfamily of context-free languages known as deterministic, *DET*. Moreover, the presence of two memories, i.e., states and a stack, introduces a variety of functioning modes, which complicate the theoretical study of such machines.

The chapter starts with the essentials of pushdown automata, and compares the general and the deterministic case in terms of expressivity and of closure with respect to language operations.

Then several useful parsing algorithms are described, which altogether meet the requirements of compiler writing. It is traditional to classify the algorithms as pushdown stack-based and tabular (or general). Of the former we only deal with the deterministic ones, which are further divided into the classes of bottom-up (also known as *LR(k)* or shift-reduce) and top-down (also named *LL(k)* or predictive), depending on the construction order of syntax tree. Both types make the assumption that the language is *DET* and impose specific restrictions on the form of grammar rules. On the other hand, the general algorithms typically use tables instead of a stack and are able to cope with ambiguous and nondeterministic grammars, but they pay the price of non-linear time complexity.

¹From Latin *pars*, *partis*, in the sense of dividing a sentence into its parts or constituents.

To avoid the tedious and irrelevant diversity of the historical parsing methods, we unify their presentation. Our approach is based on representing the grammar as a network of finite automata. Such automata networks are the natural representation for grammars with regular expressions in the right parts of their rules, the *extended context-free form* or *EBNF* (p. 82) frequently used in language reference manuals. Accordingly, all the sequential parsers to be presented are qualified as extended. Starting from the bottom-up deterministic methods, we are able to incrementally develop the top-down and the general algorithms, respectively, as specialization and as generalization of the bottom-up method.

Then we address the problem of improving performance of parsing procedures on multi-processor computers. To enable parallel execution of parsing steps, the parser must be able to make parsing decisions examining a limited part of the source text, so that several workers can independently analyze different text segments. A parallel algorithm using Floyd's operator-precedence method is presented, which performs well on very large texts.

The chapter proceeds as follows. After the automata theoretic foundations, we focus on the representation of a grammar as a network of finite automata. For them we present three cases: deterministic algorithms, bottom-up *ELR(1)* parsers and top-down *ELL(1)* parsers as special case allowing an elegant recursive implementation; and the general non-deterministic Early parser. We show how to adjust a grammar to deterministic parsing and how to make a deterministic parser more selective by using longer prospection. Then, returning to the theory, we compare the language families that correspond to the various parsers considered. Then the parallel parsing algorithm is presented. After a brief presentation of error recovery in parsing, we finish with a discussion of choice criteria for parsers.

4.2 Pushdown Automaton

Any 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, which stores the symbols:

$$\underbrace{A_1 \ A_2 \dots}_{\text{bottom symbol}} \quad \underbrace{A_k}_{\text{top symbol}}$$

The input or source string, delimited on the right end by an end-marker \dashv , is

$$a_1 a_2 \dots \underbrace{a_i}_{\substack{\text{current} \\ \text{character}}} \dots a_n \dashv$$

The following operations apply to a stack:

pushing: $\text{push}(B)$ inserts symbol B on top, i.e., to the right of A_k ; several push operations can be combined in a single command $\text{push}(B_1 B_2 \dots B_n)$ that inserts B_n on top

emptiness test: *empty*, the predicate is true if, and only if, $k = 0$

popping: *pop*, if the stack is not empty it removes the top symbol A_k

It is sometimes convenient to imagine that a special symbol is painted on the bottom of the stack, denoted Z_0 and termed the *bottom*. Such a symbol may be read but neither pushed nor popped. The presence of Z_0 on top of the stack means that the stack is empty.

The machine reads the source characters from left to right by using a reading head. The character under the reading head is termed *current*. At each instant the machine *configuration* is specified by: the remaining portion of the input string still to be read; the current state; and the stack contents. With a move the automaton may:

- read the current character and shift the reading head, or perform a spontaneous move without reading
- read the top symbol and pop it, or read the bottom Z_0 if the stack is empty
- compute the next state from the values of current state, current character and stack symbol
- push one or more symbols onto the stack (or none)

4.2.1 From Grammar to Pushdown Automaton

We show how grammar rules can be interpreted as instructions of a non-deterministic pushdown machine that recognizes the grammar language. The machine is so simple that it works without any internal state, but only uses 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 symbols are nonterminal and terminal characters of the grammar. If the stack contains symbols $A_k \dots A_1$ from top to bottom, 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 prescribed by A_k has to recognize if the source string, starting from the current character a_i , contains a substring w that derives from A_k . If so, the action will eventually shift the reading head of $|w|$ positions. Naturally enough, the goal may recursively spawn subgoals if, for recognizing the derivation from A_k , it is necessary to recognize other terminal or nonterminal symbols. Initially, the first goal is the axiom of the grammar: the task of the machine is to recognize if the source string derives from the axiom.

Algorithm 4.1 (From the grammar to the 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 form of a rule shapes the move. For rule (2), the right-hand side starts with a terminal and the move is triggered by reading it. On the contrary, rules (1) and (3) give rise to spontaneous moves that do not check the current character. Move (4) checks that a terminal, when it surfaces on the stack top (being 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.

Table 4.1 Correspondence between grammar rules and moves of a nondeterministic pushdown machine with one state; the current input character is represented by cc

#	Grammar rule	Automaton move	Comment
1	$A \rightarrow BA_1 \dots A_n$ with $n \geq 0$	If $top = A$ then pop; push $(A_n \dots A_1 B)$	To recognize A first recognize $BA_1 \dots A_n$
2	$A \rightarrow bA_1 \dots A_n$ with $n \geq 0$	If $cc = b$ and $top = A$ then pop; push $(A_n \dots A_1)$; shift reading head	Character b was expected as next one and has been read so it remains to recognize $A_1 \dots A_n$
3	$A \rightarrow \varepsilon$	If $top = A$ then pop	The empty string deriving from A has been recognized
4	For every character $b \in \Sigma$	If $cc = b$ and $top = b$ then pop; shift reading head	Character b was expected as next one and has been read
5	–	If $cc = \dashv$ and the stack is empty then accept; halt	The string has been entirely scanned and the agenda contains no goals

Initially the stack contains 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 (not deterministically) chooses a move, 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 this machine model recognizes a sentence by *empty stack*.

Surprisingly enough, this automaton never changes state and the stack is its only memory. Later on, we will be obliged to introduce states in order to make deterministic the behavior of the machine.

Example 4.2 (From a grammar to a pushdown automaton) The moves of the recognizer of the language L

$$L = \{a^n b^m \mid n \geq m \geq 1\}$$

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 the current one; similarly for the choice between moves 3 and 4.

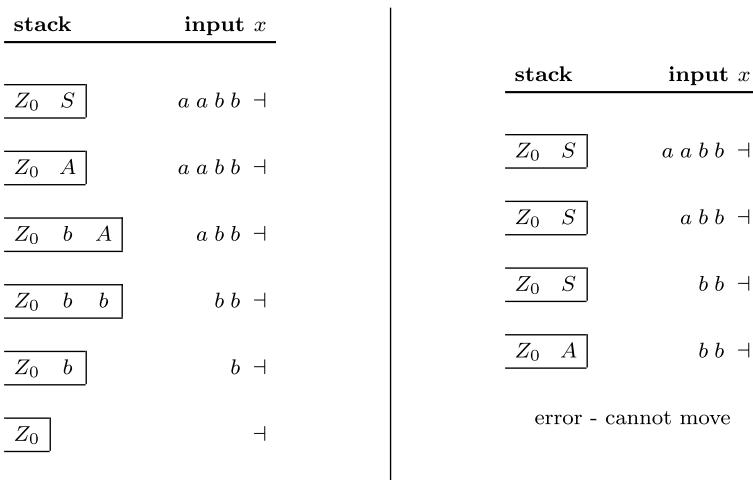
It is not difficult to see that a string is accepted by this machine if, and only if, it is generated by the grammar. In fact, for each accepting computation there exists a corresponding derivation and conversely; in other words the automaton simulates the leftmost derivations of the grammar. For instance, the derivation

$$S \Rightarrow A \Rightarrow aAb \Rightarrow aabb$$

mirrors the successful trace in Fig. 4.1 (left).

Table 4.2 Pushdown machine moves for the grammar rules of Example 4.2

#	Grammar rule	Automaton move
1	$S \rightarrow aS$	If $cc = a$ and $top = S$ then pop; push (S); shift
2	$S \rightarrow A$	If $top = S$ then pop; push (A)
3	$A \rightarrow aAb$	If $cc = a$ and $top = A$ then pop; push (bA); shift
4	$A \rightarrow ab$	If $cc = a$ and $top = A$ then pop; push (b); shift
5		If $cc = b$ and $top = b$ then pop; shift
6		If $cc = \vdash$ and the stack is empty then accept; halt

**Fig. 4.1** Two computations: accepting (*left*) and rejecting by error (*right*)

But Algorithm 4.1 does not have any a priori information about which derivation of the possible ones will succeed, if any at all, and thus it has to explore all 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. 144 is bidirectional and 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 permits us to state an important theoretical fact, which links grammars and pushdown automata.

Property 4.3 The family CF of context-free languages coincides with the family of the languages accepted with empty stack by a nondeterministic pushdown machine having one state.

We stress that the mapping from automaton to grammar does not work when the pushdown automaton has a set of states; other methods will be developed in that case.

It may appear that in little space and 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 non-deterministic and in the worst case it is forced to explore all computation paths, with a time complexity non-polynomial with respect to the length of the source string; more efficient algorithms are wanted for practical application.

4.2.1.1 Computational Complexity of Pushdown Automata

We compute an upper bound on the number of steps needed to recognize a string with the previous pushdown machine in the worst case. For simplicity we consider a grammar G in the Greibach normal form (p. 65), which as we know features rules starting with a terminal and not containing other terminals. Therefore the machine is free from spontaneous moves (types (1) and (3) of Table 4.1 on p. 144) and it never pushes a terminal character onto the stack.

For a string x of length n the derivation $S \xrightarrow{+} x$ has exactly n steps, if it exists. The same number of moves is performed by the automaton to recognize string x . For any grammar nonterminal let K be the maximum number of alternative rules $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$. At each step of a leftmost derivation, the leftmost nonterminal 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 derivations before finding the accepting one or declaring a failure, the time complexity 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 other data structures instead of a *LIFO* stack.

4.2.2 Definition of Pushdown Automaton

We are going to define several pushdown machine models. We trust the reader to adapt to the present context the analogous concepts already seen for finite automata, in order to expedite the presentation.

A pushdown automaton M is defined by seven items:

1. Q , a finite *set of states* of the control unit
2. Σ , an *input alphabet*
3. Γ , a *stack (or memory) alphabet*
4. δ , a *transition function*
5. $q_0 \in Q$, an *initial state*
6. $Z_0 \in \Gamma$, an *initial stack symbol*
7. $F \subseteq Q$, a *set of final states*

Such a machine is in general non-deterministic. The domain and range of the transition function are made of Cartesian products:

Domain	Range
$Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$	$\wp(Q \times \Gamma^*)$ i.e., the power set of the set $Q \times \Gamma^*$

The moves, i.e., the values of the function δ , fall into the following cases:
reading move

$$\delta(q, a, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \dots, (p_n, \gamma_n)\}$$

with $n \geq 1$, $a \in \Sigma$, $Z \in \Gamma$ and with $p_i \in Q$, $\gamma_i \in \Gamma^*$.

The machine, in the state q with Z on top of the stack, reads character a and enters one of the states p_i , with $1 \leq i \leq n$, after performing these operations: pop and push (γ_i).

Notes: the choice of the i th action among n possibilities is not deterministic; the reading head automatically shifts forward; the top symbol is always popped; and the string pushed onto the stack may be empty.

spontaneous move

$$\delta(q, \epsilon, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \dots, (p_n, \gamma_n)\}$$

with the same stipulations as before.

The machine, in the state q with Z on top of the stack and without reading an input character, enters one of the states p_i , with $1 \leq i \leq n$, after performing these operations: pop and push (γ_i).

From the definition it is clear that the behavior can be non-deterministic for two causes: the range of the transition function comprises a set of alternative actions and the machine may contain spontaneous moves.

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 remaining portion (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.

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 \mapsto^* . As customary, the cross instead of the star denotes a computation with at least one transition.

Depending on the move the following transitions are possible:

Current conf.	Next conf.	Applied move
$(q, az, \eta Z)$	$(p, z, \eta\gamma)$	Reading move $\delta(q, a, Z) = \{(p, \gamma), \dots\}$
$(q, az, \eta Z)$	$(p, az, \eta\gamma)$	Spontaneous move $\delta(q, \epsilon, Z) = \{(p, \gamma), \dots\}$

Although a move erases the top symbol, the same symbol can be pushed again by the move itself if the computation needs to keep it on stack.

A string x is *recognized* (or accepted) with *final state* if there exists a computation that entirely reads the string and terminates in a final state:

$$(q_0, x, Z_0) \xrightarrow{*} (q, \varepsilon, \lambda) \quad q \text{ is a final state and } \lambda \in \Gamma^*$$

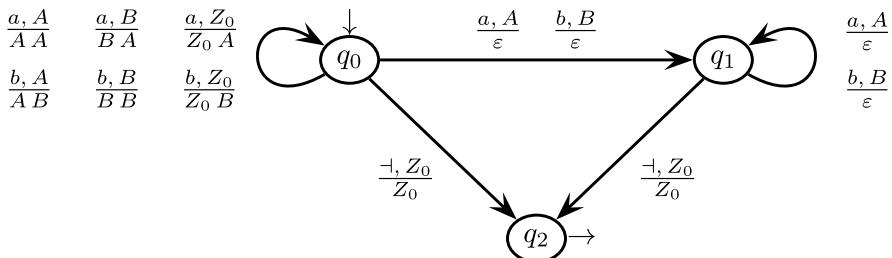
The language recognized is the set of accepted strings.

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 finite automaton can be graphically presented, 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 $L = \{uu^R \mid u \in \{a, b\}^*\}$ of the palindromes (p. 30) of even length is accepted with final state by the pushdown recognizer



The stack alphabet has three symbols: the initial symbol Z_0 ; and symbols A , B , respectively, indicating that a character a or b has been read. For instance, the arc $q_0 \xrightarrow{\frac{a, B}{B, A}} q_0$ denotes the reading move $(q_0, BA) \in \delta(q_0, a, B)$.

A non-deterministic behavior occurs in the state q_0 when reading character a from input and symbol A from stack: the machine may stay in q_0 and push a pair AA , or go to q_1 and push nothing.

In Fig. 4.2 we trace two among other possible computations for string $x = aa$. Since the computation (right) entirely reads string aa and reaches a final state, the string is accepted. Another example is the empty string, 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 in two aspects from the one obtained from a grammar by the mapping of Table 4.1 on p. 144:

stack	inp. x	state	comment	stack	inp. x	state	comment
Z_0	$a\ a$	\dashv	q_0	Z_0	$a\ a$	\dashv	q_0
$Z_0\ A$	a	\dashv	q_0	$Z_0\ A$	a	\dashv	q_0
$Z_0\ A\ A$		\dashv	q_0	Z_0		\dashv	q_1
			failure: no move is defined for (q_0, \dashv, A)	Z_0		\dashv	q_2
			recognition with final state				

Fig. 4.2 Two computations of Example 4.4 for input $x = aa$

it performs state transitions and checks if the current state is final to recognize a string. These and other differences are discussed next.

Accepting Modes Two different manners 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 *with final state* disregards the stack content; on the other hand, the latter mode with *empty stack* disregards the current state of the machine.

The two modes can also be combined into recognition with *final state and empty stack*. A natural question is whether these and other acceptance modes are equivalent.

Property 4.5 For the family of non-deterministic pushdown automata, the acceptance modes

- with empty stack
- with final state
- combined (empty stack and final state)

have the same capacity with respect to language recognition.

The statement says that any one of the three acceptance modes can be simulated by any other. In fact, if the automaton recognizes with final state, it is easy to modify it by adding new states so that it recognizes with empty stack: simply, when the original machine enters a final state, the second machine enters a new state and there stays while it empties the stack by spontaneous moves until the bottom symbol pops up.

Vice versa, to convert an automaton recognizing with 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.

Spontaneous Loops and On-line Functioning In principle an automaton can perform an unbounded series of spontaneous moves, i.e., a long computation without reading the input. When this happens we say the machine has entered a *spontaneous loop*. Such a loop may cause the machine not to read entirely the input, or it may trigger an unbounded computation after the input has been entirely scanned before deciding for acceptance or not. Both situations are in some sense pathological, as it would be for a program to enter a never-ending loop; they can be eliminated from pushdown machines without loss of generality.

The following reasoning² outlines how to construct an equivalent machine, free from spontaneous loops. Moreover, such a machine will always scan the entire source string and immediately halt after reading the last character.

First, we can easily modify the given machine by introducing a sink state, so that it never stops until the input is entirely read.

Suppose this machine has a spontaneous loop, i.e., a computation that can be repeated forever. By necessity this computation visits a configuration with state p and stack γA , such that the automaton can perform unboundedly many spontaneous moves without consuming the top stack symbol A . Then we modify the machine and add two new states as next explained. If during the spontaneous loop the automaton does not enter a final configuration, we add the new error state p_E and the move:

$$p \xrightarrow[\frac{A}{A}]{} p_E$$

Otherwise we add the new final state p_F and the moves:

$$p \xrightarrow[\frac{A}{A}]{} p_F \quad p_F \xrightarrow[\frac{A}{A}]{} p_E$$

To complete the conversion, the error state p_E should be programmed to consume the remaining suffix of the input string.

A machine is said to function *on line* if upon reading the last input character, it decides at once to accept or reject the string without any further computation.

Any pushdown machine can always be transformed so as to work on line. Since we know that spontaneous loops can be removed, the only situation to be considered is when the machine, after reading the last character, enters state p and performs a finite series of moves. Such moves will examine a finite top-most stack segment of maximal length k before accepting or rejecting; the segment is accordingly qualified as accepting or rejecting. Since the stack segment is finite, the corresponding information can be stored in the state memory. This requires to multiply the states, so that in any computation the top-most stack segment of length k is also represented in the current state.

Whenever the original machine entered state p , the second machine will enter a state that represents the combination of p and the stack segment. If the segment of the original machine is accepting then the second machine accepts as well, otherwise it rejects the input.

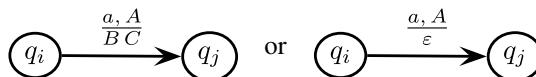
²For a thorough discussion of this point and the next one, we refer to, e.g., [18, 29].

4.3 One Family for Context-Free Languages and Pushdown Automata

We are going to show that the language accepted by a pushdown machine using states is context-free. Combined with the property (p. 145) that a context-free language can be recognized by a pushdown machine, this leads to the next central characterization of context-free languages, analogous to the characterization of regular languages by finite automata.

Property 4.6 The family CF of context-free languages coincides with the set of the languages recognized by pushdown automata.

Proof Let $L = L(M)$ be the language recognized by a pushdown machine M with the following stipulations: it 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, C are stack symbols. Thus a move pushes either two symbols or none. It turns out that this assumption does not reduce the generality of the machine. The initial stack symbol and state are Z and q_0 , respectively.

We are going to construct a grammar G equivalent to this machine. The construction may produce useless rules that later can be removed by cleaning. In the resulting grammar the axiom is S and all other nonterminal symbols are formed by a 3-tuple containing two states and a stack symbol, 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 (Table 4.1, p. 144) created a nonterminal for each stack symbol. But 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. From nonterminal $\langle q_i, A, q_j \rangle$ string z derives if, and only if, the automaton starting in state q_i with A on top of the stack performs a computation that reads string z , enters state q_j and deletes A from the stack. According to this principle, the axiom rewriting rules of the grammar have the form

$$S \rightarrow \langle q_0, Z, q_f \rangle$$

where Z is the initial stack symbol, and q_0, 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 to rule $\langle q_i, A, q_j \rangle \rightarrow a$.

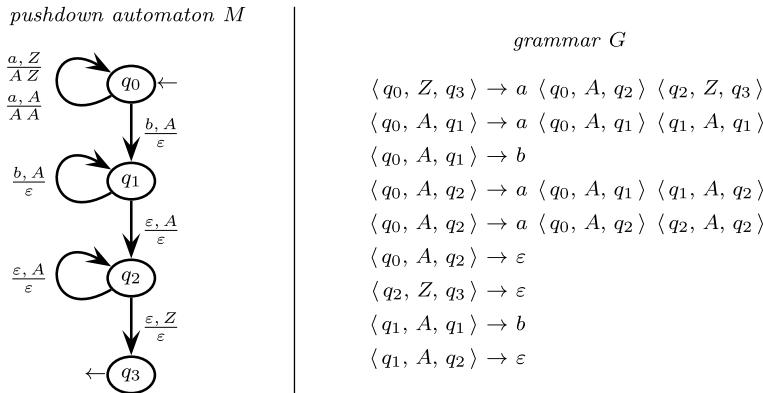


Fig. 4.3 Pushdown automaton (left) and grammar (right) for Example 4.7

2. Moves of the form $\overset{a, A}{\underset{B C}{\longrightarrow}} q_i \rightarrow q_j$ where the two states may coincide, are converted to the set of rules:

$$\{ \langle q_i, A, q_x \rangle \rightarrow a \langle q_j, B, q_y \rangle \langle q_y, C, q_x \rangle \mid \text{for all states } q_x \text{ and } q_y \text{ of } M \}$$

Notice that the non-empty grammar rules obtained by points (1) and (2) above are in the Greibach normal form. \square

We omit the correctness proof ³ of the construction and move to an example.

Example 4.7 (Grammar equivalent to a pushdown machine.⁴) The language L

$$L = \{ a^n b^m \mid n > m \geq 1 \}$$

is accepted by the pushdown automaton M in Fig. 4.3. The automaton reads a character a and 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).

The grammar rules are listed under the automaton: notice the axiom is $\langle q_0, Z, q_3 \rangle$. The useless rules created by step (2), such as

$$\langle q_0, A, q_1 \rangle \rightarrow a \langle q_0, A, q_3 \rangle \langle q_3, A, q_1 \rangle$$

which contains the undefined nonterminal $\langle q_0, A, q_3 \rangle$, are not listed.

³See for instance [18, 23, 24, 34].

⁴This example has been prepared using *JFLAP*, the formal language and automata package of Rodger and Finley [32].

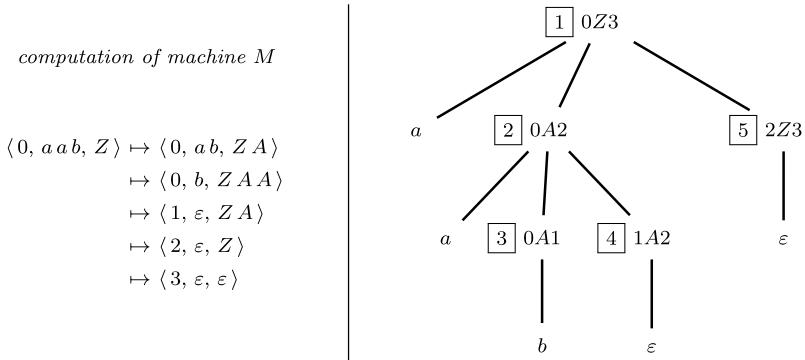


Fig. 4.4 A computation of machine M (left) and the corresponding derivation tree for Example 4.7 (right)

To understand the mapping between the two models, it helps to compare the computation and the leftmost derivation:

$$\langle 0, aab, Z \rangle \xrightarrow{5} \langle 3, \epsilon, \epsilon \rangle \quad \langle q_0, Z, q_3 \rangle \xrightarrow{5} aab$$

The computation and the corresponding derivation with steps numbered are in Fig. 4.4, where the nonterminal names are simplified to, say, $0Z3$. The following properties hold at each computation and derivation step:

- the string prefixes read by the machine and generated by the derivation are identical
- the stack contents are the mirror of the string obtained by concatenating the middle symbols (A, Z) of each 3-tuple occurring 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 the same arrow style as in the line

$$0Z3 \stackrel{+}{\Rightarrow} aa \downarrow \downarrow \downarrow \downarrow 0A1 \quad 1A2 \quad 2Z3$$

Such a stepwise correspondence between transition and derivation, ensures the two models define the same language.

4.3.1 Intersection of Regular and Context-Free Languages

As an illustration of previous results, we prove the property stated in Chap. 2 at Table 2.8 on p. 78: 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 to recognize the language $L(G) \cap L(A)$.

First we construct the one-state automaton N that recognizes the language $L(G)$ with empty stack, by using the simple method on p. 144. Then we construct the Cartesian product machine M , to simulate the respective computations of machines

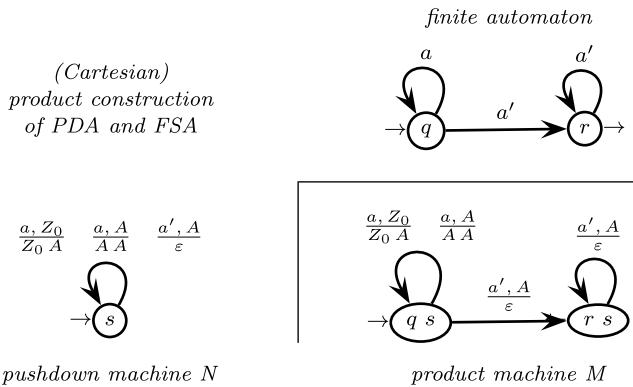


Fig. 4.5 Product machine for the intersection of the Dyck language and the regular language $a^*a'^+$ (Example 4.8)

N and A . The construction is essentially the same explained for two finite machines (p. 134), with the difference of the presence of a stack. The state set is the Cartesian product of the state sets of the component machines. The product machine M performs the same operations on the stack as the component machine N does. Recognition is with final state and empty stack; the final states of M are those containing a final state of the finite machine A . 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 with 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 4.8 (Intersection of a context-free and a regular language) We want (as in Example 2.85 on p. 79) the intersection of the Dyck language with alphabet $\Sigma = \{a, a'\}$ and the regular language $a^*a'^+$. The result is the language $\{a^n a'^m \mid n \geq 1\}$.

It is straightforward to imagine a pushdown machine with one state that accepts the Dyck language with empty stack. This Dyck machine, the finite automaton, and the resulting product machine 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 $\{r, s\}$ associated with a reading move of a' , operates on the stack exactly as automaton N : it pops A and goes from state q to state r exactly as the finite automaton. Since the component pushdown machine accepts with empty stack and the finite machine recognizes in final state r , the product machine recognizes with empty stack in final state (r, s) .

4.3.2 Deterministic Pushdown Automata and Languages

It is important to further the study of deterministic recognizers and corresponding languages because they are widely adopted in compilers thanks to their computa-

tional efficiency. Observing a pushdown machine (as defined on p. 146), we may find three forms of non-determinism, namely uncertainty between:

1. reading moves, if for a state q , a character a and a stack symbol A , the transition function has two or more values: $|\delta(q, a, A)| > 1$
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: $|\delta(q, \varepsilon, A)| > 1$

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 4.9 (Forms of non-determinism) The one-state recognizer (p. 144) of the language $L = \{a^n b^m \mid n \geq m > 0\}$ is non-deterministic of form (1):

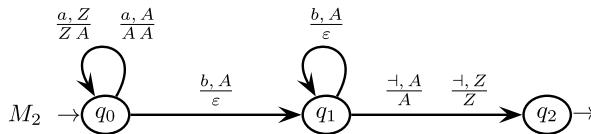
$$\delta(q_0, a, A) = \{(q_0, b), (q_0, bA)\}$$

and also of form (2):

$$\delta(q_0, \varepsilon, S) = \{(q_0, A)\} \quad \delta(q_0, a, S) = \{(q_0, S)\}$$

The same language is accepted by the deterministic machine:

$$M_2 = (\{q_0, q_1, q_2\}, \{a, b\}, \{A, Z\}, \delta, q_0, Z, \{q_2\})$$



Intuitively, the machine M_2 pushes every incoming character a , encoded as A . Upon reading the first character b , it pops an A and goes to state q_1 . Then it pops an A for any character b found in the input. If there are more characters b than a , the computation ends in error. Upon reading the end-marker, the machine moves to final state q_2 irrespectively of the stack top symbol.

Although we have been 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 subfamily of *CF*, as we shall see.

4.3.2.1 Closure Properties of Deterministic Languages

Deterministic languages are a subclass of context-free languages and have specific properties. Starting from the known properties (Table 2.8 on p. 78), we list the properties of deterministic languages in Table 4.3. We symbolize a language belonging to the family *CF*, *DET*, *REG* with L , D , R , respectively.

Table 4.3 Closure properties of families DET , REG and CF

Operation	Property	(Already known property)
Reversal	$D^R \notin DET$	$D^R \in CF$
Star	$D^* \notin DET$	$D^* \in CF$
Complement	$\neg D \in DET$	$\neg L \notin CF$
Union	$D_1 \cup D_2 \notin DET$ $D \cup R \in DET$	$D_1 \cup D_2 \in CF$
Concatenation	$D_1 \cdot D_2 \notin DET$ $D \cdot R \in DET$	$D_1 \cdot D_2 \in CF$
Intersection	$D \cap R \in DET$	$D_1 \cap D_2 \notin CF$

Next we argue for the listed properties and support them by examples.⁵
reversal (or mirror) The language

$$L = \{a^n b^n e\} \cup \{a^n b^{2n} d\} \quad \text{for } n \geq 1$$

satisfies the two equalities: $|x|_a = |x|_b$ if the sentence ends with e ; and $2|x|_a = |x|_b$ if it ends with d . The 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.

complement The complement of a deterministic language is deterministic. The proof (similar to the one for regular languages on p. 132) constructs the recognizer of the complement by creating a new sink state and interchanging final non-final states.⁶ It follows that if a context-free language has a non-context-free one as complement, it cannot be deterministic.

union Example 4.12 on p. 158 shows the union of deterministic languages (as L' and L'' obviously are) is in general non-deterministic.

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 language (regular) is deterministic (regular); and the intersection of a deterministic language and a regular one is deterministic (discussed next).

intersection $D \cap R$ The Cartesian product of a deterministic pushdown machine and a deterministic finite automaton is deterministic. To show the intersection of two deterministic languages may go out of DET , recall the language with three equal exponents (p. 76) is not context-free; but it can be defined by the intersection of two languages, both deterministic:

$$\{a^n b^n c^n \mid n \geq 0\} = \{a^n b^n c^* \mid n \geq 0\} \cap \{a^* b^n c^n \mid n \geq 0\}$$

⁵It may be superfluous to recall that a statement such as $D^R \notin DET$ means there exists some language D such that D^R is not deterministic.

⁶See for instance [18, 23]; alternatively there is a method [21] for transforming the grammar of a deterministic language to the grammar of its complement.

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 language. Therefore it is unable to decide the point for switching from the first to the second transition function. As an example, take the languages

$$L_1 = \{a^i b^j a^j \mid i, j \geq 1\} \quad L_2 = \{a^i b^j a^j \mid i, j \geq 1\} \quad \text{and} \quad R = \{c, c^2\}$$

and notice the language $L = c \cdot L_1 \cup L_2$ is deterministic, but the concatenation $R \cdot L$ is not.⁷ In fact, the presence of a prefix cc can be alternatively interpreted as coming either from $c \cdot c \cdot L_1$ or from $c^2 \cdot L_2$.

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.

Table 4.3 witnesses 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, there is no guarantee the result will be deterministic (it may even be ambiguous). The same danger threatens the concatenation and 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 of *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.⁸

4.3.2.2 Non-deterministic Languages

Unlike regular languages, there exist context-free languages that cannot be accepted by a deterministic automaton.

Property 4.10 The family *DET* of deterministic languages is strictly included in the family *CF* of context-free languages.

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 non-deterministic one; second, we have $DET \neq CF$ 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 non-deterministic context-free languages in order to evidence some language paradigms that ought to be carefully avoided by language designers.

⁷The proof can be found in [18].

⁸The algorithm is fairly complex, see [36].

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 $x \in D$ a sentence, and suppose there exists another sentence $y \in D$ that is a prefix of x , i.e., we have $x = yz$, where any strings may be empty.

Now we define another language called the *double service* of D , obtained by inserting a new terminal—the sharp sign \sharp —between strings y and z :

$$ds(D) = \{y\sharp z \mid y \in D \wedge z \in \Sigma^* \wedge yz \in D\}$$

For instance, for language $F = \{a, ab, bb\}$ the double service language is:

$$ds(F) = \{a\sharp b, a\sharp, ab\sharp, bb\sharp\}$$

Notice the original sentences are terminated by \sharp and may be followed or not by a suffix. It holds $D\sharp \subseteq ds(D)$ and in general the containment is strict.

Lemma 4.11 *If language D is deterministic, then also its double service language $ds(D)$ is deterministic.*

Proof ⁹ We are going to transform the deterministic recognizer M of D into a deterministic one M' of the double service language of D . To simplify the construction, we assume automaton M functions *on line* (p. 150). 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 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 in order to recognize string yz . \square

The reason for the curious name of the language is now clear: the automaton simultaneously performs two services inasmuch as it has to recognize a prefix and a longer string.

The lemma has the practical implication that if the double service of a *CF* language L is not *CF*, then the language L itself is not deterministic.

Example 4.12 (Non-deterministic union of deterministic languages) The language

$$L = \{a^n b^n \mid n \geq 1\} \cup \{a^n b^{2n} \mid n \geq 1\} = L' \cup L''$$

which is the union of two deterministic languages, is not deterministic.

⁹See Floyd and Beigel [15] for further reading.

Intuitively the automaton has to read and store one or more characters a on the stack. Then if the string (e.g., $aabb$) is in the first set, it must pop an a upon reading a b ; but if the string is in the second set (e.g., $aabbbb$), it must read two letters b before popping one a . Since 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 non-deterministically.

More rigorously, assume by contradiction that language L is deterministic. Then also its double service language $ds(L)$ would be deterministic, and from the closure property of DET (p. 155) under intersection with regular languages, the language:

$$L_R = ds(L) \cap (a^+ b^+ \# b^+)$$

is deterministic, too. But language L_R is not context-free, hence certainly not a deterministic one because its sentences have the form $a^i b^i \# b^i$, for $i \geq 1$, with three equal exponents (p. 76), a well-known non-context-free case.

The next example applies the same method to the basic paradigm of palindromic strings.

Example 4.13 (Palindromes) The language L of palindromes is defined by grammar:

$$S \rightarrow aSa \mid bSb \mid a \mid b \mid \varepsilon$$

To prove that language L is not deterministic, we intersect its language of double service with a regular one, with the aim of obtaining a language that is not in CF . Consider the language:

$$L_R = ds(L) \cap (a^* ba^* \# ba^*)$$

A string of the form $a^i b a^j \# b a^k$ is in L_R if, and only if, condition $j = i \wedge k = i$ holds. But then this language L_R is again the well-known non- CF one with three equal exponents.

4.3.2.3 Determinism and Unambiguity of Language

If a language is accepted by a deterministic automaton, each sentence is recognized with exactly one computation and it is provable that the language is generated by a non-ambiguous grammar.

The construction of p. 151 produces a grammar equivalent to a pushdown automaton, which simulates computations by 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 4.14 Let M be a deterministic pushdown machine; then the corresponding grammar of $L(M)$ obtained by the construction on p. 151 is not ambiguous.

But of course other grammars of $L(M)$ may be ambiguous.

A consequence is that any inherently ambiguous context-free language is non-deterministic: suppose by contradiction it is deterministic, then the preceding property states a non-ambiguous equivalent grammar does exist. This contradicts the very definition of inherent ambiguity (p. 53) that every grammar of the language is ambiguous. In other words, an inherently ambiguous language cannot be recognized by a deterministic pushdown machine. The discussion confirms what has been already said about the irrelevance of inherently ambiguous languages for technical applications.

Example 4.15 (Inherently ambiguous language and indeterminism) The inherently ambiguous language of Example 2.56 (p. 53) is the union of two languages:

$$L_A = \{a^i b^i c^* \mid i \geq 0\} \cup \{a^* b^j c^i \mid i \geq 0\} = L_1 \cup L_2$$

which are both deterministic (by the way another proof that family DET is not closed by union).

Intuitively, to recognize language L_A two different strategies must be adopted for the strings of each language. In the former the letters a are pushed onto the stack and popped from it upon reading b ; the same happens in the latter case but for letters b and c . Therefore any sentence belonging to both languages is accepted by two different computations and the automaton is non-deterministic.

The notion of ambiguity applies to any kind of automata. An *automaton* is *ambiguous* if a sentence exists, such that it is recognized by two distinct computations.

We observe the condition of determinism 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 4.16 (Relation between nondeterminism and unambiguity) The language L_A of the previous example is non-deterministic and also ambiguous because certain sentences are necessarily accepted by different computations. On the other hand, the language of Example 4.12 on p. 158:

$$L_I = \{a^n b^n \mid n \geq 1\} \cup \{a^n b^{2n} \mid n \geq 1\} = L' \cup L''$$

although non-deterministic (as argued there) is unambiguous. In fact, each one of the two sub-languages L' and L'' is easily defined by an unambiguous grammar, and the union remains unambiguous because the components are disjoint. The two recognition strategies described on p. 158 are implemented by distinct computations, but one at most of them may succeed for any given string.

4.3.2.4 Subclasses of Deterministic Pushdown Automata

The family DET is by definition 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 have no consequence in the indeterministic case, cause a restriction of the language family recognized by a deterministic machine. The main cases are briefly mentioned.¹⁰

automaton with only one state Acceptance is necessarily with empty stack and it is less powerful than recognition with final state.

limitation on the number of states The family of languages 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, i.e., without spontaneous moves It restricts the family of recognizable languages.

4.3.2.5 Some Simple Deterministic Subfamilies

In many practical situations, technical languages are designed so that they are deterministic. For instance, this is the case for almost all programming languages and for the family *XML* of mark-up languages of the Internet. Different approaches exist for ensuring that a language is deterministic by imposing some conditions on the language or on its grammar. Depending on the conditions, one obtains different subfamilies of *DET*. Two simple cases are next described to introduce the topic; others, much more important for applications, will be obtained with the *LL(k)* and *LR(k)* conditions.

Simple Deterministic Languages A grammar is called *simple deterministic* if it satisfies the next conditions:

1. every right part of a rule starts with a terminal character (hence there are no empty rules)
2. for any nonterminal A , alternatives do not exist starting with the same character, i.e.

$$\nexists A \rightarrow a\alpha \mid a\beta \quad \text{with} \quad a \in \Sigma \quad \alpha, \beta \in (\Sigma \cup V)^* \quad \alpha \neq \beta$$

An example is the simple deterministic grammar $S \rightarrow aSb \mid c$.

Clearly, if we construct the pushdown machine from a simple deterministic grammar by means of the method in Table 4.1 on p. 144, the result is a deterministic machine. Moreover, this machine consumes a character with each move, i.e., it works in real-time, which follows from the fact that such grammars are (almost) in the Greibach normal form.

We hasten to say that anyway the simple deterministic condition is too inconvenient for a practical use.

Parenthesis Languages It is easy to check that the parenthesized structures and languages, introduced in Chap. 2 on p. 41, are deterministic. 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. We assume the grammar is *distinctly*

¹⁰For further reading, see [18, 37].

parenthesized (p. 43); otherwise, if just one type of parentheses is used, we assume rules do not exist with identical right-hand sides. Either assumption ensures that the grammar is unambiguous.

A simple recognition algorithm scans the string and localizes a substring that does not contain parentheses and is enclosed between a matching pair of parentheses. Then the relevant set of the right parts of grammar rules is searched for the parenthesized substring. If none matches, the source string is rejected. Otherwise the parenthesized substring is reduced to the corresponding nonterminal (the left part of the matching rule), thus producing a shorter string to be recognized. Then the algorithm resumes scanning the new string in the same way, and finally it recognizes when the string is reduced to the axiom. It is not difficult to encode the algorithm by means of a deterministic pushdown machine.

If a grammar defines a non-deterministic language or if it is ambiguous, the transformation into a parenthesized grammar removes both defects. For instance, the language of palindromes $S \rightarrow aSa | bSb | a | b | \varepsilon$ on p. 159 is changed into a deterministic one when every right part of a rule is parenthesized:

$$S \rightarrow (aSa) | (bSb) | (a) | (b) | ()$$

Web documents and semi-structured databases are often encoded in the mark-up language *XML*. Distinct opening and closing marks are used to delimit the parts of a document in order to allow an efficient recognition and transformation. Therefore the *XML* language family is deterministic. The *XML* grammar model¹¹ (technically known as Document Type Definition or *DTD*) is similar to the case of parenthesized grammars, but as a distinguishing feature it uses various regular expression operators in the grammar rules.

4.4 Syntax Analysis

The rest of the chapter makes the distinction between top-down and bottom-up sentence recognition, and systematically covers the practical parsing algorithms used in the compilers, also for grammars 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 the error was detected (diagnosis); afterwards 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 the pushdown machine performs a move corresponding to a grammar rule, it has to save

¹¹Such *DTD* grammars generate context-free languages, but differ from context-free grammars in several ways; see [8] for an initial comparison.

the rule label into some kind of data structure. Upon termination, the data structure will represent the syntax tree.

If the source string is ambiguous, the result of the analysis is a set of trees. In that case the parser may decide to stop when the first derivation has been found, or to exhaustively produce all of them.

4.4.1 Top-Down and Bottom-Up Constructions

We know the same (syntax) tree corresponds to many derivations, notably the leftmost and rightmost ones, as well as to less relevant others that proceed in a zigzag order. Depending on the derivation being leftmost or rightmost and on the order it is constructed, we obtain two important parser classes.

top-down analysis Constructs the leftmost derivation by starting from the axiom, i.e., the root of the tree, and growing the tree towards the leaves; each step of the algorithm 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 explained by an example.

Example 4.17 (Tree visit orders) Consider the grammar

$$\left\{ \begin{array}{ll} 1 : S \rightarrow aSAB & 2 : S \rightarrow b \\ 3 : A \rightarrow bA & 4 : A \rightarrow a \\ 5 : B \rightarrow cB & 6 : B \rightarrow a \end{array} \right.$$

and the sentence $a^2b^3a^4$. The orderings corresponding to the top-down and bottom-up visits are shown in Fig. 4.6. In each node the framed number gives the visit order and the subscript of the nonterminal indicates the grammar rule applied. A top-down analyzer grafts the right part of a rule under the corresponding left part, i.e., left nonterminal. If the right part contains terminal symbols, these have to orderly match the characters of the source string. The procedure terminates when all nonterminal symbols have been transformed into terminal ones (or into empty strings).

On the other hand, starting with the source text, a bottom-up analyzer reduces to a nonterminal the substrings matching the right part of a rule, as they are encountered in a left-to-right scan. After each reduction, the modified text is again scanned to find the next reduction until the text is reduced to the axiom.

The dichotomy bottom-up/top-down can be made to apply to extended BNF grammars, too, and we postpone the discussion to later sections. In principle, both analysis approaches also work from right to left, by scanning the reversed source string, but in reality all existing languages are designed for left-to-right processing,

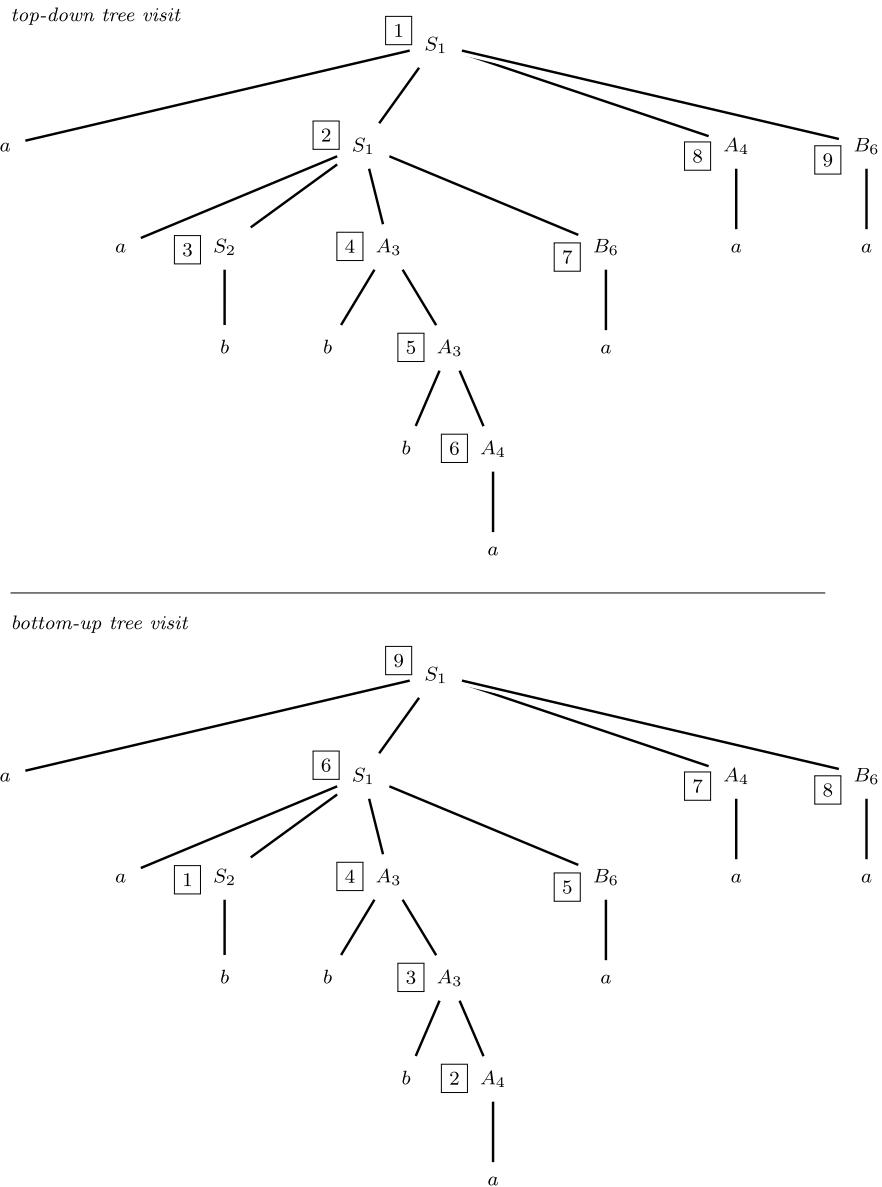


Fig. 4.6 Left-to-right visit orders of a syntax tree for Example 4.17

as happens in the natural language where the reading direction corresponds to the uttering order by a speaker. Incidentally, reversing the scanning direction may damage the determinism of a language because the *DET* family is not closed by reversal (p. 156).

Finally, the general adoption of parallel processing technology motivates the study of new algorithms that would parse a long text—such as a web page source—by means of multiple 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, permits to directly handle grammars with regular expressions, and maps quite nicely on a recursive-descent parser implementation.

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* (or *EBNF*) of p. 82, the right part of a rule may contain the union operator, which makes it possible for each nonterminal to be defined by just one rule like $A \rightarrow \alpha$, where the right α is an r.e. over the alphabet of terminals and nonterminals. The r.e. α defines a regular language, whose finite recognizer M_A can be easily constructed with 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 $B \rightarrow \beta$. Notice that the symbol B may coincide with A , 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 that the machines be deterministic; if not, they can be made deterministic by the methods of Chap. 3.

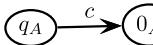
Definition 4.18 (Recursive net of finite deterministic machines)

- Let Σ and $V = \{S, A, B, \dots\}$ be, respectively, the terminal alphabet and non-terminal alphabet, and S be the axiom of an extended context-free grammar G .
- For each nonterminal A there is exactly one (extended) grammar rule $A \rightarrow \alpha$ and the right part α of the rule is an r.e. over the alphabet $\Sigma \cup V$.
- Let the grammar rules be denoted by $S \rightarrow \sigma$, $A \rightarrow \alpha$, $B \rightarrow \beta$, The symbols R_S, R_A, R_B, \dots denote the regular languages over the alphabet $\Sigma \cup V$, defined by the r.e. $\sigma, \alpha, \beta, \dots$, respectively.
- The symbols M_S, M_A, M_B, \dots are the names of the (finite deterministic) machines accepting the corresponding regular languages R_S, R_A, \dots . The set 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 $Q_A = \{0_A, \dots, q_A, \dots\}$, its only initial state is 0_A and

its set of final states is $F_A \subseteq Q_A$. The *state set* Q of a net \mathcal{M} is the union of all states:

$$Q = \bigcup_{M_A \in \mathcal{M}} Q_A$$

The transition function of all 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 as the machine state sets are all disjoint.

- For a state q_A , the symbol $R(M_A, q_A)$ or for brevity $R(q_A)$, denotes the regular language over the alphabet $\Sigma \cup V$, accepted by the machine M_A starting from state q_A . For the initial state, we have $R(0_A) \equiv R_A$.
- It is convenient to stipulate that for every machine M_A , there is no arc as  with $c \in \Sigma \cup V$, which enters the initial state 0_A . Such a normalization ensures that the initial state is not visited twice, i.e., it is not recirculated or re-entered, within a computation that does not leave machine M_A .

We observe that a rule $A \rightarrow \alpha$ is adequately represented by machine M_A , due to the one-to-one mapping from the strings over $\Sigma \cup V$ defined by the r.e. α , and the paths of machine M_A that go from the initial state to a final one.¹² Therefore a machine net $\mathcal{M} = \{M_S, M_A, \dots\}$ 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 the alphabet Σ) defined (or recognized) by the machine net, coincides with the language generated by the grammar, i.e., with $L(G)$.

We need to consider also the terminal language defined by a generic machine M_A , when starting from a state possibly other than the initial one. For any state q_A , not necessarily initial, we write as

$$L(M_A, q_A) = L(q_A) = \{y \in \Sigma^* \mid \eta \in R(q_A) \wedge \eta \xrightarrow{*} y\}$$

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

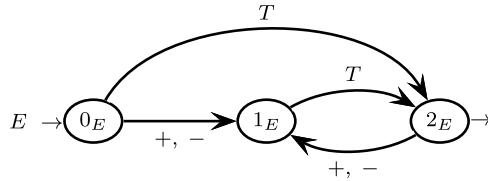
$$L(M_A, 0_A) = L(0_A) \equiv L_A(G)$$

and for the axiom it is:

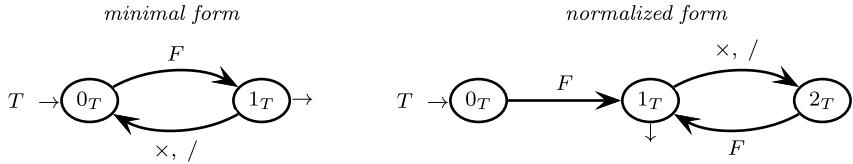
$$L(M_S, 0_S) = L(0_S) = L(\mathcal{M}) \equiv L(G)$$

¹²Of course two different but equivalent r.e. can be represented by the same normalized machine (or by equivalent machines). A machine net represents all grammars having the same non-terminals S, A, B, \dots and different though equivalent rules $S \rightarrow \sigma, \dots$ versus $S \rightarrow \sigma', \dots$ with $R(\sigma) = R(\sigma')$. Any one of them can be taken whenever we talk of a grammar corresponding to \mathcal{M} .

machine M_E



machine M_T



machine M_F

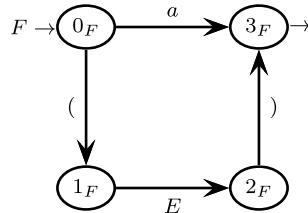


Fig. 4.7 Machine net of Example 4.19; machine M_T is shown in the minimal form (left) and after normalizing its initial state (right)

Example 4.19 (Machine net for arithmetic expressions) The EBNF grammar below has alphabet $\{+, -, \times, /, a, '(', '\)'\}$, three nonterminals E, T and F (axiom E), and therefore three rules:

$$\begin{cases} E \rightarrow [+ | -]T((+ | -)T)^* \\ T \rightarrow F((\times | /)F)^* \\ F \rightarrow a \mid ('(E ')') \end{cases}$$

Figure 4.7 shows the machines of net \mathcal{M} .

We list a few languages defined by the net and by component machines, to illustrate the definitions.

$$R(M_E, 0_E) = R(M_E) = [+ | -]T((+ | -)T)^*$$

$$R(M_T, 1_T) = R(1_T) = ((\times | /)F)^*$$

$$\begin{aligned} L(M_E, 0_E) &= L(0_E) = L_E(G) = L(G) = L(\mathcal{M}) \\ &= \{a, a + a, a \times (a - a), \dots\} \end{aligned}$$

EBNF grammar G (axiom E)

$$\Sigma = \{ a, '(', ')' \} \quad V = \{ E, T \} \quad P = \begin{cases} E \rightarrow T^* \\ T \rightarrow '(', E ')', \mid a \end{cases}$$

machine net \mathcal{M}

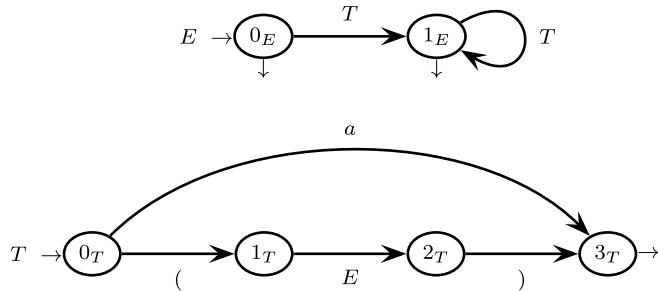


Fig. 4.8 EBNF grammar G and net \mathcal{M} of the running example (Example 4.20)

$$\begin{aligned} L(M_F, 0_F) &= L(0_F) = L_F(G) = \{a, (a), (-a + a), \dots\} \\ L(M_F, 2_F) &= L(2_F) = \{')'\} \end{aligned}$$

A few derivation examples (as defined in Chap. 2, p. 41) and the corresponding reductions are
derivations

$$T - T \Rightarrow F \times F \times F - T \quad \text{and} \quad T - T \xrightarrow{+} a \times F \times F - T$$

reductions

$$\begin{aligned} F \times F \times F - T \text{ reduces to } T - T &\quad \text{denoted } F \times F \times F - T \rightsquigarrow T - T \\ a \times F \times F - T &\xrightarrow{+} T - T \end{aligned}$$

which show how it works.

Example 4.20 (Running example) The grammar G and machine net \mathcal{M} are shown in Fig. 4.8. 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 re-entered. Most features needed to exercise different aspects of parsing are present: self-nesting, iteration, branching, multiple final states, and a nullable nonterminal.

To illustrate we list some of the languages defined by the machine net, along with their aliases:

$$R(M_E, 0_E) = R(0_E) = T^*$$

$$R(M_T, 1_T) = R(1_T) = E$$

$$L(M_E, 0_E) = L(0_E) = L(G) = L(\mathcal{M})$$

$$= \{\varepsilon, a, aa, (), aaa, (a), a(), ()a, ()(), \dots\}$$

$$L(M_T, 0_T) = L(0_T) = L_T(G) = \{a, (), (a), (aa), ((())), \dots\}$$

which show how it works.

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 have the following aliases:

$$\begin{array}{ll} 0_T \equiv T \rightarrow \bullet a \mid \bullet(E) & 2_T \equiv T \rightarrow (E \bullet) \\ 1_T \equiv T \rightarrow (\bullet E) & 3_T \equiv T \rightarrow a \bullet \mid (E) \bullet \end{array}$$

where the bullet \bullet is a character not in the terminal alphabet Σ .

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 shall 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 $A \rightarrow \alpha$. 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.

Example 4.21 (Syntax charts of arithmetic expressions (Example 4.19)) The machines of the net (Fig. 4.7) are redrawn as syntax charts in Fig. 4.9. The nodes are

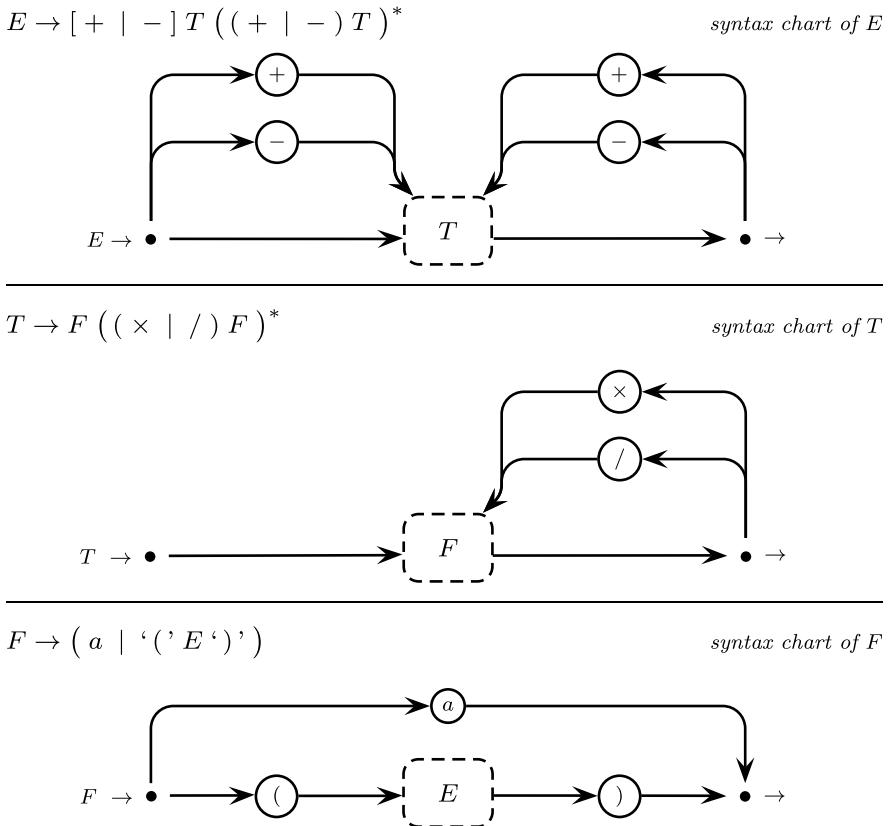


Fig. 4.9 Syntax charts equivalent to the machine net in Fig. 4.7

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:

$$T \quad + T \quad T + T \quad - T + T - T \quad \dots$$

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.

4.5.1 Derivation for Machine Nets

For *EBNF* grammars and machine nets, the preceding definition of derivation, which models a rule such as $E \rightarrow T^*$ as the infinite set of *BNF* alternative rules $E \rightarrow \varepsilon \mid T \mid TT \mid \dots$, has shortcomings because a derivation step, such as $E \Rightarrow TTT$, 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 application to parsing, 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 2.71 at p. 69) if every rule has the form $A \rightarrow uB$ or $A \rightarrow \varepsilon$, 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. 3.4.4 at p. 103) that has the machine states as nonterminal symbols.

4.5.1.1 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 $p_A \rightarrow Xr_A$ (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 $p_A \rightarrow \varepsilon$ exists.

Notice that X may be a nonterminal B , therefore a rule of \hat{G}_A can have the form $p_A \rightarrow Br_A$, 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 $L(\hat{G}_A) = R(M_A, 0_A)$ 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 $p_A \rightarrow Br_A$, thus obtaining rules of the form $p_A \rightarrow 0_Br_A$. The resulting *BNF* grammar is denoted \hat{G} and named the *right-linearized grammar* (*RLZ*) of the net: it has terminal alphabet Σ , nonterminal set Q and axiom 0_S .

The right parts have length zero or two, and may contain two nonterminal symbols, thus the grammar is *not RL*. Obviously, \hat{G} and G are equivalent, i.e., they both generate language $L(G)$.

Example 4.22 (Right-linearized grammar \hat{G} of the running example)

$$\hat{G}_E \quad \begin{cases} 0_E \rightarrow 0_T 1_E \mid \varepsilon \\ 1_E \rightarrow 0_T 1_E \mid \varepsilon \end{cases} \qquad \hat{G}_T \quad \begin{cases} 0_T \rightarrow a 3_T \mid ' (' 1_T \\ 1_T \rightarrow 0_E 2_T \\ 2_T \rightarrow ')' 3_T \\ 3_T \rightarrow \varepsilon \end{cases}$$

As said, in the right-linearized grammars we choose to name nonterminals by their alias states; an instance of non-*RL* rule is $1_T \rightarrow 0_E 2_T$.

Using the right-linearized grammar \hat{G} instead of G , we obtain derivations whose steps are elementary state transitions instead of multi-step transitions on a machine. An example should suffice to clarify.

Example 4.23 (Derivation of a right-linearized grammar) For grammar G the “classical” leftmost derivation:

$$E \xrightarrow[G]{ } TT \xrightarrow[G]{ } aT \xrightarrow[G]{ } a(E) \xrightarrow[G]{ } a() \quad (4.1)$$

is expanded into the series of truly atomic derivation steps of the right-linearized grammar \hat{G} :

$$\begin{aligned} 0_E &\xrightarrow[\hat{G}]{ } 0_T 1_E & \xrightarrow[\hat{G}]{ } a 1_E & \xrightarrow[\hat{G}]{ } a 0_T 1_E \\ &\xrightarrow[\hat{G}]{ } a(1_T 1_E) & \xrightarrow[\hat{G}]{ } a(0_E 2_T 1_E) & \xrightarrow[\hat{G}]{ } a(\varepsilon 2_T 1_E) \\ &\xrightarrow[\hat{G}]{ } a() 3_T 1_E & \xrightarrow[\hat{G}]{ } a() \varepsilon 1_E & \xrightarrow[\hat{G}]{ } a() \varepsilon \\ &= a() \end{aligned} \quad (4.2)$$

which show how it works.

We may also consider reductions, such as $a(1_T 1_E) \rightsquigarrow a 0_T 1_E$.

4.5.1.2 Call Sites and Machine Activation

An arc with nonterminal label, $q_A \xrightarrow[B]{ } r_A$, 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 activates the specified machine whenever it reaches a call site. Then the activated machine performs scanning operations and further calls on its own graph, until it reaches a final state; then 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.

Observing the left-linearized derivations, 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 4.24 (Derivation and machine return points) When looking at derivation (4.2), we find derivation $0_E \xrightarrow{*} a(0_E 2_T 1_E)$: 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 machine M_E was also suspended and will resume in state 1_E .

EBNF grammar

$$S \rightarrow A\ c \mid a\ B\ d \mid e\ A$$

$$A \rightarrow a\ b$$

$$B \rightarrow b$$

machine net

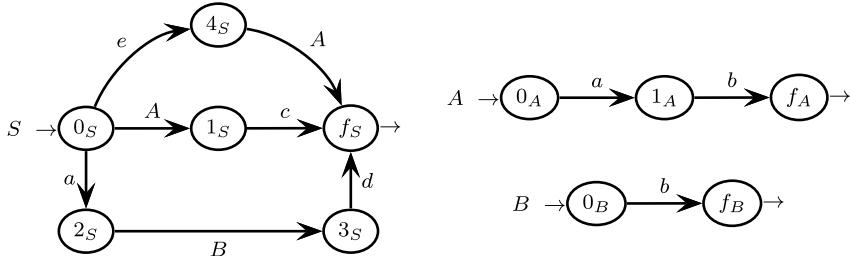


Fig. 4.10 Grammar and network for computing a few follow sets

4.5.2 Initial and Look-Ahead Characters

The deterministic parsers to be presented implement a pushdown automaton (*PDA*) and, to choose the next move, they use the stack contents (actually just the top symbol), the *PDA* state, and the next input character (the *token* or lexeme returned by the lexical analyzer). To make the choice deterministic, such parsers rely on information obtained by preprocessing the grammar, which consists of the sets of tokens that may be encountered for each possible move.

For instance in the current state p_A , suppose there are two outgoing arcs:

$$q_A \xleftarrow{B} p_A \xrightarrow{C} r_A$$

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 $A \rightarrow \alpha$ 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. 4.10.

After scanning substring ab , the *PDA* can be either in state f_A or in state f_B , and the reduction to be applied can be either $ab \rightsquigarrow A$ or $b \rightsquigarrow B$. How do we choose? By checking if the next input token is character c or d , respectively. To enable the *PDA* to perform such checks, we precalculate 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 are named *follow* sets.

However, the information provided by the follow sets can be made more accurate. We observe that nonterminal A cannot be followed by c if the input string is eab . 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 to each configuration, are as follows: set $\{c\}$ for nonterminal A with configuration 1_S ; set $\{\dashv\}$ for nonterminal A with configuration f_S ; while for the nonterminal B with configuration 3_S (the only existing one), the look-ahead set is $\{d\}$.

The sets of initial and look-ahead tokens are now formalized by means of procedural definitions.

Initials We need to define the set of the *initial* characters for the strings recognized starting from a given state.

Definition 4.25 Set Ini of initials.

$$Ini(q_A) = Ini(L(q_A)) = \{a \in \Sigma \mid a\Sigma^* \cap L(q_A) \neq \emptyset\}$$

Notice that set Ini cannot contain the null string ε . A set $Ini(q_A)$ is empty if, and only if, we have $L(q_A) = \{\varepsilon\}$ (the empty string is the only sentence).

Let symbol a be a terminal, symbols A, B be nonterminals, and states q_A, r_A be of the same machine. Set Ini can be computed by applying the following logical clauses until a fixed point is reached. It holds $a \in Ini(q_A)$ if, and only if, at least one of the following conditions holds:

$$\begin{aligned} & \exists \text{arc } q_A \xrightarrow{a} r_A \\ & \exists \text{arc } q_A \xrightarrow{B} r_A \wedge a \in Ini(0_B) \\ & \exists \text{arc } q_A \xrightarrow{B} r_A \wedge L(0_B) \text{ is nullable } \wedge a \in Ini(r_A) \end{aligned}$$

For the running example, we have $Ini(0_E) = Ini(0_T) = \{a, '(\cdot)\}$.

Look-Ahead As said, a set of look-ahead characters is associated to each PDA parser configuration. Without going into details, we assume that such configuration is encoded by one or more machine states. Remember that multiple 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 $\langle \text{state}, \text{token} \rangle$ is named a *candidate* (also known as *item*). More precisely a candidate is a pair $\langle q_B, a \rangle$ in $Q \times (\Sigma \cup \{\dashv\})$. 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 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 4.26 (Closure function) Let C be a set of candidates. The closure of C is the function defined by applying the following clause (4.3) until a fixed point is

reached:

$$\begin{aligned}
 K &:= C \\
 \textbf{repeat} \\
 K &:= K \cup \left\{ \langle 0_B, b \rangle \text{ such that } \exists \langle q, a \rangle \in K \text{ and} \right. \\
 &\quad \left. \exists \text{ arc } q \xrightarrow{B} r \text{ in } \mathcal{M} \text{ and } b \in \text{Ini}(L(r) \cdot a) \right\} \\
 \textbf{until} \text{ nothing changes} \\
 \text{closure}(C) &:= K
 \end{aligned} \tag{4.3}$$

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 such machine terminates; such a character is part of the look-ahead set. Notice the effect of term $\text{Ini}(L(r) \cdot a)$ when state r is final, i.e., when $\varepsilon \in L(r)$ holds: character a is included in the set.

When clause (4.3) terminates, the closure of C contains a set of candidates, with some of them possibly associated to the same state q , as follows:

$$\{\langle q, a_1 \rangle, \dots, \langle q, a_k \rangle\}$$

For conciseness we group together such candidates and we write

$$\langle q, \{a_1, \dots, a_k\} \rangle$$

instead of $\{\langle q, a_1 \rangle, \dots, \langle q, a_k \rangle\}$.

The collection $\{a_1, a_2, \dots, a_k\}$ 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 $\langle q, \{b\} \rangle$.

Example 4.27 We list a few values computed by the closure function for the grammar of Example 4.20 (Fig. 4.8 on p. 168):

C	Function closure (C)	
$\langle 0_E, \dashv \rangle$	$\langle 0_E, \dashv \rangle$	$\langle 0_T, \{a, '(', \dashv\} \rangle$
$\langle 1_T, \dashv \rangle$	$\langle 1_T, \dashv \rangle$	$\langle 0_E, ')' \rangle$ $\langle 0_T, \{a, '(', ')' \} \rangle$

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 (4.3) the closure contains the candidate with state 0_T and with look-ahead set computed as $\text{Ini}(L(1_E) \dashv)$. 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 $'('$. The fixed point is now reached since from 0_T no nonterminal arcs depart, hence further application of closure would be unproductive.

In the bottom row the result contains $\langle 1_T, \dashv \rangle$ by reflexivity, and $\langle 0_E, ')' \rangle$ because arc $1_T \xrightarrow{E} 2_T$ calls machine M_E and $Ini(L(2_T)) = \{')\}$. The result contains also $\langle 0_T, \{a, '(', ')' \} \rangle$ because arc $0_E \xrightarrow{T} 1_E$ calls machine M_T , language $L(2_T)$ is nullable and we have $\{a, '(', ')' \} = Ini(L(1_E)')$). Three applications of clause (4.3) 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 allowing such a parser to work was defined and named $LR(k)$ by its inventor Knuth [25]. 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 (in accordance with common use) to be $k = 1$, 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. 4.8.

The $ELR(1)$ parsers implement a deterministic automaton equipped with a push-down 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 (from p. 174). 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 *reduce* move applies as soon as the sequence of top-most 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 is permitted then a reduction is impossible. The second condition is that in every configuration at most one reduction 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 the tree is in general 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 top-most 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.

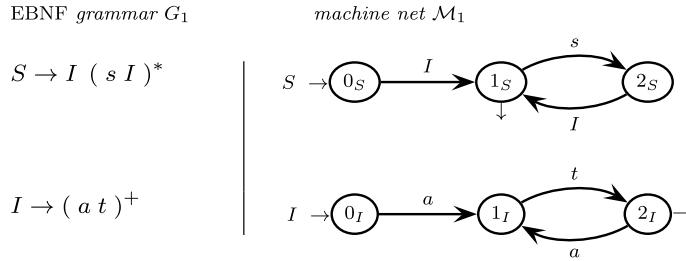


Fig. 4.11 Grammar G_1 and network \mathcal{M}_1 for Example 4.28 (two-level list)

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 a r.e. e as an *EBNF* grammar that has only one nonterminal and one rule $S \rightarrow e$. 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. 27) are a natural example, where at each level a different nonterminal specified by a r.e. generates a list.

Example 4.28 (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. 4.11.

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 $atatsat \dashv$. 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 non-terminal I . Therefore the automaton enters the initial state of machine M_I without reading 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 non-deterministic machine (Chap. 3, p. 117).

Table 4.4 Parsing trace of string $x = atatsat \dashv$ (Example 4.11)

stack bottom	string to be parsed (with end-marker) and stack contents							effect after	#			
	1	2	3	4	5	6	7					
{0 _S 0 _I }	a	t	a	t	s	a	t	⊣	initialization of the stack 1			
	a	t	a	t	s	a	t	⊣				
	a {1 _I }	t	a	t	s	a	t	⊣	terminal shift on a: 0 _I \xrightarrow{a} 1 _I 2			
	a	t	a	t	s	a	t	⊣				
	a {1 _I }	t {2 _I }	a	t	s	a	t	⊣	terminal shift on t: 1 _I \xrightarrow{t} 2 _I 3			
	a {1 _I }	t {2 _I }	a {1 _I }	t	s	a	t	⊣	(do not reduce) 4			
	a {1 _I }	t {2 _I }	a {1 _I }	t	s	a	t	⊣	terminal shift on a: 2 _I \xrightarrow{a} 1 _I 5			
	a {1 _I }	t {2 _I }	a {1 _I }	t {2 _I }	s	a	t	⊣	terminal shift on t: 1 _I \xrightarrow{t} 2 _I 6			
	I				s	a	t	⊣	since s follows I reduce atat ~ I			
	I				s	a	t	⊣	nonterminal shift on I: 0 _S \xrightarrow{I} 1 _S 7			
	I	{1 _S }			s	a	t	⊣	(do not reduce) 8			
	I	{1 _S }			s {2 _S 0 _I }			⊣	terminal shift on s: 1 _S \xrightarrow{s} 2 _S 9			
	I	{1 _S }			s {2 _S 0 _I }	a {1 _I }	t	⊣	terminal shift on a: 0 _I \xrightarrow{a} 1 _I 10			
	I	{1 _S }			s {2 _S 0 _I }	a {1 _I }	t {2 _I }	⊣	terminal shift on t: 1 _I \xrightarrow{t} 2 _I 11			
	I	{1 _S }			s {2 _S 0 _I }	I		⊣	since ⊣ follows I reduce at ~ I 12			
	I	{1 _S }			s {2 _S 0 _I }	I	{1 _S }	⊣	nonterminal shift on I: 2 _S \xrightarrow{I} 1 _S 13			
	S							⊣	since ⊣ follows I reduce IsI ~ S			
	the input string is reduced to the axiom S							⊣				
	the stack contains only the axiom {0 _S 0 _I }							⊣	stop and accept 14			

The PDA stores the current m-state $\{0_S, 0_I\}$ on the stack for re-entering it in the future, after a series of transitions. Then it performs a *shift* operation associated to 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 the 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 4.4. 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 in gray; 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: first, when there is a choice between a *shift* and a *reduction*, known as potential *shift-reduce conflict*; and 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 that the need occurs to choose between shift (of a) or reduction (to I) is 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. 174), 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 top-most 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 possibility:

bottom	top
<i>segment (handle) to be reduced by: $a \at a t \rightsquigarrow I$</i>	
$\{0_S \ 0_I\} \ a \ \{1_I\} \ t \ \{2_I\} \ a \ \{1_I\} \ t \ \{2_I\}$	

But more complex situations occur in other languages.

The reduction $at \rightsquigarrow I$ 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 non-terminal 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 4.4, 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.

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 the 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 $\dots \rightsquigarrow I$ is chosen; furthermore, the top-most stack segment congruent with a string in the language $L(0_I)$ is found to be string at (disregarding the intervening m-states in the stack).

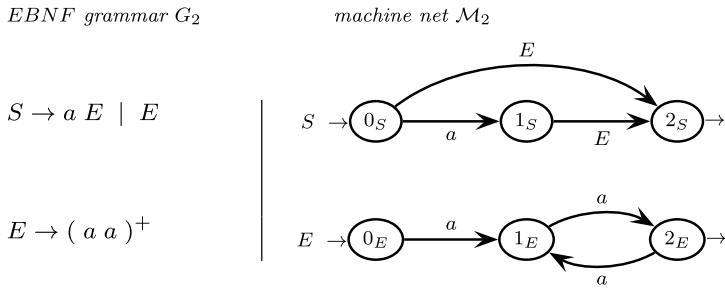
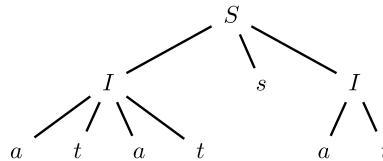


Fig. 4.12 Grammar G_2 and network \mathcal{M}_2 for Example 4.29

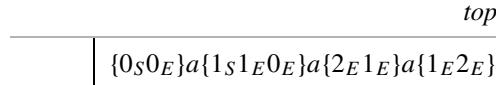
Step 12 shifts nonterminal I and produces a stack where a reduction is applied to nonterminal S . In the last PDA configuration the input has been entirely consumed and the stack contains only the initial m-state. Therefore the parser stops and accepts the string as valid, with the syntax tree



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.

Example 4.29 (Grammar requiring more information to locate handles) In this contrived example, machine M_S of net \mathcal{M}_2 in Fig. 4.12 recognizes by separate computations the strings $L(0_E) = (aa)^+$ of even length and those of odd length $a \cdot L(1_S)$. After processing string aaa , the stack configuration of the PDA is as



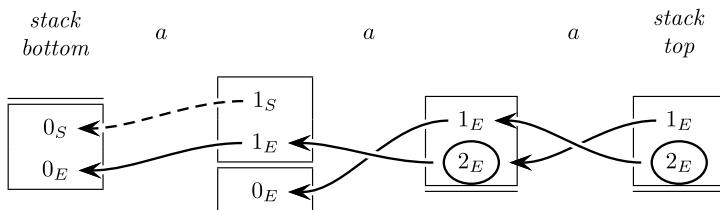
An essential ability of $ELR(1)$ parsers is to carry on multiple tentative analyzes at once. After shifting a , the presence of three states in m-state $\{1_S, 1_E, 0_E\}$ says that the PDA has found three different computations for analyzing the first token a :

$$0_S \xrightarrow{a} 1_S \quad 0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \quad 0_S \xrightarrow{a} 1_S \xrightarrow{\varepsilon} 0_E$$

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 $\{2_E, 1_E\}$ and $\{1_E, 2_E\}$ are identical. Let now \dashv be the next token; since state 2_E is final, a reduction $\dots \rightsquigarrow E$ 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 divisor 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 $aa \rightsquigarrow E$ 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 latter 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:

- 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).
- 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.
- 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 with handles of unbounded length.

Finally, 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 4.30) 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 $I_0 = \text{closure}(\langle 0_S, \dashv \rangle)$, where the closure function was defined on p. 174
- the m-state set $R = \{I_0, I_1, \dots\}$ and the state-transition function $\vartheta : R \times (\Sigma \cup V) \rightarrow R$ 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 :

$$\vartheta(C, X) = \bigcup_{\forall \text{ candidate } \gamma \in C} \vartheta(\gamma, X)$$

¹³For 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 that we denote by theta (to avoid confusion with the state-transition function of the network machine denoted delta) is traditionally named the “go to” function.

Algorithm 4.30 (Construction of the $ELR(1)$ pilot graph) Computation of the m-state set $R = \{I_0, I_1, \dots\}$ 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 (each m-state  $I \in R$  and symbol  $X \in \Sigma \cup V$ ) do
    -- compute the base of an m-state and its closure
     $I' := closure(\vartheta(I, X))$ 
    -- check if the m-state is not empty and add the arc
    if ( $I' \neq \emptyset$ ) then
      add arc  $I \xrightarrow{X} I'$  to the graph of  $\vartheta$ 
      -- check if the m-state is a new one and add it
      if ( $I' \notin R$ ) then
        add m-state  $I'$  to the set  $R'$ 
      end if
    end if
  end for
  while ( $R \neq R'$ )                                -- 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:

$$I_{\text{base}} = \{\langle q, \pi \rangle \in I \mid q \text{ is not an initial state}\}$$

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 :

$$I_{\text{closure}} = \{\langle q, \pi \rangle \in I \mid q \text{ is an initial state}\}$$

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:

$$I_{\text{kernel}} = \{q \in Q \mid \langle q, \pi \rangle \in I\}$$

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 4.31 (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 we have $\delta(q, X) = \delta(r, X)$. 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 4.32 (Pilot graph of the running example) The pilot graph \mathcal{P} of the machine net of Example 4.20 (see Fig. 4.8, p. 168) is shown in Fig. 4.13. 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 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.

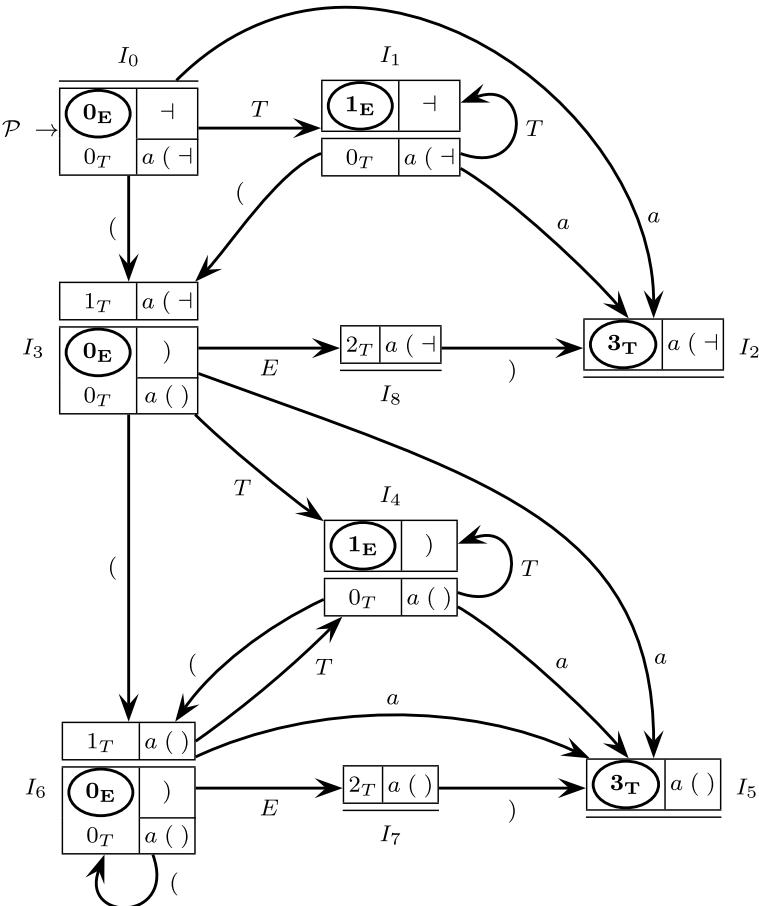


Fig. 4.13 ELR(1) pilot graph \mathcal{P} of the machine net in Fig. 4.8

Definition 4.33 (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*:

for all the candidates $\langle q, \pi \rangle \in I$ s.t. state q is final
 and for all the arcs $I \xrightarrow{a} I'$ that go out from I with terminal label a
 it must hold $a \notin \pi$ (4.4)

no *reduce-reduce conflict*:

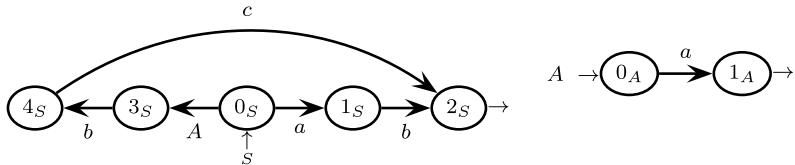
for all the candidates $\langle q, \pi \rangle, \langle r, \rho \rangle \in I$ s.t. states q, r are final
 it must hold $\pi \cap \rho = \emptyset$ (4.5)

EBNF grammar

$$S \rightarrow A b c \mid a b$$

$$A \rightarrow a$$

machine net



pilot graph

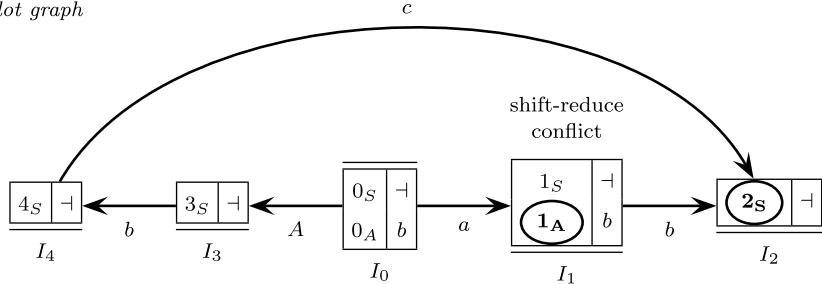


Fig. 4.14 Grammar with net and pilot with shift–reduce conflict (Example 4.34)

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. 4.13) does not have either shift–reduce or reduce–reduce conflicts. First, consider requirement (1). Clause (4.4) is met in every m-state that contains a final state; for instance in I_6 , the look-ahead set associated to reduction is disjoint from the labels of the outgoing arcs. Clause (4.5) 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 4.34 illustrates shift–reduce conflicts.

Example 4.34 (Shift–reduce conflicts) The grammar of Fig. 4.14 has a shift–reduce conflict in the m-state I_1 , where character b occurs in the look-ahead set of the final state 1_A and also as the label of the arc going out to m-state 2_S .

¹⁴The original acronym comes for “Left-to-right Rightmost” [25].

Next in Example 4.35, 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 4.35 (Reduce–reduce conflicts in a non-deterministic language) Consider the nondeterministic language:

$$L = \{a^n b^n \mid n \geq 1\} \cup \{a^{2n} b^n \mid n \geq 1\}$$

generated by the grammar

$$S \rightarrow A \mid B \quad A \rightarrow aAb \mid ab \quad B \rightarrow aaBb \mid aab$$

Since language L is nondeterministic, every grammar that generates it necessarily has conflicts. Figure 4.15 shows the pilot graph of the grammar.

The pilot graph has 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 $aaaabb$: after reading its prefix $aaaa$, the next token b is compatible with both reductions $ab \rightsquigarrow A$ and $aab \rightsquigarrow B$, which makes the analysis nondeterministic.

Finally in Example 4.36 we show the case of convergent transitions and convergence conflicts, which are a characteristic of EBNF grammars not possessed by purely BNF grammars.

Example 4.36 (Convergence conflicts) Consider the EBNF grammar and the corresponding machines in Fig. 4.16. To illustrate the notion of convergent transition with and without conflict, we refer to the pilot in Fig. 4.16. 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. 4.16. 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. 4.16 does not 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. 4.19 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.

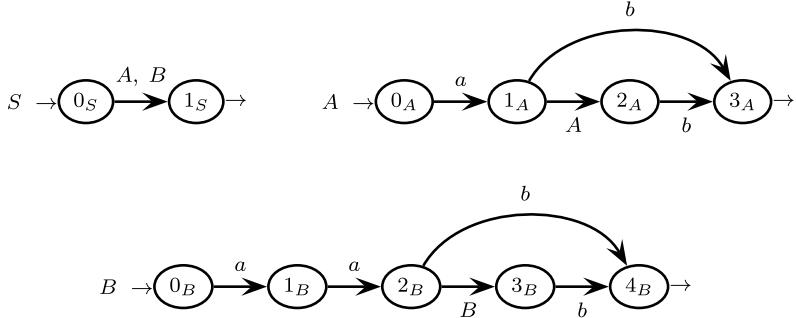
4.6.3.1 Parser Algorithm

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.

EBNF grammar

$$S \rightarrow A \mid B \quad A \rightarrow a (b \mid A b) \quad B \rightarrow a a (b \mid B b)$$

machine net



pilot graph

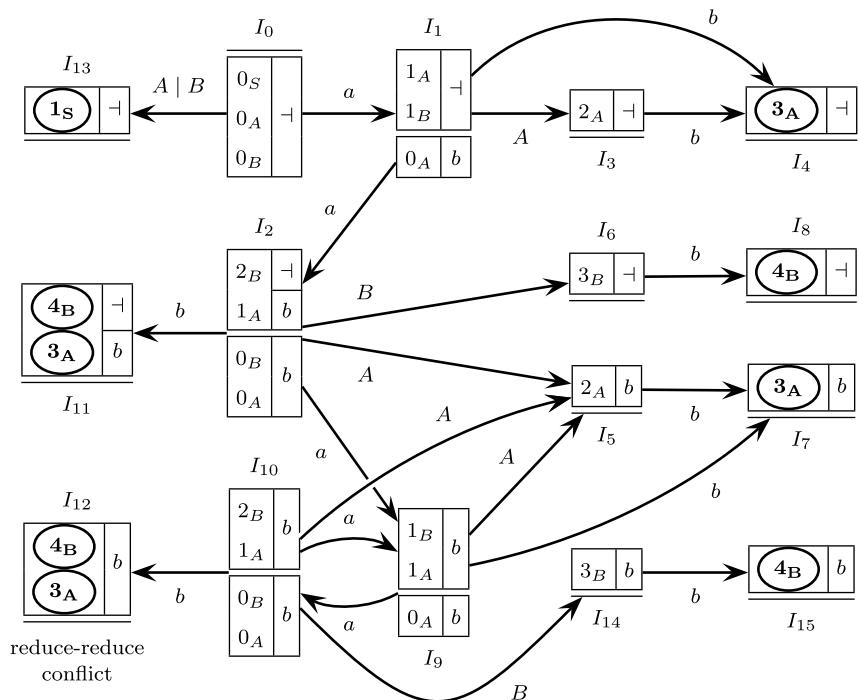


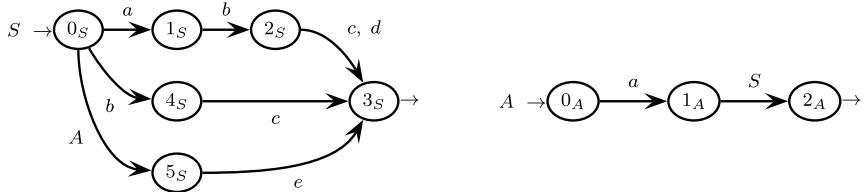
Fig. 4.15 Grammar with net and pilot with reduce–reduce conflict (Example 4.35)

EBNF grammar

$$S \rightarrow a b (c \mid d) \mid b c \mid A e$$

$$A \rightarrow a S$$

machine net



pilot graph

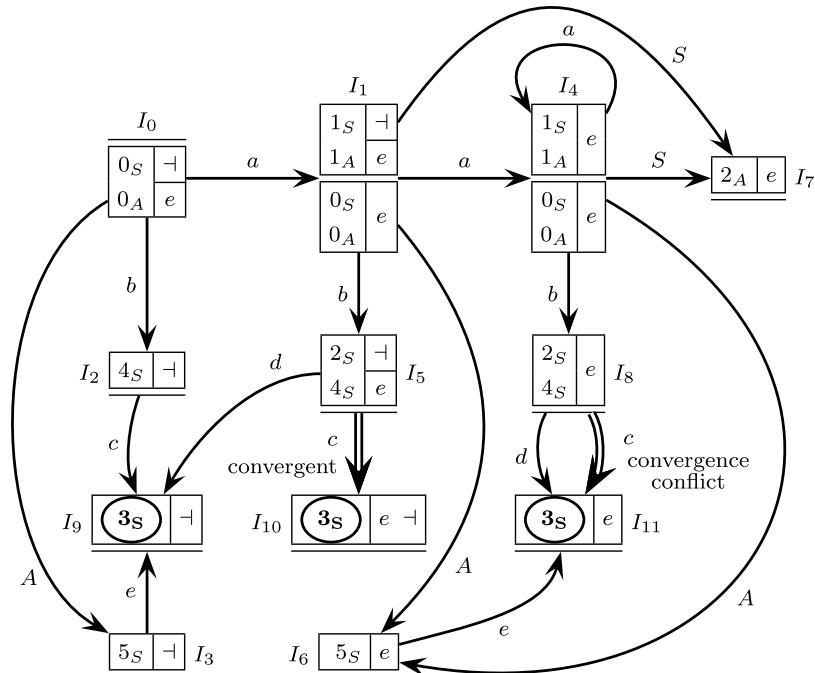


Fig. 4.16 Grammar with net and pilot (Example 4.36); the double-line arcs in the pilot are convergent; the pilot has a convergence conflict

At the cost of some repetition, we recall the three sorts of abstract machines involved: the net \mathcal{M} of DFA's M_S, M_A, \dots with state set $Q = \{q, \dots\}$ (states are drawn as circular or oval nodes); the pilot DFA \mathcal{P} with m-state set $R = \{I_0, I_1, \dots\}$ (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 top-most 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

$$\langle q_A, \pi, cid \rangle \quad \text{where} \quad \begin{cases} q_A \in Q & \text{(state)} \\ \pi \subseteq \Sigma \cup \{\dashv\} & \text{(look-ahead set)} \\ 1 \leq cid \leq |Q| \text{ or } cid = \perp & \text{(candidate identifier)} \end{cases}$$

where \perp represents the nil value for the *cid* pointer. For readability, a *cid* value will be prefixed by a \sharp marker, e.g., $\sharp 1$, $\sharp 2$, 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 4.37 (*ELR(1)* parser \mathcal{A} as a *DPDA*) Let the current stack be as follows:

$$J[0]a_1 J[1]a_2 \dots a_k J[k]^{\text{top}}$$

where a_i is a grammar symbol.

initialization The initial stack just contains the *sms*

$$J_0 = \{s \mid s = \langle q, \pi, \perp \rangle \text{ for every candidate } \langle q, \pi \rangle \in I_0\}$$

(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:

$$J' = \{ \langle q'_A, \rho, \sharp i \rangle \mid \langle q_A, \rho, \sharp j \rangle \text{ is at position } i \text{ in } J \text{ and } q_A \xrightarrow{a} q'_A \in \delta \} \quad (4.6)$$

$$\cup \{ \langle 0_A, \sigma, \perp \rangle \mid \langle 0_A, \sigma \rangle \in I'_{\text{closure}} \} \quad (4.7)$$

(thus $\mu(J') = I'$)

(note: by the last condition in (4.6), state q'_A is of the base of I')
reduction move (non-initial state) The current stack is as follows:

$$J[0]a_1J[1]a_2 \dots a_kJ[k]$$

and let the corresponding m-states be $I[i] = \mu(J[i])$ with $0 \leq i \leq k$; 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:

$$a_{h+1}a_{h+2} \dots a_k \rightsquigarrow A$$

2. pop the following stack symbols:

$$J[k]a_kJ[k-1]a_{k-1} \dots J[h+1]a_{h+1}$$

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 .

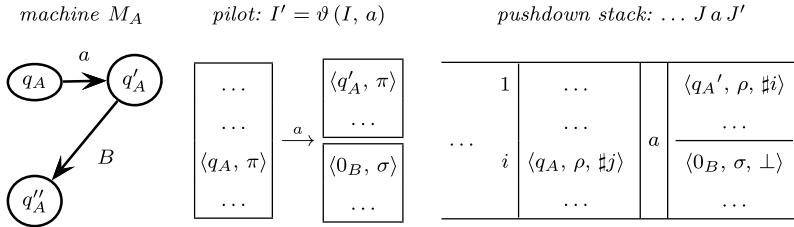


Fig. 4.17 From left to right: machine transition $q_A \xrightarrow{a} q'_A$, shift move $I \xrightarrow{a} I'$ and stack configuration. The *cid* value $\#i$ in the top-most stack m-state, points to the candidate at offset i in the underlying stack m-state

For shift moves, we note that the set computed at line (4.6), 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. (4.6) and (4.7)) schematized in Fig. 4.17.

Example 4.38 (Parsing trace) The step-by-step execution of the parser on input string $((a))$ produces the trace shown in Fig. 4.18. 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 $\#i$ 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., $(\circled{1}_E)$. To avoid clogging, the look-ahead sets are not shown but can be found in the pilot graph at p. 185 (they are only needed for convergent transitions, which do not occur here).

Figure 4.18 highlights the shift moves as dashed forward arrows that link two top-most stack candidates. For instance the first (from the figure top) terminal shift, on ‘(’, is $\langle 0_T, \perp \rangle \xrightarrow{\cdot} \langle 1_T, \#2 \rangle$. The first nonterminal shift is the third one, on E , i.e., $\langle 1_T, \#3 \rangle \xrightarrow{E} \langle 2_T, \#1 \rangle$, soon after the null reduction $\varepsilon \sim E$.

In the same figure, the reduction handles are evidenced by means of solid backward arrows that link the candidates involved. For instance, for reduction $(E) \sim T$ the stack configuration shows the chain of three pointers $\#1$, $\#1$ and $\#3$ —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 second shift move on ‘(’—see the effect on the line below) highlights the null reduction $\varepsilon \sim E$, which does not pop anything from the stack, and immediately triggers a shift on E .

stack bottom	string to be parsed (with end-marker) and stack contents					effect after
	1	2	3	4	5	
	(()	a)	+1
						initialization of the stack
	(()	a)	+1
	(()	a)	shift on (
	(()	a)	+1
	(()	a)	shift on (
	((E))	+1
	((E))	reduction $\varepsilon \rightsquigarrow E$ and shift on E
	((E))	+1
	((E))	shift on)
	((T	a)	+1
	((T	a)	reduction ($E \rightsquigarrow T$) and shift on T
	((T	a)	+1
	((T))	shift on a
	((T))	+1
	((E))	reduction $a \rightsquigarrow T$ and shift on T
	((E))	+1
	((E))	reduction $TT \rightsquigarrow E$ and shift on E
	((E))	+1
	((E))	shift on)
	((E))	+1
	((E))	reduction ($E \rightsquigarrow T$) and shift on T
	((E))	+1
	((E))	reduction $T \rightsquigarrow E$ and accept (do not shift on E)

Fig. 4.18 Parsing steps for string $((a))$; the grammar and the $ELR(1)$ pilot are in Figs. 4.8 and 4.13, p. 185; the name J_h of a sms maps onto the corresponding m-state $\mu(J_h) = I_h$

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:

$$\varepsilon \rightsquigarrow E \quad (E) \rightsquigarrow T \quad a \rightsquigarrow T \quad TT \rightsquigarrow E \quad (E) \rightsquigarrow T \quad T \rightsquigarrow E$$

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. 4.17 (p. 192), 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 $\langle q'_A, \rho, \#i \rangle$ is linked via $\#i$ to $\langle q_A, \rho, \#j \rangle$); 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 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 4.39 (Convergent arcs in the pilot) The net, the pilot graph, and the trace of a parse are shown in Fig. 4.19, 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 4.40).

Example 4.40 (Several candidates in the base of a pilot m-state) We refer to Fig. 4.34, on p. 217. The grammar

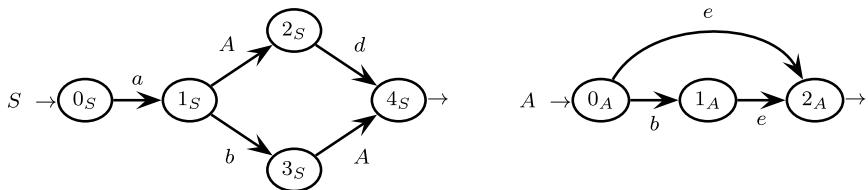
$$S \rightarrow a^*N \quad N \rightarrow aNb \mid \varepsilon$$

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:

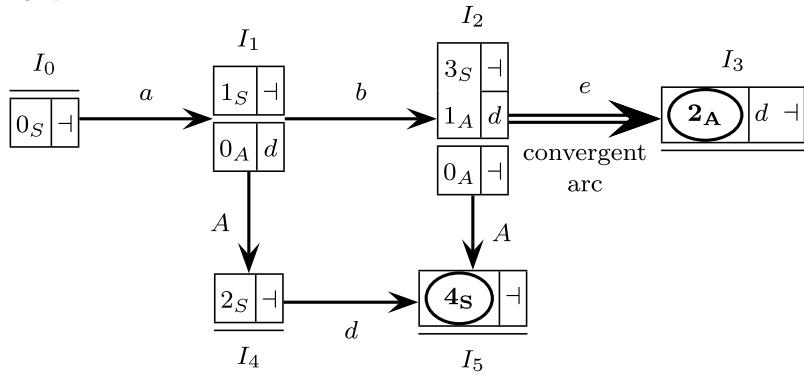
- At most one reduction candidate is present in any m-state, therefore reduce-reduce conflicts are ruled out.
- No shift-reduce conflict is present, because for every m-state that includes a reduction candidate with look-ahead set ρ , no outgoing arc is labeled with a token that is in ρ . We check this: 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 $aaab$ with the pilot in Fig. 4.34, would show that several parsing attempts proceed simultaneously, which correspond to paths in the machines M_S and M_N of the net.

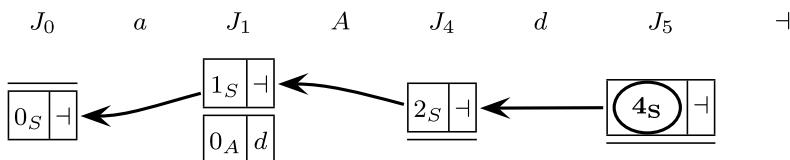
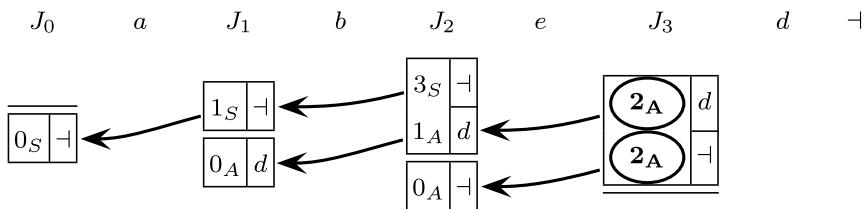
machine net



pilot graph



parse traces



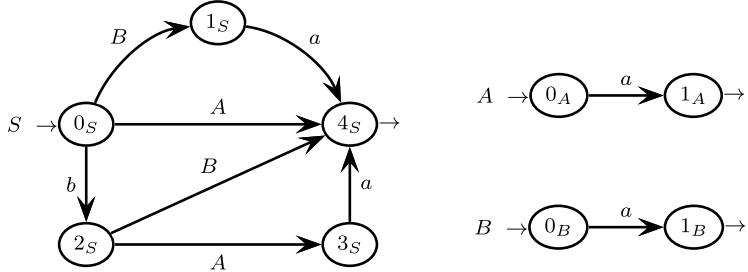
reductions $b e \rightsquigarrow A$ (above) and $a A d \rightsquigarrow S$ (below)

Fig. 4.19 ELR(1) net and pilot with convergent arcs (double line), and parsing trace of string $abed \dashv$

EBNF grammar

$$S \rightarrow B a \mid A \mid b (B \mid A a) \quad A \rightarrow a \quad B \rightarrow a$$

machine net



pilot graph

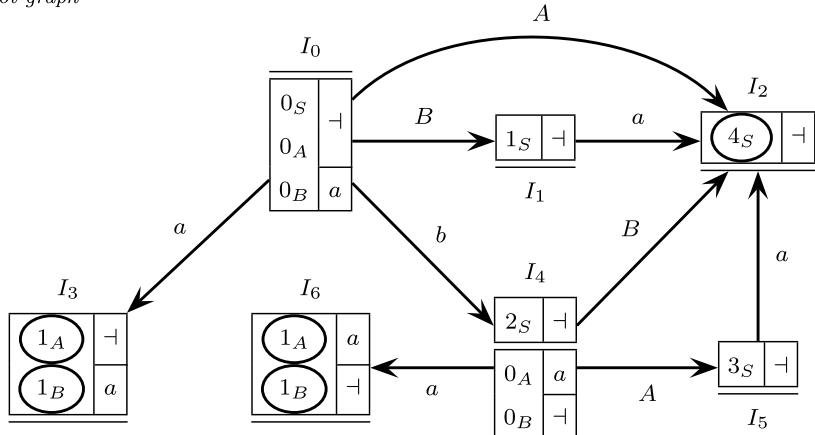


Fig. 4.20 Grammar with net and pilot with non-conflicting reductions

Next we illustrate the case of non-conflicting reductions in the same m-state.

Example 4.41 (Multiple reductions in an ELR(1) m-state) Figure 4.20 shows the grammar of a finite language, along with its machine network representation and the corresponding pilot automaton.

In the pilot of Fig. 4.20, 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 the derivation $S \Rightarrow A \Rightarrow a$; instead, if a second a is scanned after the first one, then the correct reduction is $a \rightsquigarrow B$ because the derivation is $S \Rightarrow Ba \Rightarrow aa$.

Similarly but with reference to m-state I_6 , if the input symbol is \dashv after reading string ba , then a reduction $a \rightsquigarrow B$ must be performed and the derivation is $S \Rightarrow bB \Rightarrow ba$; otherwise, if after prefix ba the input is a , then the correct reduction is $a \rightsquigarrow A$ and the derivation is $S \Rightarrow bAa \Rightarrow baa$.

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 in practical parser design should jump to p. 201, 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 $A \rightarrow \alpha \mid \beta \mid \dots$. 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 (4.4) and (4.5) of Definition 4.33 on p. 185.

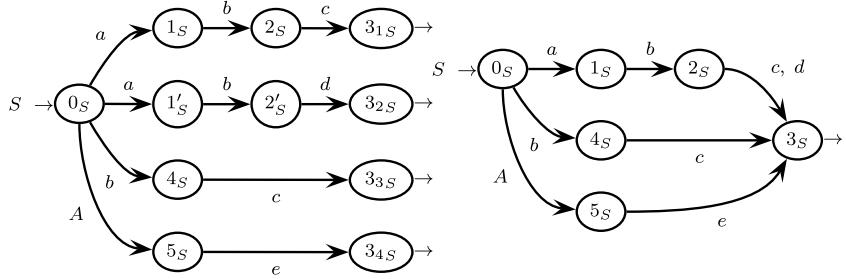
Our machine M_A may differ from N_A in two ways: determinism and minimality. First, machine M_A is assumed to be deterministic, but this assumption is made out of convenience, not of necessity. For instance, consider a nonterminal C with two alternatives: $C \rightarrow \text{if } E \text{ then } I \mid \text{if } E \text{ then } I \text{ else } I$. Determinization has the effect of normalizing the alternatives by left factoring the longest common prefix; this yields the equivalent *EBNF* grammar $C \rightarrow \text{if } E \text{ then } I(\epsilon \mid \text{else } I)$. 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.

¹⁵We mainly refer to the original paper by Knuth [25] and to its dissemination in many classical textbooks such as [1, 17]. There have been numerous attempts to extend the theory to *EBNF* grammars; we refer to the recent critical survey in [22].

grammars

<i>BNF</i>		<i>EBNF</i>
$S \rightarrow a b c \mid a b d \mid b c \mid A e$		$S \rightarrow a b (c \mid d) \mid b c \mid A e$
$A \rightarrow a S$		$A \rightarrow a S$

machine nets



net common part

$$A \rightarrow 0_A \xrightarrow{a} 1_A \xrightarrow{S} 2_A \rightarrow$$

Fig. 4.21 BNF and EBNF grammars and their machine nets N_S (left), M_S (right), and $N_A = M_A$ (bottom)

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 re-entered. Clearly in the graph of machine N_A , no arc may reenter the initial state, 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 (4.4) and (4.5), Definition 4.33 imposes that any convergent arc be free from conflicts. Since this point is quite subtle, we illustrate it by the next example.

Example 4.42 (State reduction and convergent transitions) Consider the equivalent BNF and EBNF grammars and the corresponding machines in Fig. 4.21.

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

$$S \Rightarrow Ae \Rightarrow aSe \Rightarrow abce \quad \text{and} \quad S \Rightarrow Ae \Rightarrow aSe \Rightarrow aabce$$

On the other hand, for the EBNF grammar the pilot $\mathcal{P}_{\{M_S, M_A\}}$, already shown in Fig. 4.16 on p. 189, has two convergent arcs (double line), one with a conflict and

EBNF grammar

$$S \rightarrow E (s E)^* \quad E \rightarrow b^+ F \mid F e \quad F \rightarrow b F e \mid \epsilon$$

machine net

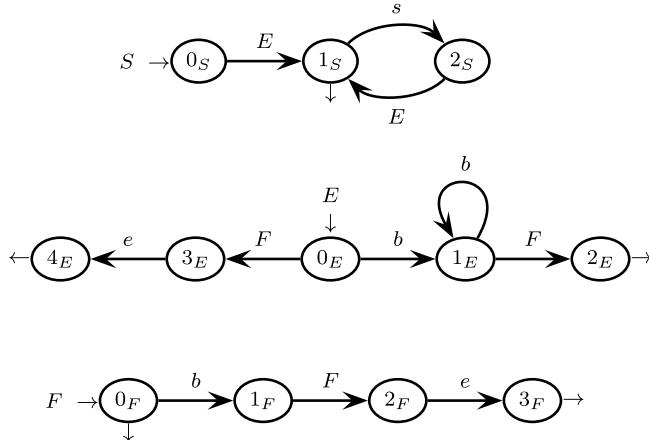


Fig. 4.22 An EBNF grammar that is *ELR*(1) (Example 4.43)

the other without. Arc $I_8 \xrightarrow{c} I_{11}$ violates the *ELR*(1) condition because the lookahead 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.

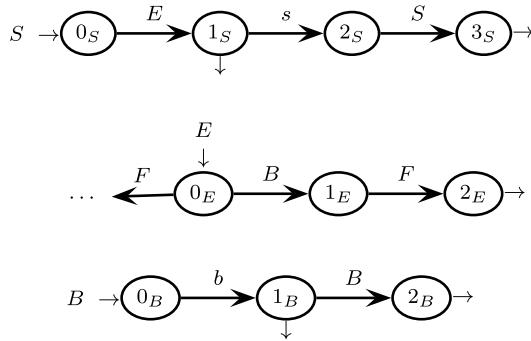
From these findings it is tempting to assert that, given a *BNF* grammar and an equivalent net, either both ones are conflict-free or both ones have conflicts (although not always of the same type). But the assertion needs to be made more precise, since starting from an *ELR*(1) net, several equivalent *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. 171).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 4.43 (Grammar transformation may not preserve *ELR*(1)) A phrase S has the structure $E(sE)^*$, 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. 4.22, which are *ELR*(1).

¹⁶A rigorous presentation of these and other theoretical properties is in [20].

machine net (partial)



pilot graph (partial)

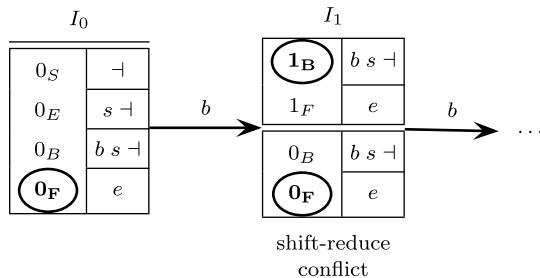


Fig. 4.23 An equivalent network that has conflicts (Example 4.43)

On the contrary, there is a conflict in the equivalent BNF grammar

$$S \rightarrow EsS \mid E \quad E \rightarrow BF \mid Fe \quad F \rightarrow bFe \mid \varepsilon \quad B \rightarrow bB \mid b$$

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. 4.23.

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 bbe , the parser, after shifting the first character b , should perform a reduction $b \sim B$ 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:

$$\begin{array}{lll} 0_S \rightarrow 0_E 1_S & 0_E \rightarrow b1_E \mid 0_F 3_E & 0_F \rightarrow \varepsilon \mid b1_F \\ 1_S \rightarrow \varepsilon \mid s2_S & 1_E \rightarrow b1_E \mid 0_F 2_E & 1_F \rightarrow 0_F 2_F \\ 2_S \rightarrow 0_E 1_S & 2_E \rightarrow \varepsilon & 2_F \rightarrow e3_F \\ 3_E \rightarrow e4_E & 3_F \rightarrow \varepsilon & \\ 4_E \rightarrow \varepsilon & & \end{array}$$

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 in the rules, some features of the $ELR(1)$ parsing algorithm (p. 190) become superfluous. We briefly discuss them to highlight the differences between extended and basic shift-reduce parsers. Since 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 $A \rightarrow \alpha \mid \beta \mid \dots$ are recognized in distinct final states $q_{\alpha,A}, q_{\beta,A}, \dots$ 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 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.

4.6.5 Parser Implementation Using a Vector Stack

Before finishing with bottom-up parsing, we present a parser implementation alternative to Algorithm 4.37, p. 190, 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 non-deterministic tabular parsers (Sect. 4.10); 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*, *vsm*s and grammar symbols. A *vsm*, 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 ele-

ments 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 4.44 (*ELR(1)* parser that uses a vector stack) The current (vector-)stack is denoted by

$$J[0]a_1 J[1]a_2 \dots a_k J[k]$$

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 *vsm*s J' (more precisely do $k++$ and $J[k] := J'$) that contains the following candidates:

<pre> 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 </pre>	<pre> if cand. $c = \langle 0_A, \pi \rangle \in I'_{\text{closure}}$ then J' contains cand. $\langle 0_A, \pi, k \rangle$ end if </pre>
---	--

$$(\text{thus } \mu(J') = I')$$

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:

$$J[0]a_1 J[1]a_2 \dots a_k J[k]$$

and let the corresponding m-states be $I[0]I[1]\dots I[k]$. 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:

$$J[k]a_k J[k-1]a_{k-1} \dots J[h+1]$$

2. build a portion of the syntax forest by applying the following reduction:

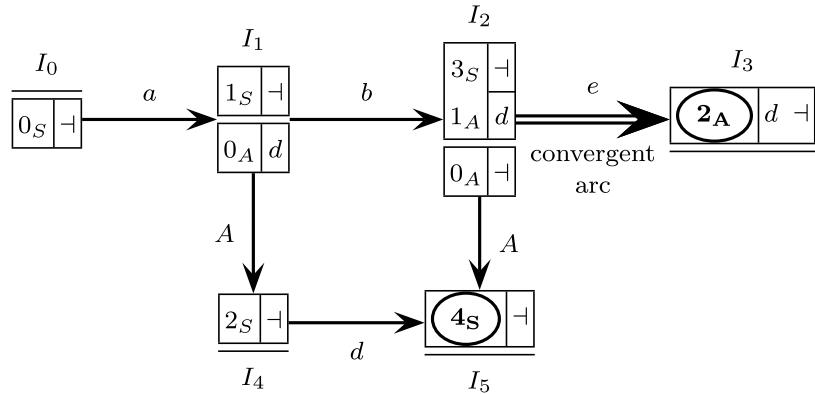
$$a_{h+1} \dots a_k \rightsquigarrow A$$

3. execute the nonterminal shift move $\vartheta(J[h], A)$

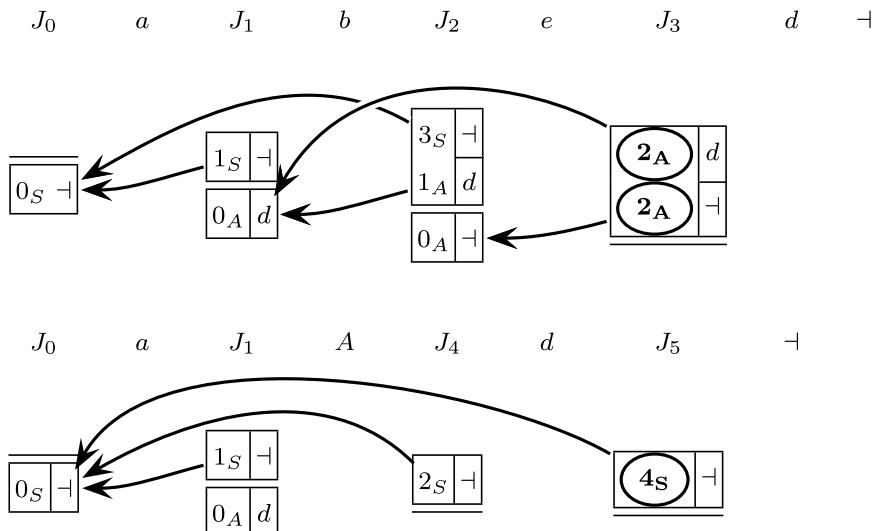
stack bottom	string to be parsed (with end-marker) and stack contents (with indices)						effect after
0	1	2	3	4	5	6	
	(()	a)	-	initialization of the stack
	(()	a)	-	shift on (
	(()	a)	-	shift on (
	(()	a)	-	reduction $\varepsilon \rightsquigarrow E$ and shift on E
	((E)	a)	shift on)
	(T		a)	-	reduction ($E \rightsquigarrow T$) and shift on T
	(T		a)	-	shift on a
	(T	T)	-	-	reduction $a \rightsquigarrow T$ and shift on T
	(E)	-	-	reduction $TT \rightsquigarrow E$ and shift on E
	(E)	-	-	shift on)
	T		-	-	-	-	reduction ($E \rightsquigarrow T$) and shift on T
	E		-	-	-	-	reduction $T \rightsquigarrow E$ and accept (do not shift on E)

Fig. 4.24 Tabulation of parsing steps for string $((a))$; the parser uses a vector stack and is generated by the grammar in Fig. 4.8 with the $ELR(1)$ pilot in Fig. 4.13

pilot graph



parse traces



reductions $b e \rightsquigarrow A$ (above) and $a A d \rightsquigarrow S$ (below)

Fig. 4.25 Parsing steps of the parser using a vector stack for string $abed \dashv$ recognized by the net in Fig. 4.19, with the pilot here reproduced for convenience

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 4.45 (Parsing trace) Figure 4.24, to be compared with Fig. 4.18 on p. 193, shows the same step-by-step execution of the parser as in Example 4.38, on input string $((a)$; the graphical conventions are the same.

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. 4.18, as index *elemid* goes directly to the origin of the reduction handle. The forward shift arrows are the same as those in Fig. 4.18 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 4.38.

To complete the exemplification, we apply the vector-stack parser to a previous net featuring a convergent arc (Fig. 4.19, p. 195): the parsing traces are shown in Fig. 4.25, 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)* condition, but one may find a grammar that needs some adjustment. Here we consider the case of *BNF* grammars, because it is the only one discussed in the scientific literature. For such grammars, the *ELR* property is equivalent to the classical *LR* one, therefore here we consider the latter. Remember, however, that an *EBNF* grammar can always be transformed into an equivalent *BNF* one: for instance into the right-linearized form; see Sect. 4.5.1.1 on p. 171.

In Sect. 4.6 on p. 176, we have intuitively hinted what a look-ahead of length two (or more) means: 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: $S \rightarrow Acd \mid Bce$; and that both nonterminals A and B are nullable, i.e., $A \rightarrow \varepsilon$ and $B \rightarrow \varepsilon$. 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 ones have look-ahead c if we limit ourselves to look at only one character after them, that is, if we set $k = 1$. But if we look at two consecutive characters that may follow nonterminals A and B , that is, if we set $k = 2$, then we, respectively, see strings cd and ce , which are

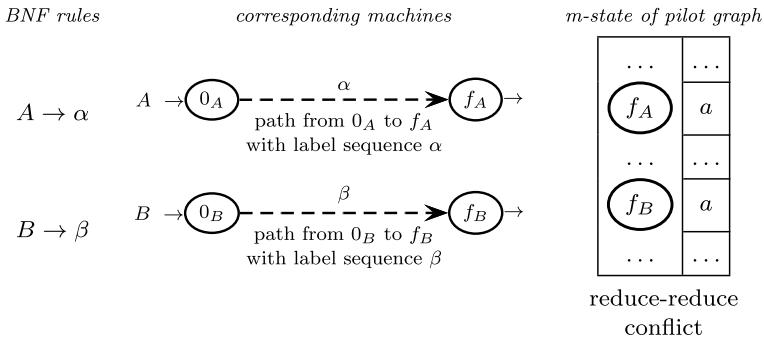


Fig. 4.26 BNF grammar not of type $LR(1)$ with a reduce–reduce conflict

different and so make the conflict vanish. 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. Thus a BNF grammar is of type $LR(k)$, for some value $k \geq 1$, if it has a deterministic bottom-up syntax analyzer that uses a look-ahead of length k .

The interesting case is when we have a grammar that is of type LR , but with a look-ahead of length k equal to two or more, i.e., $LR(k)$ with $k > 1$. In this case we may wish to lower the look-ahead length parameter k , possibly down to $k = 1$. In principle this target can always be achieved; see Sect. 4.8 on p. 242. Here we show a few useful BNF grammar transformations from $LR(2)$ down to $LR(1)$, which do not pretend of being exhaustive.

From Sect. 4.6.4.1 we know that confining to BNF grammars, means that the net machines do not have 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. 4.27. First we introduce two transformations useful for lowering the value of the look-ahead length parameter k in the BNF case, then we hint a possible transformation for non- $LR(k)$ grammars.¹⁷

Grammar with a Reduce–Reduce Conflict Suppose a BNF grammar is of type $LR(2)$, but not $LR(1)$. The situation is as in Fig. 4.26, where we see two rules $A \rightarrow \alpha$ and $B \rightarrow \beta$, 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, we can conveniently represent the conflict by means of marked rules with look-ahead instead of state candidates, as was suggested in

¹⁷For a broader discussion of all such grammar transformations, we refer to [11].

Sect. 4.5 on p. 169. In this case we have

Reduction	Reduction	Representation
$A \rightarrow \alpha \bullet, \{a\}$	$B \rightarrow \beta \bullet, \{a\}$	<i>Marked rule</i>
$\langle f_A, a \rangle$	$\langle f_B, a \rangle$	<i>State candidate</i>

Both marked rules are reductions as the bullet \bullet is at their end, and have the same look-ahead set $\{a\}$ with a common character 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 conflict causing character a . 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 the two new nonterminals $\langle Aa \rangle$, $\langle Ba \rangle$, and their rules, as

$$\langle Aa \rangle \rightarrow \alpha a \quad \langle Ba \rangle \rightarrow \beta a$$

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

$$\langle Aa \rangle \xrightarrow{+} \gamma a$$

exists if, and only if, the original grammar has the following derivation:

$$A \xrightarrow{+} \gamma$$

and character a can follow nonterminal A . The next Example 4.46, with both marked rules and pilot graphs, instances an early scanning case.

Example 4.46 (Early scanning) Consider the BNF grammar G_1

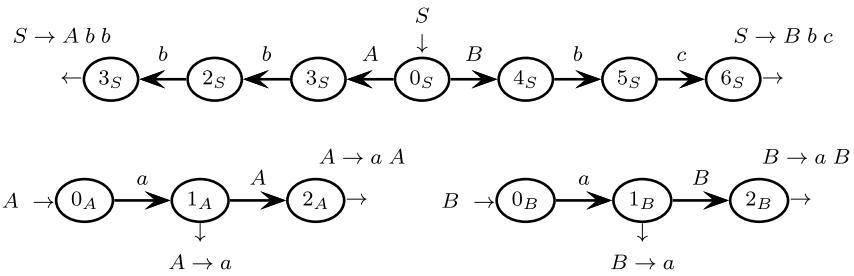
$$\begin{array}{lll} S \rightarrow Abb & A \rightarrow aA & A \rightarrow a \\ S \rightarrow Bbc & B \rightarrow aB & B \rightarrow a \end{array}$$

Grammar G_1 is of type $LR(2)$, but not $LR(1)$ because it clearly has a reduce-reduce conflict between marked rules $A \rightarrow a \bullet, \{b\}$ and $B \rightarrow a \bullet, \{b\}$: the look-ahead is b for both ones. The situation is as in Fig. 4.27, 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 $\{bb\}$ and $\{bc\}$, which are disjoint strings. Thus by applying early scanning, we obtain the $LR(1)$ grammar

$$\begin{array}{lll} S \rightarrow \langle Ab \rangle b & \langle Ab \rangle \rightarrow a \langle Ab \rangle & \langle Ab \rangle \rightarrow ab \\ S \rightarrow \langle Bb \rangle c & \langle Bb \rangle \rightarrow a \langle Bb \rangle & \langle Bb \rangle \rightarrow ab \end{array}$$

machine net



partial pilot graph

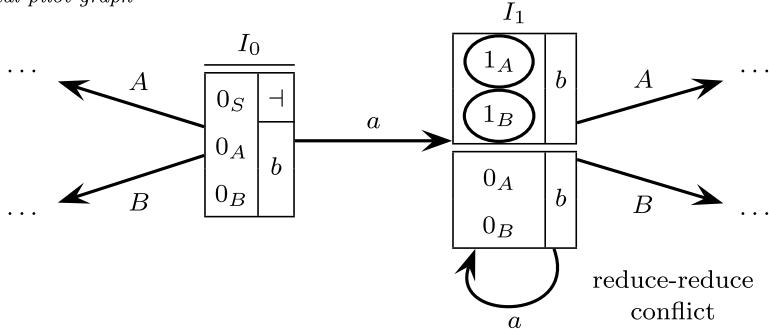


Fig. 4.27 BNF grammar G_1 before early scanning

The two reduction marked rules are $\langle Ab \rangle \rightarrow ab\bullet, \{b\}$ and $\langle Bb \rangle \rightarrow ab\bullet, \{c\}$, which do not conflict because their look-ahead sets are disjoint. Now the situation becomes as in Fig. 4.28, where there is not any conflict in the m-states I_1 or I_2 , which replace the original conflicting one.

Therefore after the early scanning transformation, the reduce-reduce conflict has vanished and the grammar has become of type $LR(1)$.

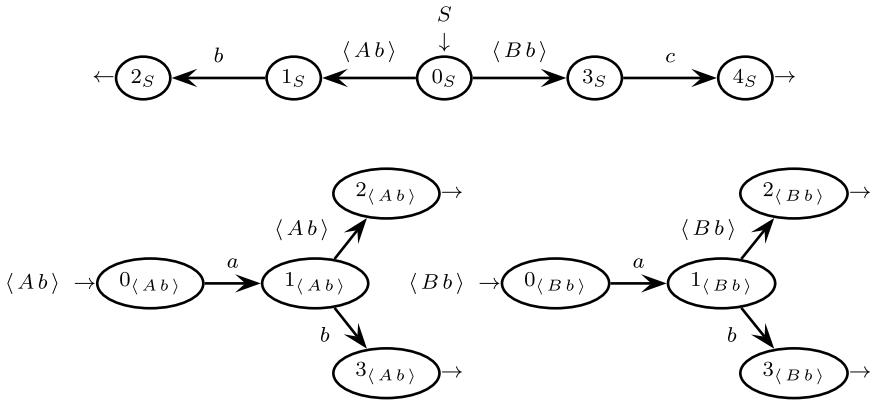
Grammar with a Shift–Reduce Conflict Another common source of conflict occurs when an m-state of the pilot contains two marked rules, which are a shift and a reduction candidate, as

$$A \rightarrow \alpha \bullet a\beta, \pi \quad B \rightarrow \gamma \bullet, \{a\}$$

These candidates cause a shift–reduce conflict, due to character a : a shift on character a is enabled by candidate $A \rightarrow \alpha \bullet a\beta$; and character a belongs to the look-ahead set $\{a\}$ of the reduction candidate $B \rightarrow \gamma \bullet$ (the look-ahead set π of the shift candidate is not relevant for causing the conflict).

How to fix the conflict: create a new nonterminal $\langle Ba \rangle$ to do the same service as nonterminal B followed by terminal a ; and replace rule $B \rightarrow \gamma$ with rule $\langle Ba \rangle \rightarrow$

machine net



partial pilot graph

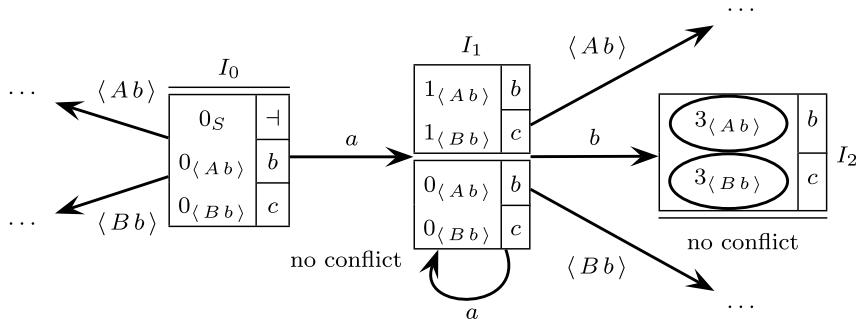


Fig. 4.28 BNF grammar G_1 after early scanning

γa . For consistency, adjust all the rules having nonterminal B in their right parts so as to preserve grammar equivalence. For instance, if there is a rule like $X \rightarrow Bac$, replace it with rule $X \rightarrow \langle Ba \rangle c$. Since we assume the grammar is of type $LR(2)$, the look-ahead set of rule $\langle Ba \rangle \rightarrow \gamma a$ may not include character a ,¹⁸ and so the conflict vanishes.

The transformation is more involved if character a , which causes the conflict between candidates $A \rightarrow \alpha \bullet a\beta, \pi$ and $B \rightarrow \gamma \bullet, \{a\}$, is produced as initial character by a derivation from nonterminal C , which immediately follows nonterminal B in a sentential form, like

$$S \stackrel{+}{\Rightarrow} \dots BC \dots \stackrel{+}{\Rightarrow} \dots Ba \dots$$

¹⁸More generally, in this case the $LR(2)$ assumption implies that the look-ahead set (with $k = 1$) of rule $\langle Ba \rangle \rightarrow \gamma a$ and the set $Ini(L(\beta) \cdot \pi)$, 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:

$$\langle a /_L C \rangle \stackrel{+}{\Rightarrow} \gamma \quad \text{if and only if} \quad C \stackrel{+}{\Rightarrow} a\gamma \quad \text{in the original grammar}$$

Then the affected grammar rules are adjusted so as to preserve equivalence. Now the grammar thus 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 4.47 shows the transformation. For simplicity, we represent it only with marked rules (with state candidates it goes the same).

Example 4.47 (Left quotient preparation for early scanning) Consider the BNF grammar G_2

$$\begin{array}{lll} S \rightarrow Ad & A \rightarrow ab \\ S \rightarrow BC & B \rightarrow a & C \rightarrow bC \\ & & C \rightarrow c \end{array}$$

Grammar G_2 is of type $LR(2)$, but not $LR(1)$ as evidenced by the following shift-reduce conflict, due to character b :

$$A \rightarrow a \bullet b, \{d\} \quad B \rightarrow a \bullet, \{b, c\}$$

After a left quotient transformation applied to nonterminal C , we obtain the grammar below (nonterminal C is left-quotiented by b and c):

$$\begin{array}{llll} S \rightarrow Ad & A \rightarrow ab \\ S \rightarrow Bb \langle b /_L C \rangle & B \rightarrow a & \langle b /_L C \rangle \rightarrow b \langle b /_L C \rangle & \langle b /_L C \rangle \rightarrow c \\ S \rightarrow Bc \langle c /_L C \rangle & & \langle c /_L C \rangle \rightarrow \varepsilon & \end{array}$$

Now the shift-reduce conflict can be removed by applying the early scanning transformation, as done in Example 4.46, by means of two more nonterminals $\langle Bb \rangle$ and $\langle Bc \rangle$, as follows:

$$\begin{array}{llll} S \rightarrow Ad & A \rightarrow ab \\ S \rightarrow \langle Bb \rangle \langle b /_L C \rangle & \langle Bb \rangle \rightarrow ab & \langle b /_L C \rangle \rightarrow b \langle b /_L C \rangle & \langle b /_L C \rangle \rightarrow c \\ S \rightarrow \langle Bc \rangle \langle c /_L C \rangle & \langle Bc \rangle \rightarrow ac & \langle c /_L C \rangle \rightarrow \varepsilon & \end{array}$$

The shift-reduce conflict has vanished, and there are not any reduce-reduce conflicts. In fact the only rules that reduce together are $A \rightarrow ab \bullet, \{d\}$ and $\langle Bb \rangle \rightarrow ab \bullet, \{b, c\}$, but their look-ahead sets (with $k = 1$) are disjoint. Therefore the grammar has eventually become of type $LR(1)$.

Finally we observe that a grammar transformation based on left quotient and early scanning, can also be used when the grammar meets the LR condition for a value of the look-ahead length parameter k larger than two.

Transformation of Non-LR(k) Grammars It is known that for every $LR(k)$ grammar with $k > 1$, there is an equivalent $LR(1)$ grammar; see Property 4.73 on p. 245. We have seen some grammar transformations that allow to lower the look-ahead length parameter k . Anyway, if the language is deterministic yet the given grammar is not of type $LR(k)$ for any $k \geq 1$, then we do not have any systematic transformations to turn it into a $LR(k)$ one for some $k \geq 1$. However, we may try to study the languages generated by the critical nonterminals, to identify the conflict sources, and finally to adjust such sub-grammars and make them of type $LR(1)$.

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. 63. 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.

Example 4.48 (List of balanced strings) The machine net in Fig. 4.29, equivalent to the $EBNF$ grammar:

$$S \rightarrow P(','P)^* \quad P \rightarrow aPa \mid c$$

generates the language $\{a^n ca^n \mid n \geq 0\} (',\{a^n ca^n \mid n \geq 0\})^*$.

¹⁹The 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 [26, 33]. A sound method to extend such parsers to $EBNF$ grammars was popularized by [41] recursive-descent compiler, systematized in the book [28], and included in widely known compiler textbooks, e.g., [1], where top-down deterministic parsing is presented independently of shift-reduce methods. Taking a novel approach from [9], this section shows that top-down parsing for $EBNF$ grammars is a corollary of the $ELR(1)$ parser construction that we have just presented. 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 [7].

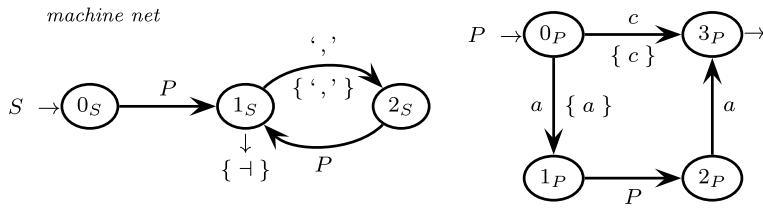


Fig. 4.29 Machine net for lists of balanced strings (Example 4.48). On the arcs and final darts exiting furcating nodes, the tokens within braces (so-called *guide sets*, see p. 227 or p. 233) tell the parser which branch to follow

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 $x = aca \dashv$, current parser state 0_S and current token a , the only possibility is arc $0_S \xrightarrow{a} 1_S$, 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 $S \Rightarrow P \dots \Rightarrow aP \dots \Rightarrow ac \dots$. 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 $S \Rightarrow P \Rightarrow aPa \Rightarrow aca$. On the other hand, if the input were $aca, c \dashv$, after processing aca the parser is in the state 1_S , the next token is a comma, and arc $1_S \xrightarrow{P} 0_S$ is taken.

Syntactic Procedures Our wording in terms 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 procedures, which correspond to the nonterminal symbols. The syntactic procedures for the net in Fig. 4.29 are listed in Fig. 4.30.

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 pro-

<pre> procedure <i>S</i> LOOP: call <i>P</i> 1: if <i>cc</i> = ‘-’ then accept stop 2: else if <i>cc</i> = ‘,’ then <i>cc</i> := <i>next</i> goto LOOP 3: else error end if end procedure </pre>	<pre> procedure <i>P</i> 1: if <i>cc</i> = ‘a’ then <i>cc</i> := <i>next</i> call <i>P</i> 1a: if <i>cc</i> = ‘a’ then <i>cc</i> := <i>next</i> 1b: else error end if 2: else if <i>cc</i> = ‘c’ then <i>cc</i> := <i>next</i> 3: else error end if end procedure </pre>
---	---

Fig. 4.30 Syntactic procedures of a recursive-descent parser (Example 4.48)

gram 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 the branch to be taken? Similarly, if the state is final and an arc originates 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 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 4.49 (Choices at furcating states) The net in Fig. 4.31 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 *aaab*.... To make the right decision, one has to look a long distance ahead of the current token, to see if the string is, say, *aaab* - or *aaabb* -. We shall see that this confusing situation is easily diagnosed by examining the macro-states of the *ELR(1)* pilot.

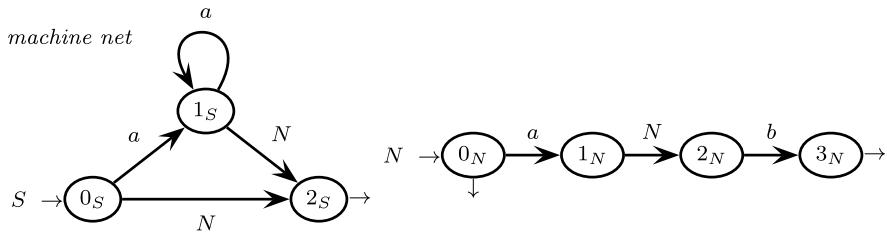


Fig. 4.31 Net of Example 4.49. The bifurcation in the node 0_S causes a nondeterministic choice (see also the *ELR(1)* pilot graph on p. 217)

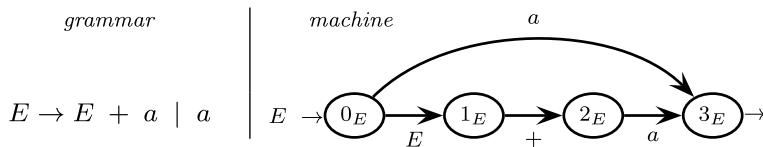


Fig. 4.32 A left-recursive grammar and machine that makes top-down deterministic choices impossible. The case of left-recursive grammars is discussed at length on p. 236

The next example of nondeterministic bifurcation point is caused by a left-recursive rule in the grammar depicted in Fig. 4.32. In the state 0_E , both arcs agree with 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.

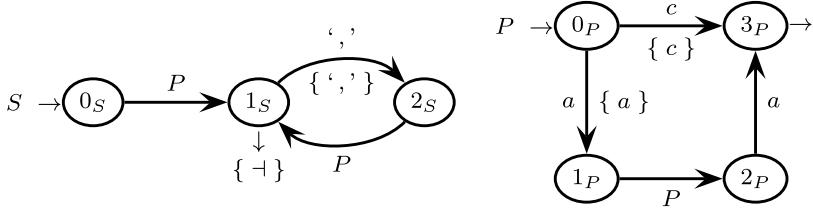
4.7.1 *ELL(1)* Condition

Before we state the definition of top-down deterministically parsable grammar or network, 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. 184) *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 4.50 (*ELL(1)* condition)²⁰ A machine net \mathcal{M} meets the *ELL(1)* condition if the following three clauses are satisfied:

²⁰The historical acronym “*ELL(1)*” has been introduced over and over in the past by several authors with slight differences. We hope that reusing again the acronym will not be considered an abuse.

machine net



ELR(1) pilot

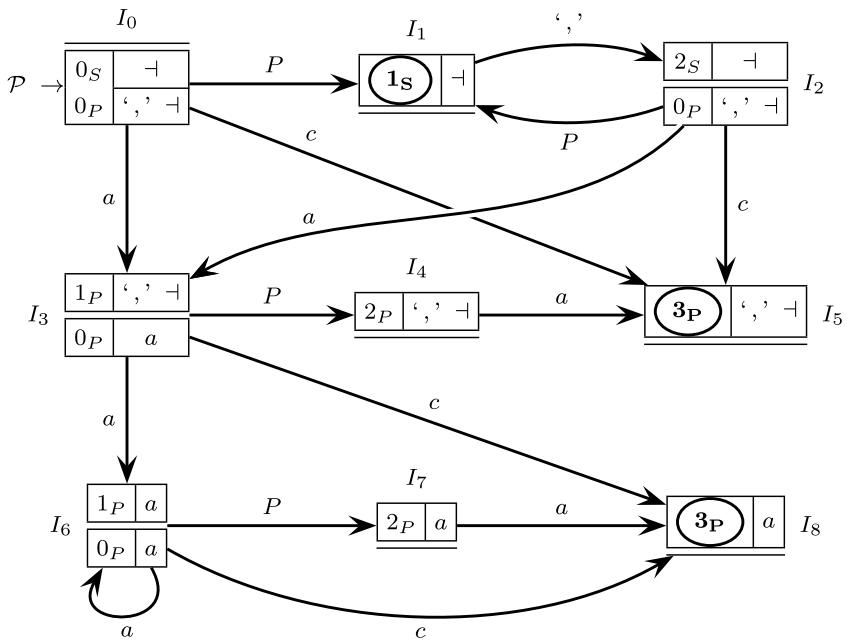


Fig. 4.33 Machine net (reproduced from Fig. 4.29 on p. 212) and its *ELR(1)* pilot. The *ELL(1)* condition is satisfied

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. 184)
3. the net has the single-transition property (*STP*)

Example 4.51 (Clauses of the *ELL(1)* condition) We check that the machine net that introduced top-down parsing on p. 212, and reproduced in Fig. 4.33, 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. 4.33 does not have any 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. 4.30.

The previous examples that raised difficulties for deterministic decisions at furtionation points are now revisited to illustrate violations of the *ELL(1)* condition.

Clause 1: the grammar in Fig. 4.32, p. 214, 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, p. 37. 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 the conversion method presented on p. 63. 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 4.50 states that the net must enjoy *STP*; we discuss some cases of violation.

Example 4.52 (Violations of *STP* in *ELR(1)* nets) Three cases are examined, none of them left recursive. First, the net in Fig. 4.34 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 runtime configuration consists of only one machine state.

Second, an older example, the net in Fig. 4.19 on p. 195 illustrates the case of multiple candidates in the base of an m-state, I_3 , that has a convergent arc.

Third, we observe the net of grammar $S \rightarrow b(a \mid Sc) \mid a$ in Fig. 4.35. 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.

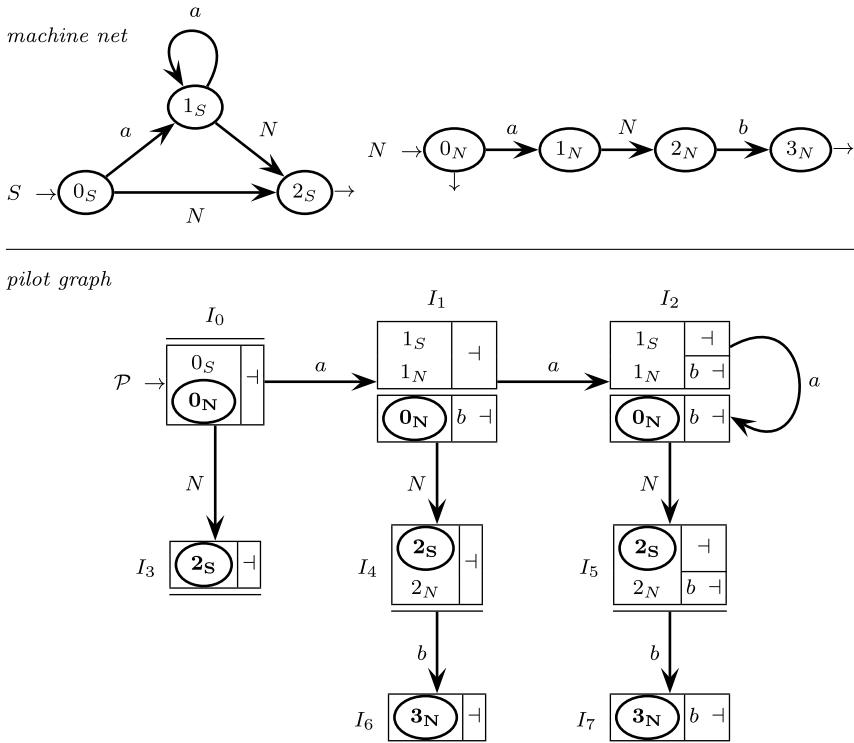


Fig. 4.34 $ELR(1)$ net of Example 4.52 with multiple candidates in the m-state base of I_1 , I_2 , I_4 and I_5

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.

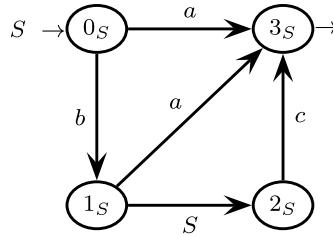
Readers aiming for immediate practical indications for building such parsers, should skip Sect. 4.7.2 and jump ahead to Sect. 4.7.3, p. 231. Conversely, readers interested in understanding the relationship between bottom-up and top-down parsers, should continue reading the next section.

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

²¹The reader is referred to [9] for a more formal presentation of the various steps, including the proofs of their correctness.

machine net



pilot graph

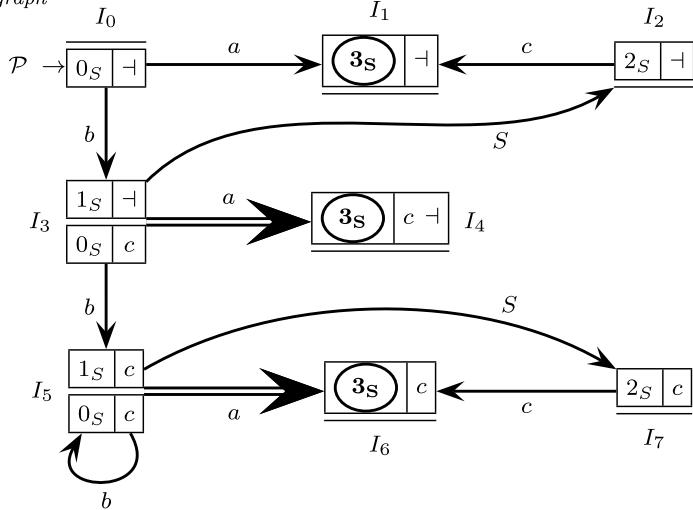


Fig. 4.35 $ELR(1)$ net of Example 4.52 with just one candidate in each m-state base, yet with convergent arcs (evidenced by double arrows)

To start, we focus on and show two important effects. First, condition STP permits to greatly reduce the number of m-states, to the number of net states. Second, convergent 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. 183) 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 4.53 (Operation $\text{Merge}(I_1, I_2)$)

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:

$$\langle p, \pi \rangle \in I_{1,2} \iff \langle p, \pi_1 \rangle \in I_1 \quad \text{and} \quad \langle p, \pi_2 \rangle \in I_2 \quad \text{and} \quad \pi = \pi_1 \cup \pi_2$$

2. m-state $I_{1,2}$ becomes the target for all the labeled arcs that entered I_1 or I_2 :

$$I \xrightarrow{X} I_{1,2} \iff I \xrightarrow{X} I_1 \quad \text{or} \quad I \xrightarrow{X} I_2$$

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:

$$\text{if } \vartheta(I_1, X) \neq \vartheta(I_2, X) \quad \text{then call } \text{Merge}(\vartheta(I_1, X), \vartheta(I_2, X))$$

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 *Merge* (Algorithm 4.53) to the members of every equivalence class, we construct a new graph called the *compact pilot*,²² denoted by \mathcal{C} .

Example 4.54 (Compact pilot) We reproduce (from the running example on p. 168 and 185) in Fig. 4.36 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, we will in a moment extract the initial states from the m-states, to obtain the already mentioned parser control-flow graph.

²²For 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 [1]) 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.

The look-ahead sets in the compact pilot \mathcal{C} are larger than in \mathcal{P} : e.g., in K_{17} 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 4.55 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 proof²³ 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 $I_{1,2} = \text{Merge}(I_1, I_2)$ 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. 187) 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 $\langle q, \{\dots\} \rangle$ 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.

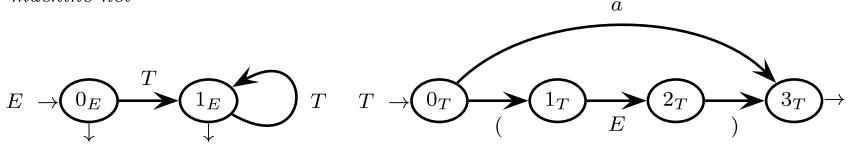
At the risk of repetition, we stress that Property 4.55 does not hold in general for an *ELR(1)* but not *STP*-compliant pilot, because the graph obtained by merging kernel-equivalent m-states is not guaranteed to be free from conflicts.

4.7.2.2 Candidate Identifiers or Pointers Unnecessary

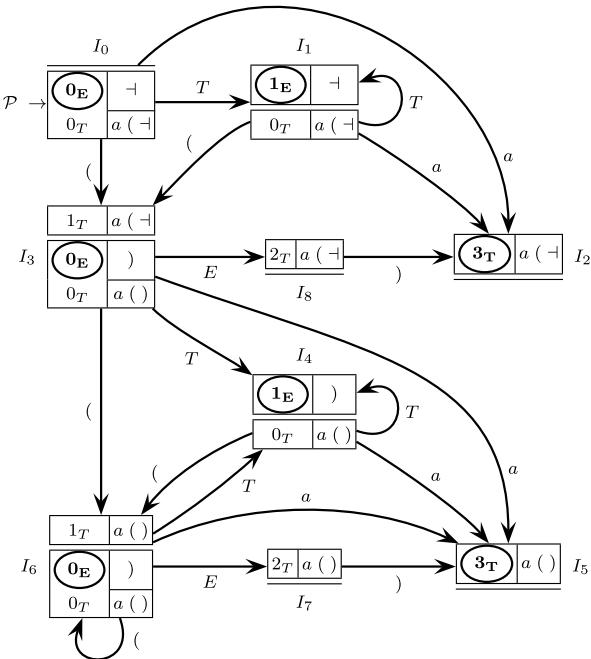
Thanks to the *STP* property, Algorithm 4.37 (p. 190) will be further simplified to remove the need for *cid*'s (or stack pointers). We recall that a *cid* was needed to find the reach of a non-empty reduction move into the stack: elements were popped

²³It can be found in [9].

machine net



pilot graph



compact pilot graph

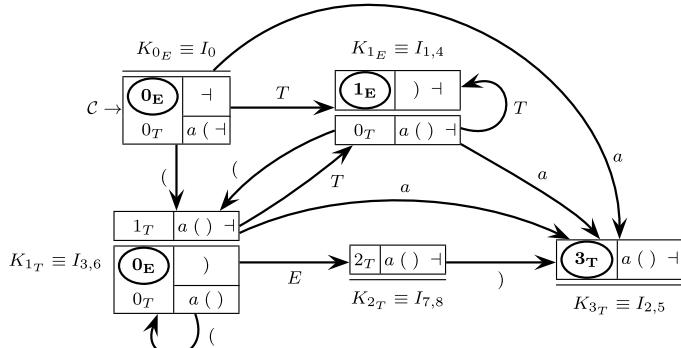


Fig. 4.36 From top to bottom: machine net \mathcal{M} , ELR(1) pilot graph \mathcal{P} and compact pilot \mathcal{C} ; the equivalence classes of m-states are: $\{I_0\}$, $\{I_1, I_4\}$, $\{I_2, I_5\}$, $\{I_3, I_6\}$, and $\{I_7, I_8\}$; the m-states of \mathcal{C} are named K_{0_E}, \dots, K_{3_T} to evidence their correspondence with the states of net \mathcal{M}

until the *cid* chain reached an initial state, the end-of-list sentinel. Under the *STP* hypothesis, that test is now replaced by a simpler device, to be later incorporated in the final *ELL(1)* parser (Algorithm 4.68 on p. 237). With reference to the “old” Algorithm 4.37, 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 $J|_{\text{base}} = \langle q_A, \dots \rangle$, 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 $\langle q_A, \dots \rangle$, 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. Having eliminated 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, 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*’s 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 4.37.

Algorithm 4.56 (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 $q_A \in K|_{\text{base}}$ (i.e., state q_A is not initial), eliminate all other candidates from H , i.e., set H equal to $K|_{\text{base}}$
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:

$$H[0]a_1H[1]a_2 \dots a_kH[k]$$

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 top-most stack element such that $0_A \in K[h]|_{\text{kernel}}$. The move does:

1. grow the syntax forest by applying the reduction:

$$a_{h+1}a_{h+2} \dots a_k \rightsquigarrow A$$

and pop the stack symbols:

$$H[k]a_k H[k-1]a_{k-1} \dots H[h+1]a_{h+1}$$

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 4.57 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 4.56 is equivalent to the *ELR(1)* parser \mathcal{A} of Algorithm 4.37.

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 $J[0] \dots J[k]$ and $K[0] \dots K[k]$. 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 4.56 may have discarded a few candidates.

Second, we examine the stack elements at the same positions i and $i - 1$. The following relations hold:

- in $J[i]$, candidate $\langle q_A, \pi, \sharp j \rangle$ points to $\langle p_A, \pi, \sharp l \rangle$ in $J[i-1]_{\text{base}}$
if and only if
candidate $\langle q_A, \pi \rangle \in K[i]$ and the only candidate in $K[i-1]$ is $\langle p_A, \pi \rangle$
- in $J[i]$, candidate $\langle q_A, \pi, \sharp j \rangle$ points to $\langle 0_A, \pi, \perp \rangle$ in $J[i-1]_{\text{closure}}$
if and only if
candidate $\langle q_A, \pi \rangle \in K[i]$
and elem. $K[i-1]$ equals the projection of $J[i-1]$ on $\langle \text{state}, \text{look-ahead} \rangle$

Then by the way the reduction moves operate, parser \mathcal{A} performs reduction $a_{h+1}a_{h+2} \dots a_k \rightsquigarrow A$ 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 4.58 (Pointerless parser trace for the compact pilot (Fig. 4.36, p. 221)) Given the input string $((0)a) \dashv$, Fig. 4.37 shows the execution trace, which should be compared with the one of the *ELR(1)* parser in Fig. 4.18, p. 193, where *cid*'s 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., (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 4.56 from an m-state are striked 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 $(E) \sim T$ 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 striked out) candidate 0_T , which is initial for machine M_T , but is not the correct initial state for the reduction: the correct one, unstruck, 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 4.56 avoids to cancel an initial candidate from a stack element, if a shift move is to be later executed starting from it: see the *Shift move* case in Algorithm 4.56. 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 put 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 $TT \rightsquigarrow E$), 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.

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.

stack bottom	string to be parsed (with end-marker) and stack contents						effect after	
	1	2	3	4	5	+		
$0_E \quad 0_T$	(()	a)	+	initialisation of the stack	
	(()	a)	+	shift on (
	1_T $\boxed{1_T} \quad \overbrace{0_E}$ 0_T							
	(()	a)	+	shift on (
	1_T $\boxed{1_T} \quad \overbrace{0_E}$ 0_T	1_T $\boxed{1_T} \quad \overbrace{0_E}$ 0_T					reduction $\varepsilon \rightsquigarrow E$	
	((E)	a)	+	
	1_T $\boxed{1_T} \quad \overbrace{0_E}$ 0_T	1_T $\boxed{1_T} \quad \overbrace{0_E}$ 0_T					shifts on E and)	
	1_T $\boxed{1_T} \quad \overbrace{0_E}$ 0_T	1_T $\boxed{1_T} \quad \overbrace{\theta_E}$ θ_T	2_T 2_T	3_T $\circled{3_T}$				
	(T		a)	+		
	1_T $\boxed{1_T} \quad \overbrace{0_E}$ 0_T	1_E $\boxed{1_E}$ 0_T					reduction $(E) \rightsquigarrow T$ and shift on T	
	(T	a)	+			
	1_T $\boxed{1_T} \quad \overbrace{0_E}$ 0_T	1_E $\circled{1_E}$ 0_T	3_T $\circled{3_T}$				shift on a	
	(T	T)	+			
	1_T $\boxed{1_T} \quad \overbrace{0_E}$ 0_T	1_E $\circled{1_E}$ θ_T	1_E $\circled{1_E}$ 0_T				reduction $a \rightsquigarrow T$ and shift on T	
	(E)	+			
	1_T $\boxed{1_T} \quad \overbrace{\theta_E}$ θ_T	2_T 2_T	3_T $\circled{3_T}$				reduction $TT \rightsquigarrow E$ and shifts on E and)	
		T				+		
		1_E $\circled{1_E}$ 0_T					reduction $(E) \rightsquigarrow T$ and shift on T	
		E				+	reduction $T \rightsquigarrow E$ and accept (do not shift on E)	

Fig. 4.37 Steps of the pointerless parsing Algorithm 4.56 \mathcal{A}_{PL} ; irrelevant and potentially confusing initial candidates are canceled by shift moves as explained in the algorithm

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 on-line the left derivation of the input.

Parser Control-Flow Graph 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. Finally, 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 4.59 (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 to 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:

$$\pi = \bigcup_{\forall \langle f_A, \pi_i \rangle \in \mathcal{C}} \pi_i \quad (4.8)$$

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 $q_A \xrightarrow{\gamma_1} 0_{A_1}$, 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 $\langle r_A, \pi_{r_A} \rangle$.

²⁴Although 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.

The call arc label $\gamma_1 \subseteq \Sigma \cup \{-\}$, 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 4.60 (Guide set)

1. For every call arc $q_A \xrightarrow{\gamma_1} 0_{A_1}$ 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:

$$b \in Ini(L(0_{A_1})) \quad (4.9)$$

$$A_1 \text{ is nullable and } b \in Ini(L(r_A)) \quad (4.10)$$

$$A_1 \text{ and } L(r_A) \text{ are both nullable and } b \in \pi_{r_A} \quad (4.11)$$

$$\exists \text{ in } \mathcal{F} \text{ a call arc } 0_{A_1} \xrightarrow{\gamma_2} 0_{A_2} \text{ and } b \in \gamma_2 \quad (4.12)$$

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 $Gui(f_A \rightarrow) := \pi$.

Guide sets are usually represented as arc labels enclosed within braces.

Relations (4.9), (4.10), and (4.11) 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 . Relation (4.12) is recursive and traverses the net as far as the chain of call sites activated. We observe that relation (4.12) 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 [9] for a proof).

Property 4.61 (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 4.50.

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)* gram-

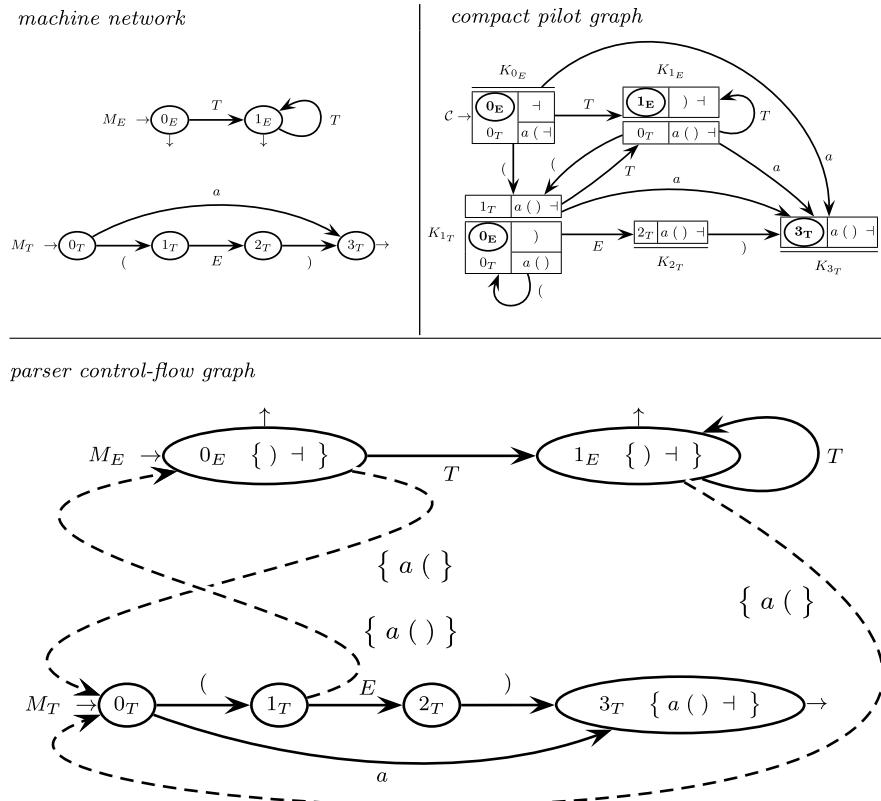


Fig. 4.38 The Parser Control-Flow Graph \mathcal{F} (PCFG) of the running example, from the net and the compact pilot

mars. We will later see that this condition can be directly checked on the PCFG, thus saving the effort to build the $ELR(1)$ pilot.

It is time to illustrate the previous definitions and properties.

Example 4.62 (PCFG of the running example) The PCFG of the running example is represented in Fig. 4.38, 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} . Having created 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 relation (4.8)):

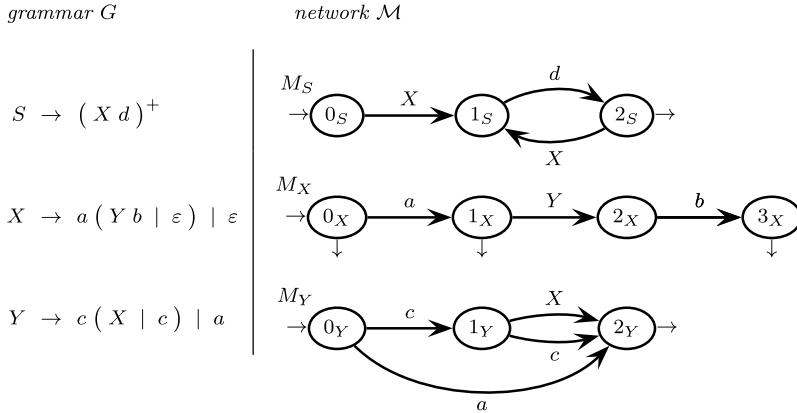


Fig. 4.39 Grammar and net of Example 4.63

- 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 .
- 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, $\langle 0_E, \{')\}, \dashv \rangle$ 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 $\{a, '(\}\}$, since both terminals a and $'($ can be shifted starting from state 0_T , which is the call arc destination (see relation (4.9))
- 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 $\{a, '(\}\}$ (see relation (4.12))
 - $)'$, 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 we have $)' \in Ini(2_T)$ (see relation (4.11)).

Observe the chain of two call arcs, $1_T \dashrightarrow 0_E$ and $0_E \dashrightarrow 0_T$; relation (4.12) implies that the guide set of the former arc includes that of the latter.

In accordance with Property 4.61, the terminal labels of all the arcs (shift and call) that originate from the same node, do not overlap.

Since relation (4.10) has not been used, we illustrate it in the next example.

Example 4.63 Consider the EBNF grammar G and net \mathcal{M} in Fig. 4.39.

The machine net \mathcal{M} is $ELL(1)$; its pilot \mathcal{P} in Fig. 4.40, 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. 4.41.

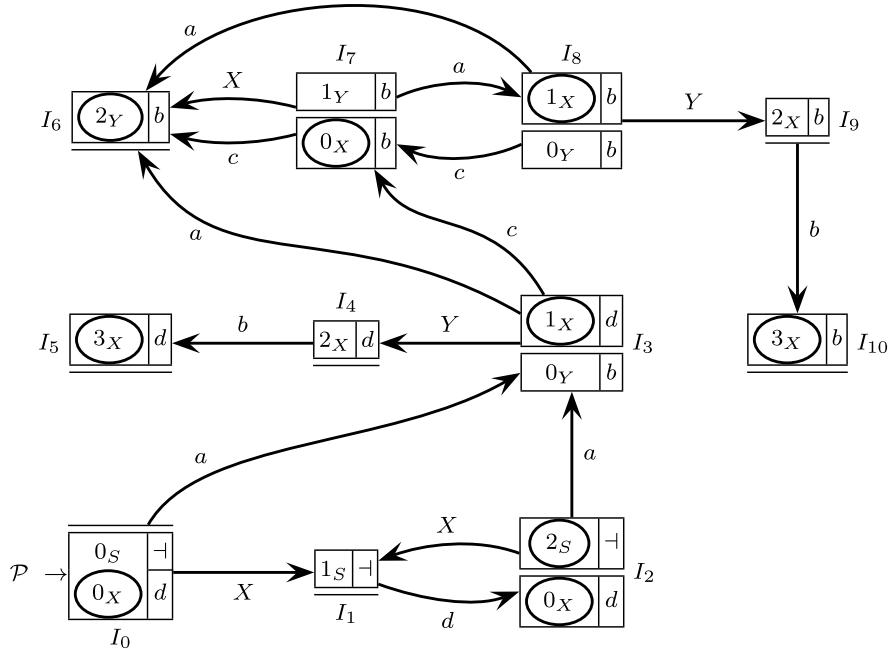


Fig. 4.40 ELR(1) pilot of the machine net in Fig. 4.39

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 relation (4.8)):

- 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

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 relation (4.9)) 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 relation (4.10))
- $Gui(2_S \dashrightarrow 0_X)$ is the union of terminal a on the shift arc from 0_X (see relation (4.9)) 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 relation (4.10))
- $Gui(1_Y \dashrightarrow 0_X)$ is the union of terminal a on the shift arc from 0_X (see relation (4.9)) 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

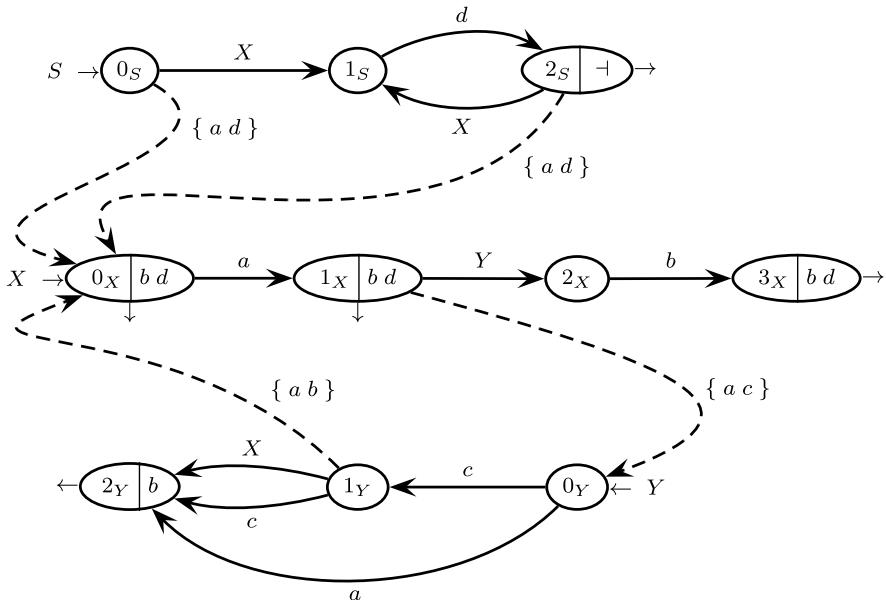


Fig. 4.41 Parser control-flow graph derived from the pilot graph in Fig. 4.40

nonterminal X is nullable and additionally state 2_Y is final, i.e., language $L(2_Y)$ is nullable (see relation (4.11))

- $Gui(1_X \dashrightarrow 0_Y)$ includes only the terminals a and c that label the shift arcs from state 0_Y (see relation (4.9)); 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 one call arc $1_Y \dashrightarrow 0_Y$, 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 (4.9), (4.10) and (4.11) for computing the guide sets on call arcs. Relation (4.12) is not used as in the PCFG concatenated call arcs do not occur (that case has been already illustrated in the preceding example in Fig. 4.38).

Having thus obtained the PCFG of a net, a short way remains to construct the deterministic predictive parser.

4.7.3 Direct Construction of Top-Down Predictive Parsers

Section 4.7.2 has rigorously derived step-by-step the PCFG starting from the ELR(1) pilot graph of the given machine net, by assuming it complies with the ELL(1) condition on p. 214. 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.

For the theoretically interested reader, we add that the next formulation of the *ELL(1)* condition includes some marginal cases that violate the *STP* clause of the pilot-based Definition 4.50. We exhibit a contrived example, where a violation of *STP*, due to the multiple candidates in the base of an m-state, does not hinder a top-down parser from working deterministically [7]. Consider the *LR(1)* grammar $S \rightarrow Aa \mid Bb$, $A \rightarrow C$, $B \rightarrow C$ and $C \rightarrow \varepsilon$. Without having to draw the pilot, it is evident that one of the m-states has two candidates in the base: $\langle A \rightarrow C \bullet, a \rangle$ and $\langle B \rightarrow C \bullet, b \rangle$ (here we use marked rules instead of machine states). Yet the choice between the alternatives of the axiom S is determined by the following character, a or b . A necessary condition for such a situation to occur is that the language derived from some nonterminal of the grammar consists *only* of the empty string. However, this case can be usually removed without penalty in all grammar applications: thus, the grammar above can be replaced by the equivalent simpler grammar $S \rightarrow A \mid B$, $A \rightarrow a$ and $B \rightarrow b$, which complies with *STP* and is *ELL(1)*.

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 such a call arc is a set of terminals, named a *guide set*. Intuitively, such a call arc represents a (possibly recursive) conditional invocation of the parsing procedure associated with machine M_B , subject 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 Eqs. (4.13), (4.14) on p. 233.

guide sets Let us name *arrow* a terminal shift arc, a call arc, and also a dangling arrow (named *dart*) 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 $q_A \xrightarrow{\gamma} 0_B$ or, equivalently, $\gamma = Gui(q_A \dashrightarrow 0_B)$; its exact computation follows in Eq. (4.15) on p. 233
- 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 4.61, p. 227).

Definition 4.64 (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

$$\pi_{0_A} := \pi_{0_A} \cup \bigcup_{q_i \xrightarrow{A} r_i} (\text{Ini}(L(r_i)) \cup \text{if } \text{Nullable}(L(r_i)) \text{ then } \pi_{q_i} \text{ else } \emptyset) \quad (4.13)$$

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

$$\pi_q := \bigcup_{p_i \xrightarrow{X_i} q} \pi_{p_i} \quad (4.14)$$

The two sets of rules apply in an exclusive way to disjoint 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 the prospect set.

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 $\text{Gui}(q_A \dashrightarrow 0_{A_1})$ of the call arc is defined as follows, see also rules (4.9)–(4.12) at p. 227:

$$\text{Gui}(q_A \dashrightarrow 0_{A_1}) := \bigcup \left\{ \begin{array}{l} \text{Ini}(L(A_1)) \\ \hline \text{if } \text{Nullable}(A_1) \text{ then } \text{Ini}(L(r_A)) \\ \text{else } \emptyset \text{ endif} \\ \hline \text{if } \text{Nullable}(A_1) \wedge \text{Nullable}(L(r_A)) \text{ then } \pi_{r_A} \\ \text{else } \emptyset \text{ endif} \\ \hline \bigcup_{0_{A_1} \dashrightarrow 0_{B_i}} \text{Gui}(0_{A_1} \dashrightarrow 0_{B_i}) \end{array} \right\} \quad (4.15)$$

2. For a final state $f_A \in F_A$, the guide set of the tagging dart equals the prospect set:

$$Gui(f_A \rightarrow) := \pi_{f_A} \quad (4.16)$$

3. For a terminal shift arc $q_A \xrightarrow{a} r_A$ with $a \in \Sigma$, the guide set is simply the shifted terminal:

$$Gui(q_A \xrightarrow{a} r_A) := \{a\} \quad (4.17)$$

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.

Notice that the rules for computing the prospect sets are consistent with the definition of look-ahead set given in Sect. 4.5.2; furthermore, the rules for computing the guide sets are consistent with the definition of parser control-flow graph listed in Definition 4.59 on p. 226.

Example 4.65 (PCFG for list of balanced strings) The net in Fig. 4.29, p. 212, is now represented by the PCFG in Fig. 4.42.

The following table shows the computation of the prospect sets for the PCFG.

Node	Init.	Equation	Arc instance	Prospect set
0_S	$\{\dashv\}$			$\{\dashv\}$
1_S	\emptyset	(4.14)	$0_S \xrightarrow{P} 1_S$	$\{\dashv\}$
2_S	\emptyset	(4.14)	$1_S \xrightarrow{'} 2_S$	$\{\dashv\}$
0_P	\emptyset	(4.13)	$0_S \xrightarrow{P} 1_S$ $2_S \xrightarrow{P} 1_S$ $1_P \xrightarrow{P} 2_P$	$\{‘,’\} \cup \{\dashv\} \cup$ $\{‘,’\} \cup \{\dashv\} \cup$ $\{a\} \cup \emptyset = \{a, ‘,’, \dashv\}$
1_P	\emptyset	(4.14)	$0_P \xrightarrow{a} 1_P$	$\{a, ‘,’, \dashv\}$
2_P	\emptyset	(4.14)	$1_P \xrightarrow{P} 2_P$	$\{a, ‘,’, \dashv\}$
3_P	\emptyset	(4.14)	$2_P \xrightarrow{P} 3_P$ $0_P \xrightarrow{P} 3_P$	$\{a, ‘,’, \dashv\} \cup$ $\{a, ‘,’, \dashv\} = \{a, ‘,’, \dashv\}$

The guide sets are computed by three equations, but the last two are trivial and we comment only Eq. (4.15) 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.

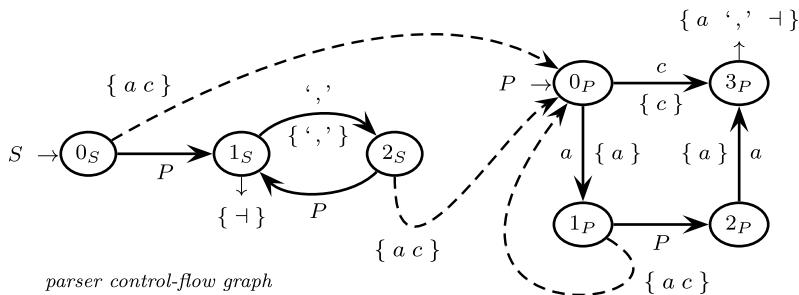


Fig. 4.42 Parser control-flow graph (PCFG) for Example 4.65 (formerly Example 4.48, p. 211); the call arcs are *dashed*; the guide sets are *in braces*

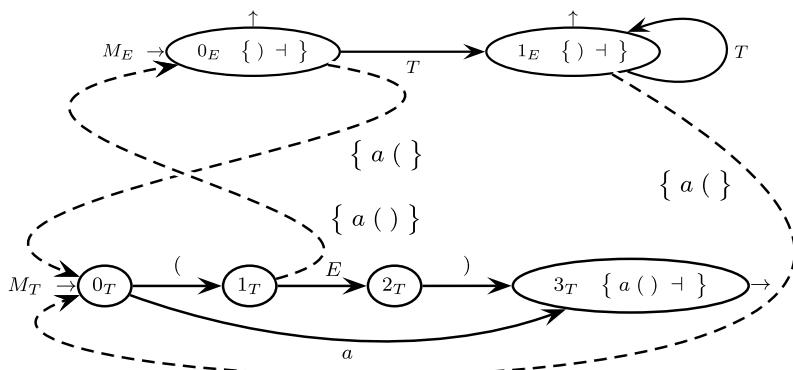


Fig. 4.43 The parser control-flow graph (PCFG) of the running example

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 4.66 (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. 4.38 (p. 228) here reproduced in Fig. 4.43. For both tables, the computation is completed at the third step.

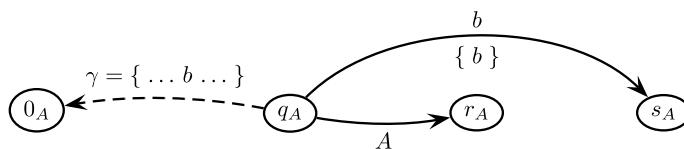
Prospect sets of states					
0_E	1_E	0_T	1_T	2_T	3_T
\dashv	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
$) \dashv$	$) \dashv$	$a() \dashv$	$a() \dashv$	$a() \dashv$	$a() \dashv$
$) \dashv$	$) \dashv$	$a() \dashv$	$a() \dashv$	$a() \dashv$	$a() \dashv$

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 $1_T \xrightarrow{\gamma_1} 0_E \xrightarrow{\gamma_2} 0_T$, 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



By Eq. (4.15) the guide set γ of the call arc $q_A \dashrightarrow 0_A$ includes set $Ini(L(A))$, 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 4.67 (Guide sets in a non-*ELL(1)* grammar) The grammar with rules $S \rightarrow a^*N$ and $N \rightarrow aNb \mid \varepsilon$ of Example 4.52 (p. 216), violates the singleton base property as shown in Fig. 4.34, 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 push-down 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 4.68 (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 4.61 or Definition 4.64), 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 to use the rules of the equivalent *right-linearized (RLZ) grammar* \hat{G} , as anticipated in Sect. 4.5.1, p. 171. 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 RLZ 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 4.5 reports the output action, i.e., the rule of \hat{G} , which is printed.

The correspondence is so straightforward that it does not need justification and an example suffices.

Table 4.5 Derivation steps computed by a predictive parser; the current token is $cc = b$

Parser move	RL rule used
Scan move for transition $q_A \xrightarrow{b} r_A$	$q_A \xrightarrow[\hat{G}]{} br_A$
Call move for call arc $q_A \xrightarrow{\gamma} 0_B$ and trans. $q_A \xrightarrow{0_B} r_A$	$q_A \xrightarrow[\hat{G}]{} 0_B r_A$
Return move from state $q_A \in F_A$	$q_A \xrightarrow[\hat{G}]{} \varepsilon$

Example 4.69 (Running example: trace of predictive parser DPDA) The RLZ grammar (from Example 4.22, p. 171) is the following:

$$\begin{array}{ll} 0_E \rightarrow 0_T 1_E \mid \varepsilon & 1_E \rightarrow 0_T 1_E \mid \varepsilon \\ 0_T \rightarrow a 3_T \mid '1_T & 1_T \rightarrow 0_E 2_T \\ 2_T \rightarrow '3_T & 3_T \rightarrow \varepsilon \end{array}$$

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. 4.43, p. 235.

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	$0_E \xrightarrow[\dagger]{} (1_T 1_E$
$\langle 1_E \rangle \langle 1_T \rangle$	$a \dashv$	$a \in \gamma = \{a()\}$	$0_E \xrightarrow[\dagger]{} (0_E 2_T 1_E$
$\langle 1_E \rangle \langle 2_T \rangle \langle 0_E \rangle$	$a \dashv$	$a \in \gamma = \{a()\}$	$0_E \xrightarrow[\dagger]{} (0_T 1_E 2_T 1_E$
$\langle 1_E \rangle \langle 2_T \rangle \langle 1_E \rangle \langle 0_T \rangle$	$a \dashv$	scan	$0_E \xrightarrow[\dagger]{} (a 3_T 1_E 2_T 1_E$
$\langle 1_E \rangle \langle 2_T \rangle \langle 1_E \rangle \langle 3_T \rangle$	$) \dashv$	$) \in \pi = \{a() \dashv\}$	$0_E \xrightarrow[\dagger]{} (a \varepsilon 1_E 2_T 1_E$
$\langle 1_E \rangle \langle 2_T \rangle \langle 1_E \rangle$	$) \dashv$	$) \in \pi = \{() \dashv\}$	$0_E \xrightarrow[\dagger]{} (a \varepsilon 2_T 1_E$
$\langle 1_E \rangle \langle 2_T \rangle$	$) \dashv$	scan	$0_E \xrightarrow[\dagger]{} (a) 3_T 1_E$
$\langle 1_E \rangle \langle 3_T \rangle$	\dashv	$\dashv \in \pi = \{a() \dashv\}$	$0_E \xrightarrow[\dagger]{} (a) \varepsilon 1_E$
$\langle 1_E \rangle$	\dashv	$\dashv \in \pi = \{() \dashv\}$ accept	$0_E \xrightarrow[\dagger]{} (a) \varepsilon$

For the original EBNF grammar, the corresponding derivation is:

$$E \Rightarrow T \Rightarrow (E) \Rightarrow (T) \Rightarrow (a)$$

Parser Implementation by Recursive Procedures We have seen in the introduction to predictive parsers at p. 212 that they are often implemented using recursive procedures. Each machine is transformed into a parameter-less *syntactic procedure*,

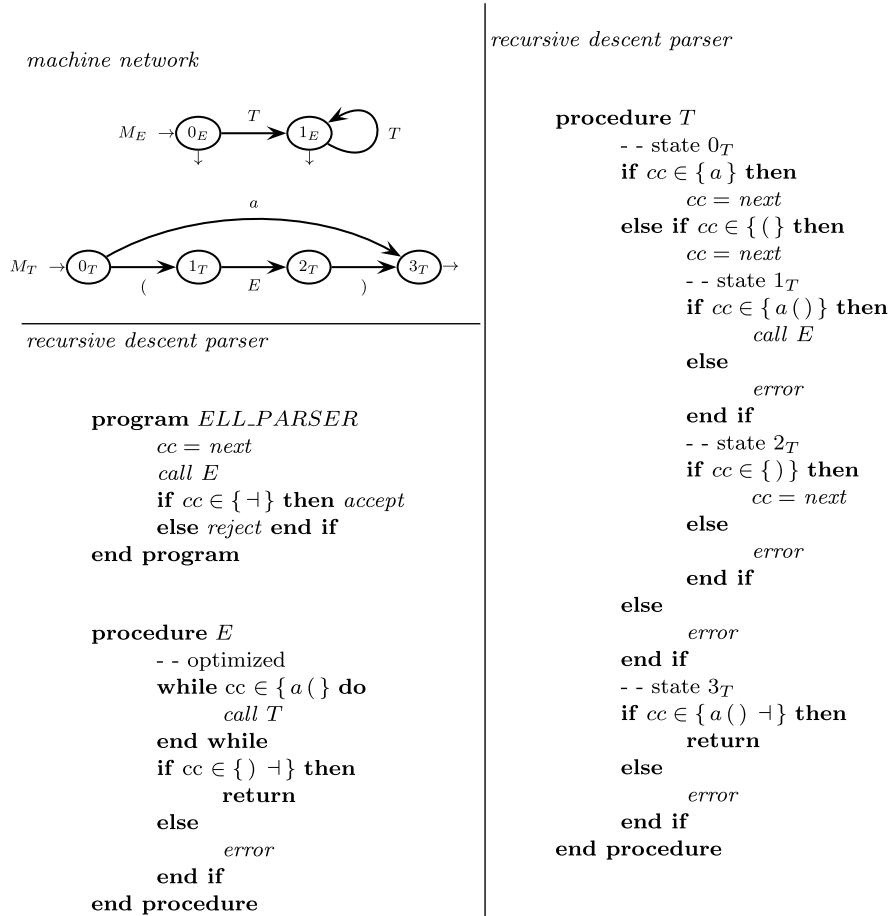


Fig. 4.44 Main program and syntactic procedures of a recursive-descent parser (Example 4.70 and Fig. 4.43); function *next* is the programming interface to the lexical analyzer or scanner; function *error* is the messaging interface

having a control-flow graph matching the corresponding *PCFG* subgraph, so that the current state of a machine is encoded at runtime 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 runtime 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 4.70 (Recursive-descent parser) From the *PCFG* of Fig. 4.43, p. 235, we obtain the syntactic procedures shown in Fig. 4.44. The pseudo-code can be optimized in several ways, which we leave out.

Any practical parser has to cope with input errors, provide a reasonable error diagnosis, and be able to resume parsing after an error. A few hints on error management in parsing will be presented in Sect. 4.12.

4.7.4 Increasing Look-Ahead

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 4.68 on p. 237 (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 a computation example of a few guide sets of length $k = 2$.

Example 4.71 (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*).²⁵

$$\begin{aligned} \langle \text{progr} \rangle &\rightarrow [\langle \text{label} \rangle \text{`}:] \langle \text{stat} \rangle (\text{`} ; \langle \text{stat} \rangle)^* \\ \langle \text{stat} \rangle &\rightarrow \langle \text{assign_stat} \rangle \mid \langle \text{for_stat} \rangle \mid \dots \\ \langle \text{assign_stat} \rangle &\rightarrow \text{id} \text{`}=’ \langle \text{expr} \rangle \\ \langle \text{for_stat} \rangle &\rightarrow \text{for } \text{id} \dots \\ \langle \text{label} \rangle &\rightarrow \text{id} \\ \langle \text{expr} \rangle &\rightarrow \dots \end{aligned}$$

The net machines are depicted in Fig. 4.45 with the relevant guide sets. To avoid clogging the PCFG, 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

²⁵The symbols in the hook brackets $\langle \rangle$ are nonterminals, all the others are terminals or are assumed to be terminals; and the square brackets mean their contents are optional.

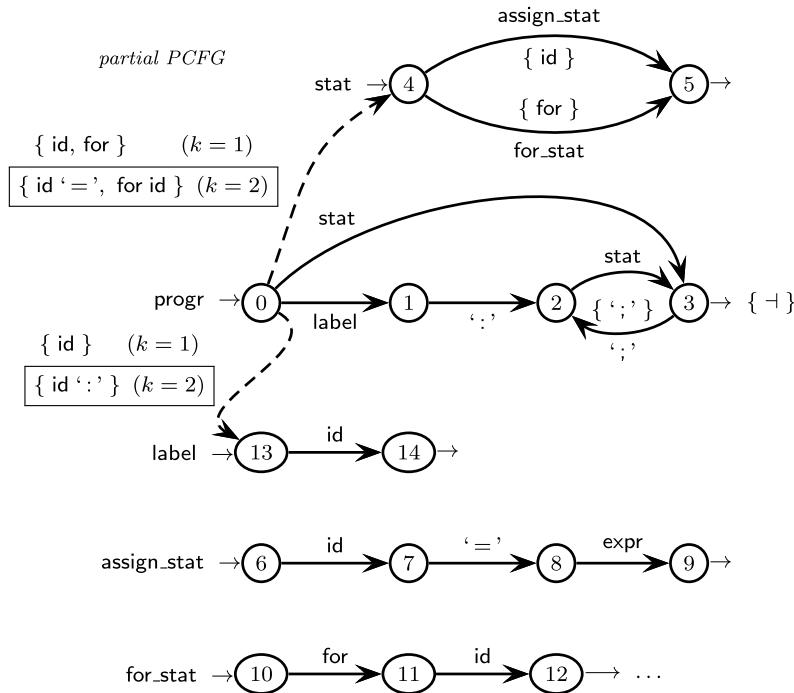


Fig. 4.45 Machine net and the relevant parts of the parser control-flow graph of Example 4.71. With look-ahead length $k = 2$ the graph gets deterministic

exiting the furcation nodes. The states are simply enumerated and the hooks around the 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, share the terminal symbol id ; thus these guide sets 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.²⁶ 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 runtime by the parser. Notice that in the bifurcation states 3 and 4, a look-ahead $k = 1$ suffices to make the choice deterministic; in fact see that the guide sets

²⁶In 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.

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.

Formally the elements of an $ELL(k)$ guide set are strings of length up to k terminals.²⁷ 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 [30], which uses a look-ahead with variable and unbounded length, depending on the current 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, 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 Deterministic Language Families: A Comparison

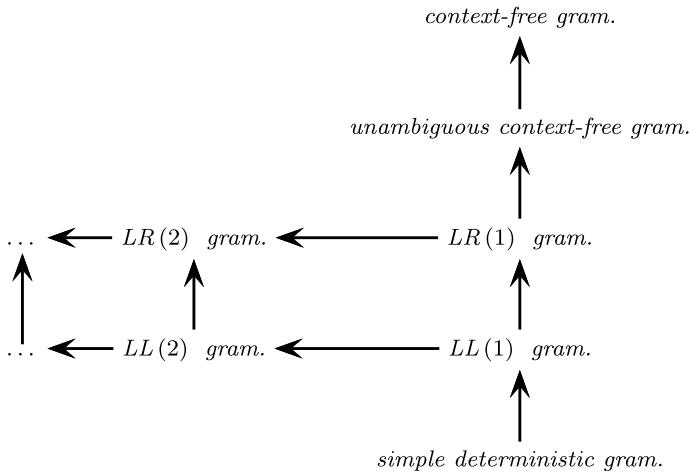
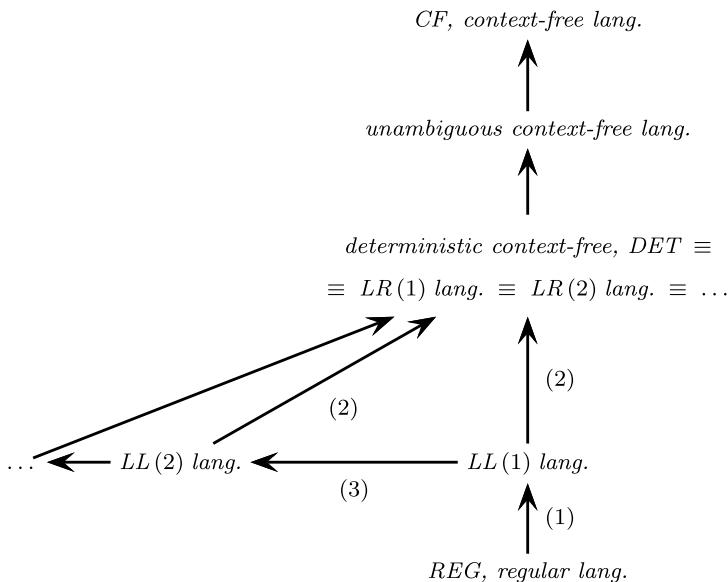
The family DET of *deterministic* (context-free) *languages*, characterized by the deterministic pushdown automaton as their recognizer, was defined in Sect. 4.3.2, p. 154, as well as the subclasses of *simple deterministic* (p. 161) and *parenthesis languages*. To complete the theoretical picture of deterministic families, we include in the comparison the families of deterministic bottom-up and top-down parsable 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 that hold for the latter are obviously extended to the former.

As we proceed, we have to clarify what we actually compare: grammar families 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. 4.46 and 4.47, respectively, and are now discussed case by case. Notice that some facts already known from previous chapters and sections are not repeated here.

We briefly comment the inclusions in Fig. 4.46, which are quite immediate. In the *simple deterministic* case, for every nonterminal A the grammar rules must start with distinct terminals, i.e., they are $A \rightarrow b\beta$ and $A \rightarrow c\gamma$ with $b \neq c$. Therefore

²⁷They may be shorter than k only when they end with the end-marker \dashv .

**Fig. 4.46** Inclusion relationships between families of grammars**Fig. 4.47** Inclusion relationships between families of languages; all the inclusions are strict; the numbers refer to the comments

the guide sets of the corresponding bifurcation from the initial state of machine M_A are disjoint, thus proving that the grammar is $LL(1)$. 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 4.7.4 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. 4.45 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 [38, 39] cover such properties in depth) that the cross inclusion of $LL(k)$ within $LR(k)$, which we already know to hold for $k = 1$, is valid for any value of k .

We compare several language families in Fig. 4.47 (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 [18].

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 4.72 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 [25]) 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 family $LR(1)$ is included in the $ELR(1)$ one, the latter too coincides with DET . On the other hand, any context-free language that is not deterministic, as the cases in Sect. 4.3.2.2, p. 157, 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 a pilot with length $k = 1$, because some m-states of the latter pilot are split, due to the look-ahead sets with $k = 2$ 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, with the only peculiarity of possibly having more internal states than an $ELR(1)$ parser, from the preceding property (Property 4.72) we have the following one:

Property 4.73 For every integer $k > 1$, the family of the languages generated by $ELR(k)$ grammars coincides with the family of the languages 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. 4.46, 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 4.73 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.

We complete this comparison topic with a twofold negative statement.

Property 4.74 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.

The following comments are keyed to the numbered relationships among languages, shown in Fig. 4.47.

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 $p \rightarrow aq$, and each final state f as $f \rightarrow \varepsilon$. 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 \dashv .

2. Every $LL(k)$ language with $k \geq 1$ is deterministic, hence also $LR(1)$ by Property 4.72. 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. 4.34, p. 217, where the grammar with rules $S \rightarrow a^*N$ and $N \rightarrow aNb \mid \epsilon$, 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 $a \dots aa^n b^n$ where the balanced part of the string is preceded by unboundedly many letters a . A natural question is the following: can we find an equivalent $ELL(k)$ grammar for this language? The answer, proved in [6], is negative.
3. The collection of language families defined by $LL(k)$ grammars, for $k = 1, 2, \dots$, form 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 [6, 38, 39]). Notice the contrast with the $LR(k)$ collection of language families, which all collapse into DET for every value of the parameter k .

To finish with Fig. 4.47, the fact that a deterministic language is unambiguous is stated in Property 4.14 on p. 159.

4.9 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. 4.8 (p. 242 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

re-factoring the alternative grammar rules, so that the guide sets become disjoint. For instance, the rules $X \rightarrow A \mid B$, $A \rightarrow a \mid cC$ and $B \rightarrow b \mid cD$ are re-factored 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. However, the construction of the pilot automaton can be done by hand only for toy grammars and at present we only know of tools for pure *BNF* grammars. At the moment, the practical advantage of the top-down approach is its direct handling of *EBNF* grammars, in contrast to the current limitation of the popular *LR* parser generation tools that do not accept *EBNF* rules. Awaiting for some new *ELR(1)* parser generators to come, the annoying but not serious consequence is that the regular expressions present in the rules must be eliminated by means of standard transformations.

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.

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 Coke–Younger–Kasami) and Earley (see for instance [17]); 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 look-ahead in order to decide the next move. For bottom-up parsing, a practical almost deterministic algorithm existing in several variants is due to Tomita [40]. 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.

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.

4.10 A General Parsing Algorithm

The *LR* and *LL* parsing methods are inadequate for dealing with ambiguous and nondeterministic grammars. In his seminal work [13], 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 [35]. Until very recently [2], the Earley parsers computed the syntax tree in a second pass, after string recognition [17].

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 [3], we 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 [17, 31].

We present a version of the Earley parser for 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 the pure recognizer algorithm is defined and illustrated, and the algorithm for building the syntax tree is presented. At the end, we mention some extensions of the Earley algorithm and other related issues.

4.10.1 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, p. 201, is a good analogy to the *Earley vector* to be next introduced. When analyzing a string $x = x_1 \dots x_n$ or $x = \varepsilon$ of length $n \geq 0$, the algorithm uses a vector $E[0 \dots n]$ 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 $E[j]$, which precedes or coincides with $E[i]$, and contains a pair $\langle 0_X, j \rangle$, with the same machine M_X and the same integer j . This pointer j marks the position \uparrow in the input string x from where the current derivation of nonterminal X has started, i.e., $x_1 \dots x_j \uparrow \overbrace{x_{j+1} \dots x_i \dots x_n}^X$, namely from character x_j , excluded, to the right as far as character x_i , included (case $j = i$ is for the null string $x = \varepsilon$).

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 S .

An intuitive example familiarizes us with the Earley method. It embodies concepts but leaves out refinements, which are added later in a formal way.

Example 4.75 (Introduction to the Earley method) The non-deterministic language:

$$\{a^n b^n \mid n \geq 1\} \cup \{a^{2n} b^n \mid n \geq 1\}$$

is defined by the grammar and machine net in Fig. 4.48, which violate the *ELR(k)* condition, as any other equivalent grammar would do (the *ELR(k)* pilot is in Fig. 4.15 on p. 188).

Suppose the string to be analyzed is $x = x_1 x_2 x_3 x_4 = aabb$ with length $n = 4$. We prepare the Earley vector E , with the elements $E[0], \dots, E[4]$, all initialized to empty, except $E[0] = \langle 0_S, 0 \rangle$. In this pair, the *state* is set to 0_S , i.e., to 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 Earley vector is shown in Table 4.6, with all the pairs progressively put into it.

In broad terms, three operations types, to be next applied and explained on the example, can be carried out on the current element $E[i]$: *closure*, *terminal shift*, and *nonterminal shift*; their names are mindful of the similar *ELR* operations (p. 174).

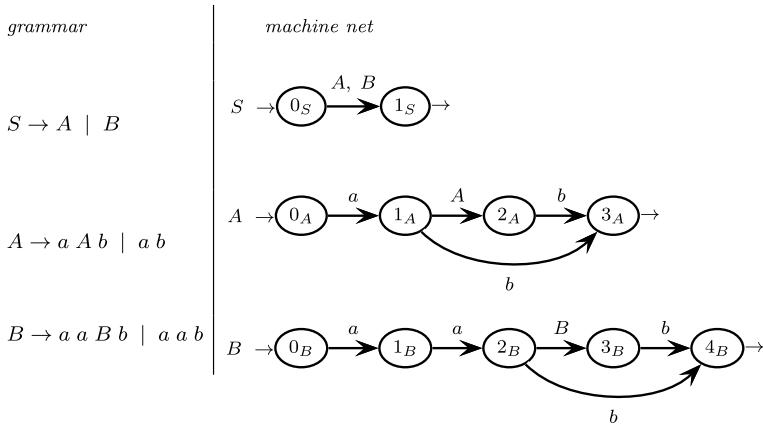


Fig. 4.48 Grammar and machine net of Example 4.75

Closure It applies to a pair with a state from where an arc with a nonterminal label X originates. Suppose that the pair $\langle p, j \rangle$ is in the element $E[i]$ and that the net has an arc $p \xrightarrow{X} q$ with a nonterminal label X ; the destination state q of the arc is not relevant. The operation adds a new pair $\langle 0_X, i \rangle$ to the same element $E[i]$. The state of this pair is the initial one 0_X of the machine M_X of that nonterminal X , and the pointer has value i , which means that the pair is created at step i starting from a pair already in the element $E[i]$.

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} . In our example, the closure writes into $E[0]$ the pairs:

$$\langle 0_A, 0 \rangle \quad \langle 0_B, 0 \rangle$$

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 pair $\langle p, j \rangle$ is in element $E[i-1]$ and that the net has arc $p \xrightarrow{x_i} q$ labeled by 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 $x_1 = a$, from element $E[0]$ terminal shift puts into element $E[1]$ the pairs:

$$\langle 1_A, 0 \rangle \quad \langle 1_B, 0 \rangle$$

Then closure applies again to the new pairs in element $E[1]$, with the effect of adding to $E[1]$ the pair (where we notice the pointer has value 1):

$$\langle 0_A, 1 \rangle$$

If at step i no terminal shift can be applied to a pair in element $E[i - 1]$, thus causing the next element $E[i]$ to remain empty, then the algorithm stops and rejects the string. In the example, however, this never happens.

Continuing from element $E[1]$ and scanning character $x_2 = a$, through terminal shift we put into $E[2]$ the pairs:

$$\langle 2_B, 0 \rangle \quad \langle 1_A, 1 \rangle$$

and, through their closure, we add to $E[2]$ the pairs:

$$\langle 0_B, 2 \rangle \quad \langle 0_A, 2 \rangle$$

Scanning the next character $x_3 = b$, through terminal shift we write into $E[3]$ the pairs:

$$\langle 4_B, 0 \rangle \quad \langle 3_A, 1 \rangle$$

Now the third operation, which is the most complex, comes into play.

Nonterminal Shift The operation is triggered by the presence in $E[i]$ of a final pair $\langle f_X, j \rangle$, to be qualified as *enabling*. Then, to locate the pairs to be shifted, the operation goes back to the element $E[j]$ referred to the pointer of the enabling pair. In this example the pointer always directs to a preceding element, i.e., we have $j < i$. However, if the grammar contains nullable nonterminals, the pointer j may also direct to $E[i]$, as we will see.

In the element $E[j]$, the nonterminal shift operation searches for a pair $\langle p, l \rangle$ such that the net contains an arc $p \xrightarrow{X} q$, with a label that matches the machine of state f_X in the enabling pair. The pointer l may range in the interval $0 \leq l \leq j$. It is certain 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 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 put into 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 $E[i - 1]$ to $E[i]$, two adjacent elements, and uses an arc $p \xrightarrow{x_i} q$, which scans token x_i ; the latter works from element $E[j]$ to $E[i]$, which can be separated by a long

Table 4.6 Earley parsing trace of string $aabb$ for Example 4.75

Earley vector $E[0 \dots 4]$								
$E(0)$	x_1	$E(1)$	x_2	$E(2)$	x_3	$E(3)$	x_4	$E(4)$
q	j	a	q	j	a	q	j	
0_S	0	1_A	0	2_B	0	4_B	0	3_A
0_A	0	1_B	0	1_A	1	3_A	1	
0_B	0	0_A	1	0_B	2	1_S	0	2_A
				0_A	2			

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. If string x ended at this point, then its prefix aab 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. Actually one more character $x_4 = b$ is 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 $x = aabb$ to be accepted. The whole computation trace is in Table 4.6.

The divider on 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.

4.10.2 Earley Algorithm

The Earley algorithm for EBNF grammars uses two procedures, *Completion* and *TerminalShift*. The input string is denoted $x = x_1 \dots x_n$ or $x = \varepsilon$, with $|x| = n \geq 0$. Here is *Completion*:

$\text{Completion}(E, i)$ - - with index $0 \leq i \leq n$

do

- - loop that computes the *closure* operation
- for each pair that launches machine M_X

for (each pair $\langle p, j \rangle \in E[i]$ and $X, q \in V, Q$ s.t. $p \xrightarrow{X} q$) **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 (each pair $\langle f, j \rangle \in E[i]$ and $X \in V$ such that $f \in F_X$) **do**

- - for each pair that shifts on nonterminal X

for (each pair $\langle p, l \rangle \in E[j]$ and $q \in Q$ s.t. $p \xrightarrow{X} q$) **do**

add pair $\langle q, l \rangle$ to element $E[i]$

end for

end for

while (some pair has been added)

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 non-terminal 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 (each pair $\langle p, j \rangle \in E[i - 1]$ and $q \in Q$ s.t. $p \xrightarrow{x_i} q$) **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 4.76 (Earley syntactic analysis)

- - analyze the terminal string x for possible acceptance
- - define the Earley vector $E[0 \dots n]$ 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  $E[1 \dots n]$ 
   $E[i] := \emptyset$ 
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 ( $i \leq n$   $\wedge$   $E[i - 1] \neq \emptyset$ ) 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 4.30 on p. 183, 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 4.30 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 a 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 4.76 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 4.77 When the Earley algorithm (Algorithm 4.76) 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 we have $f_S \in F_S$.

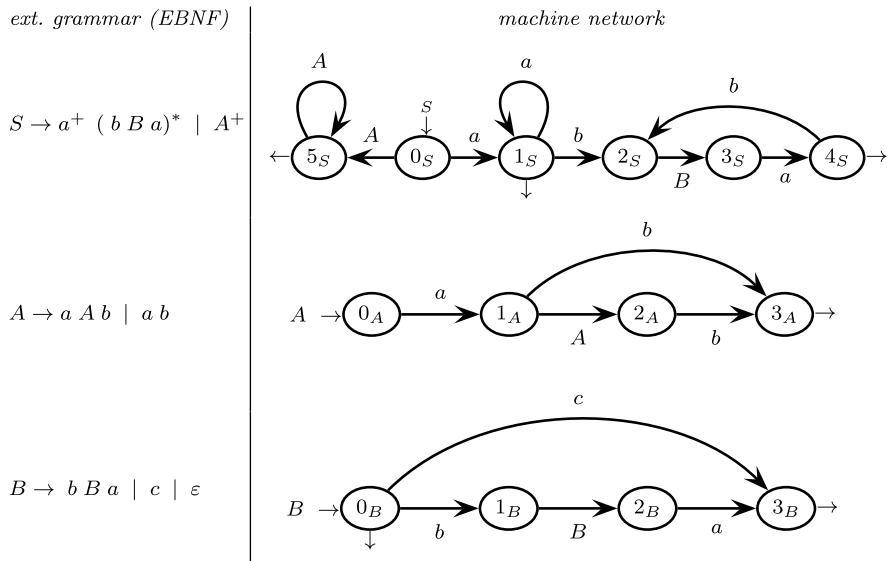


Fig. 4.49 Non-deterministic EBNF grammar G and network of Example 4.78

We have seen a complete run of the algorithm in Table 4.6, but that grammar is too simple to illustrate other interesting points: iterations in the EBNF grammar (circuits in the net), and nullable nonterminals.

Example 4.78 (Non-deterministic EBNF grammar with iterative operators and a nullable nonterminal) Figure 4.49 lists an unambiguous nondeterministic grammar (EBNF) with the corresponding machine net. The string $aabbba$ is analyzed and its Earley vector is shown in Fig. 4.50. 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 4.77.

The arcs represented in Fig. 4.50 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., $\langle 3_B, 3 \rangle \rightarrow \langle 3_S, 0 \rangle$ (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 a complete recognizing path within the machine of the nonterminal. The solid non-terminal shift arc, which has the same origin and destination as the path, cuts short

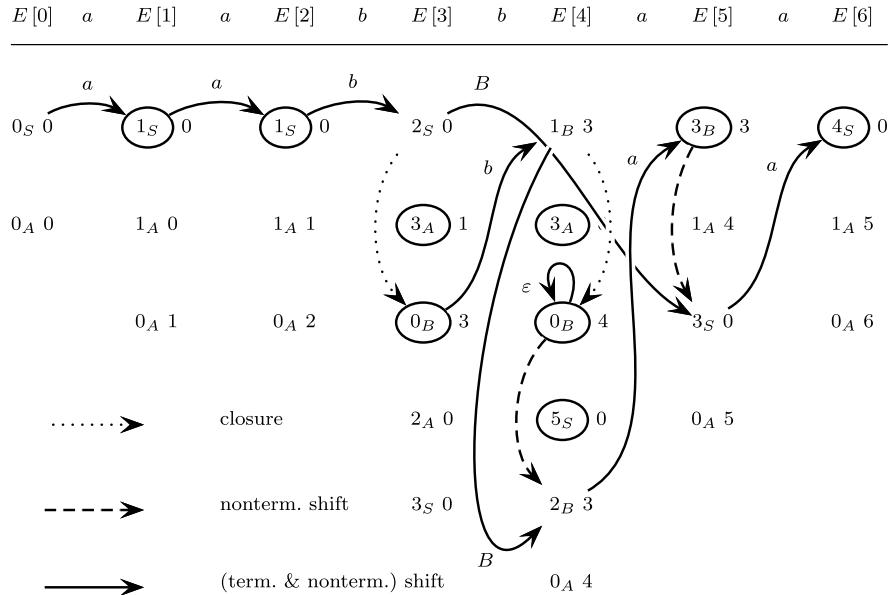


Fig. 4.50 Parse trace of string *aabbbaa* with the machine net in Fig. 4.49

the whole derivation. See for instance:

$$\begin{array}{c}
 \langle 2S, 0 \rangle \xrightarrow[\text{(dotted)}]{\text{closure}} \langle 0B, 3 \rangle \xrightarrow[\text{(sl.)}]{b} \langle 1B, 3 \rangle \xrightarrow[\text{(sl.)}]{B} \langle 2B, 3 \rangle \xrightarrow[\text{(sl.)}]{a} \langle 3B, 3 \rangle \xrightarrow[\text{(dashed)}]{\text{n.t. shift}} \langle 3S, 0 \rangle \\
 \langle 2S, 0 \rangle \xrightarrow[\text{nonterminal shift on } B]{\text{(solid)}} \langle 3S, 0 \rangle
 \end{array}$$

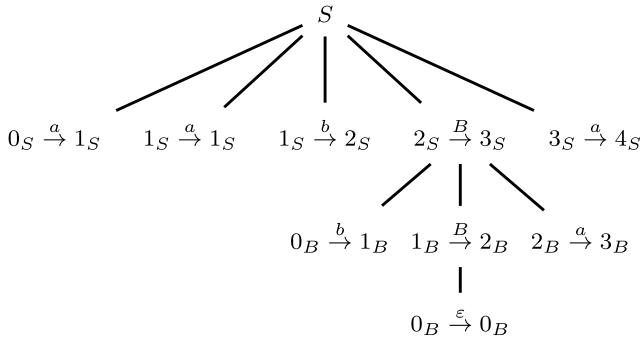
Notice the self-loop on pair $\langle 0_B, 4 \rangle$ with label ε , as state 0_B is both initial and final or equivalently nonterminal B is nullable. It means that here machine M_B starts and immediately finishes the computation. Equivalently it indicates that the corresponding instance of B immediately 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 $E[4]$:

$$\begin{array}{c}
 \langle 1B, 3 \rangle \xrightarrow[\text{(dotted)}]{\text{closure}} \langle 0B, 4 \rangle \xrightarrow[\text{(solid)}]{\varepsilon} \langle 0B, 4 \rangle \xrightarrow[\text{(dashed)}]{\text{n.t. shift}} \langle 2B, 3 \rangle \\
 \langle 1B, 3 \rangle \xrightarrow[\text{nonterminal shift on } B]{\text{(solid)}} \langle 2B, 3 \rangle
 \end{array}$$

In fact if the nonterminal shift arc $\langle 1B, 3 \rangle \xrightarrow{B} \langle 2B, 3 \rangle$ spanned a few vector elements, then nonterminal B would generate the terminals in between, not the empty string.

representation with shift arcs or net transitions



projection over the total alphabet of the grammar

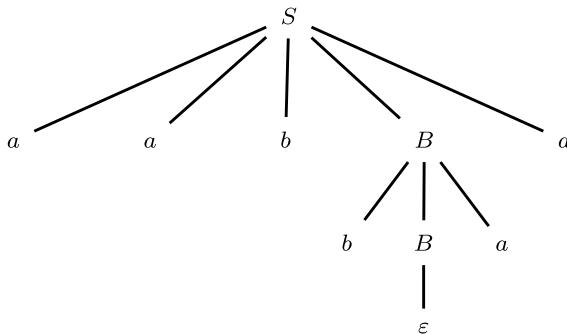


Fig. 4.51 Syntax tree for the string *aabbbaa* of Example 4.78

In practice the complete thread of arcs in Fig. 4.50 represents the syntax tree of the string, which is explicitly shown in Fig. 4.51. The nodes of the top tree are the machine transitions that correspond to the solid shift arcs (terminal and nonterminal), 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 the Earley Algorithm Property 4.79 links certain pairs in the elements of the Earley vector with the existence of a leftmost derivation for the string prefix analyzed so far. Along with the consequential Property 4.80 it states the *correctness* of the method: if a string is accepted by Algorithm 4.76 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 4.79 If we have $\langle q, j \rangle \in E[i]$, 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 we have

$\langle 0_X, j \rangle \in E[j]$ and the right-linearized grammar \hat{G} admits a leftmost derivation $0_X \xrightarrow{*} x_{j+1} \dots x_i q$ if $j < i$ or $0_X \xrightarrow{*} q$ if $j = i$.

For instance: in Example 4.75 we have $\langle 3_A, 1 \rangle \in E[3]$ with $q = 3_A$ and $1 = j < i = 3$, then we have $\langle 0_A, 1 \rangle \in E[1]$ and grammar \hat{G} admits a derivation $0_A \Rightarrow a1_A \Rightarrow ab3_A$; in Example 4.78 we have $\langle 0_B, 4 \rangle \in E[4]$ with $q = 0_B$ and $j = i = 4$, then the property holds because state q itself is initial already.

Property 4.80 If the Earley acceptance condition (Property 4.77) is satisfied, i.e., $\langle f, 0 \rangle \in E[n]$ with $f \in F_S$, then the EBNF grammar G admits a derivation $S \xrightarrow{+} x$, i.e., $x \in L(S)$, and string x belongs to language $L(G)$.

For instance, again in Example 4.75 we have $\langle 1_S, 0 \rangle \in E[4]$ ($n = 4$) with $f = 1_S \in F_S$, then the right-linearized and EBNF grammars \hat{G} and G admit derivations $0_S \xrightarrow{*} aabb1_S$ and $S \xrightarrow{*} aabb$, respectively, hence we have $aabb \in L(S)$ and string $aabb$ 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 . So Algorithm 4.76 is shown to be correct.

Property 4.81 has two points 1 and 2 that are the converse of Properties 4.79 and 4.80, 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 4.76 executed for G .

Property 4.81 Take an EBNF grammar G and a string $x = x_1 \dots x_n$ of length n that belongs to language $L(G)$. In the right-linearized grammar \hat{G} , consider any leftmost derivation d of a prefix $x_1 \dots x_i$ ($i \leq n$) of x , that is:

$$d : 0_S \xrightarrow{+} x_1 \dots x_i q W$$

with $q \in Q_X$ and $W \in Q^*$. The two points below apply:

1. if we have $W \neq \varepsilon$, i.e., $W = rZ$ for some $r \in Q_Y$, then we have $\exists j 0 \leq j \leq i$ and $\exists p \in Q_Y$ such that the machine net has an arc $p \xrightarrow{X} r$ and grammar \hat{G} admits two leftmost derivations $d_1 : 0_S \xrightarrow{+} x_1 \dots x_j p Z$ and $d_2 : 0_X \xrightarrow{+} x_{j+1} \dots x_i q Z$, so that derivation d decomposes as follows:

$$\begin{aligned} d : 0_S &\xrightarrow{d_1} x_1 \dots x_j p Z \xrightarrow{p \rightarrow 0_X r} x_1 \dots x_j 0_X r Z \\ &\xrightarrow{d_2} x_1 \dots x_j x_{j+1} \dots x_i q r Z = x_1 \dots x_i q W \end{aligned}$$

- as an arc $p \xrightarrow{X} r$ 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 we have $W = \varepsilon$, then we have $X = S$, i.e., nonterminal X is the axiom, $q \in Q_S$ and $\langle q, 0 \rangle \in E[i]$

- b. if it also holds $x_1 \dots x_i \in L(G)$, i.e., the prefix also belongs to language $L(G)$, then we have $q \in F_S$, i.e., state q is final for the axiomatic machine M_S , and the prefix is accepted (Property 4.77) by the Earley algorithm

Limit cases: if we have $i = 0$ then we have $x_1 \dots x_i = \varepsilon$; if we have $j = i$ then we have $x_{j+1} \dots x_i = \varepsilon$; and if we have $x = \varepsilon$ (so $n = 0$) then both cases hold, i.e., $j = i = 0$.

We skip over Property 4.81 as entirely evident.

If the prefix coincides with the whole string x , i.e., $i = n$, then step b 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 4.77 is satisfied. For instance in Example 4.78 we have $\langle 5_S, 0 \rangle \in E[4]$ with state 5_S final, so the corresponding prefix *aabb* of length 4 is accepted and it actually belongs to language $L(G)$.

In conclusion, according to Properties 4.79, 4.80, and 4.81, Algorithm 4.76 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 4.76 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 $\langle \text{state}, \text{pointer} \rangle$ 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 we have $j \leq i$. Since we also have $i \leq n$, we conservatively assume that 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:

$$\begin{aligned}\text{Earley cost} &= \text{TerminalShift} \times n + \text{Completion} \times (n + 1) \\ &= \mathcal{O}(n) \times n + \mathcal{O}(n^2) \times (n + 1) = \mathcal{O}(n^3)\end{aligned}$$

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 4.82 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.

4.10.2.1 Remarks on Complexity and Optimizations

In practice the Earley algorithm (Algorithm 4.76) performs faster in some realistic 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 [13]. For the non-extended (BNF) grammars that are deterministic, it approaches a linear time, i.e., $\mathcal{O}(n)$, in most cases [17].

It is worthy hinting how the non-ambiguous case might work. 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 so, then the grammar would have a few different ways to generate a string and thus it would be ambiguous. Then 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., $\langle q, j_1, \dots, j_t \rangle$. Since while the element builds, each pair adds only once to it, a tuple may not have any identical pointers and therefore inequality $t \leq n + 1$ holds. Shifting 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 is kept and makes 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 before it was cubic (Property 4.82).

This justification is independent of the grammar being non-extended, and reasonably holds for non-ambiguous extended (*EBNF*) grammars as well. But remember that the element contents need a suitable rearrangement.

We also mention the issue of the 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 [3]. 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. See also [17] for an overview of a few parsing methods and practical variants.

4.10.3 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 a representation is described in Chap. 2, p. 41).

Take an *EBNF* grammar $G = (V, \Sigma, P, S)$ and machine net \mathcal{M} . Consider a string $x = x_1 \dots x_n$ 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 nonnegative 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 :

$$X \xrightarrow[G]{+} x_{j+1} \dots x_i \quad \text{if } j < i \quad X \xrightarrow[G]{+} \varepsilon \quad \text{if } j = i$$

Grammar G admits derivation $S \xrightarrow{+} x_1 \dots x_n$ or $S \xrightarrow{+} \varepsilon$, 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 $BuildTree(S, f, 0, n)$; then the function will recursively build all the subtrees and will assemble them in the final tree. The commented code follows.

Algorithm 4.83 (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  $\varepsilon$  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  $\left( \exists h = k-1 \ \exists p \in Q_X \text{ such that } \langle p, j \rangle \in E[h] \wedge \text{net has } p \xrightarrow{x_k} q \right)$  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  $\left( \exists Y \in V \ \exists e \in F_Y \ \exists h \leq h \leq k \leq i \ \exists p \in Q_X \text{ s.t. } \langle e, h \rangle \in E[k] \wedge \langle p, j \rangle \in E[h] \wedge \text{net has } p \xrightarrow{Y} q \right)$  then
      - - recursively build the subtree of the derivation:
      - -  $Y \xrightarrow[G]{+} x_{h+1} \dots x_k$  if  $h < k$  or  $Y \xrightarrow[G]{+} \varepsilon$  if  $h = k$ 
      - - and concatenate to list  $\mathcal{C}$  the subtree of node  $Y$ 
       $\mathcal{C} := BuildTree(Y, e, h, k) \cdot \mathcal{C}$ 
  
```

```

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 4.76).

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.

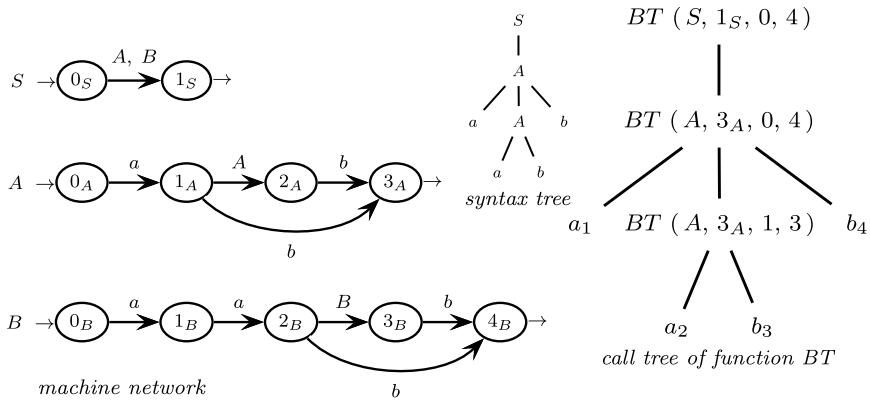
A limit case: if the parent nonterminal X immediately generates the empty string, then leaf ε is the only child, and the loop is skipped from start (see also Fig. 4.53).

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. 4.53). 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 4.83, 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.



Earley vector and how function BT walks it back for constructing the syntax tree above

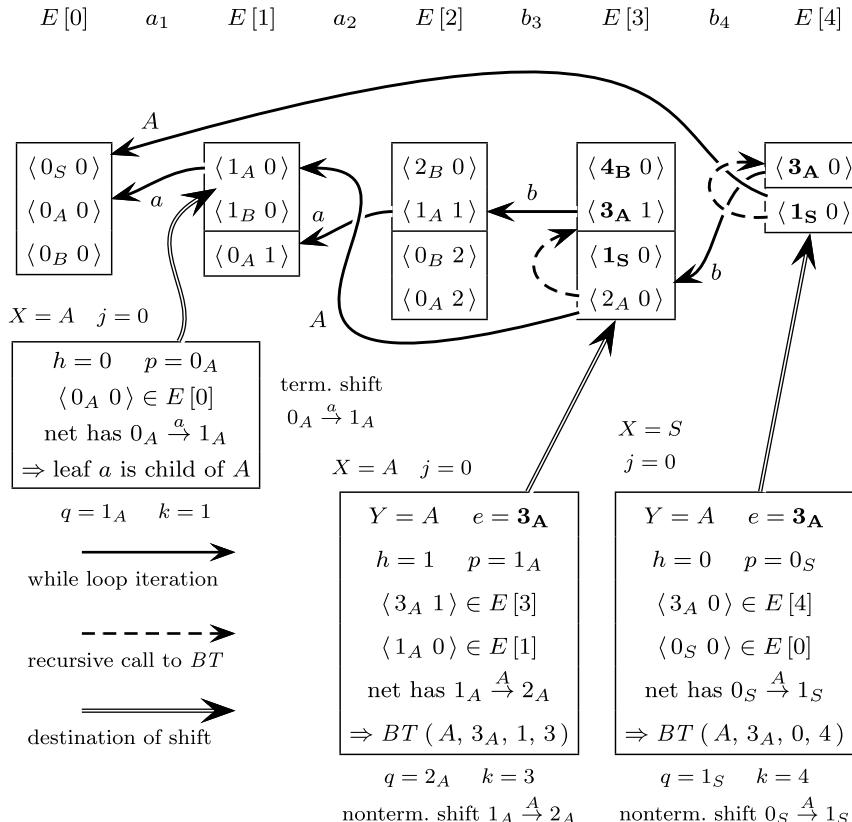


Fig. 4.52 Actions and calls of function *BuildTree* on the shifts for Example 4.75

Example 4.84 (Construction of the syntax tree from the Earley vector) We reproduce the machine net of Example 4.75 in Fig. 4.52, top left. For input string $x_1 \dots x_4 = aabb$ 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 4.76). The Earley vector (the same as in Table 4.6) 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. 4.50, and trace the terminal and nonterminal shifts.

The string is accepted thanks to pair $\langle 1_S, 0 \rangle$ in element $E[4]$ (Property 4.77). The initial call to function *BuildTree* is $BT(S, 1_S, 0, 4)$, which builds the tree for the whole string $x_1 \dots x_4 = aabb$.

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 4.83 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 $x_1 \dots x_4 = aabb$.

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 $x_2 x_3 = ab$.

The two boxes at the bottom of Fig. 4.52 explain the nonterminal shifts (point (b) in Algorithm 4.83), 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. 4.52 and compare it with the if-then statement (a) in Algorithm 4.83. Three more terminal shifts take place, but we do not describe them.

Example 4.85 (Syntax tree with nullable nonterminals) Figure 4.53 shows the call tree of function *BuildTree* of Example 4.78. The corresponding parse trace is in Fig. 4.50, p. 256: 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 4.53 also shows the parenthesized subtrees returned by each call to BT .

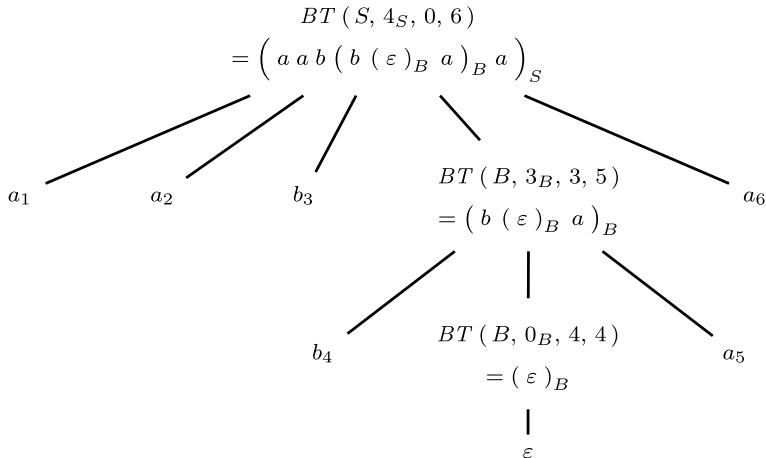


Fig. 4.53 Calls and return values of *BuildTree* for Example 4.78

The innermost call to the function *BuildTree* in Fig. 4.53 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 .

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 4.83 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 (non-terminal) 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:

$$\begin{aligned} BT \text{ cost} &= (\text{if-then (a)} + \text{if-then (b)}) \times \# \text{ of tree nodes} \\ &= (\mathcal{O}(n) + \mathcal{O}(n^2)) \times \mathcal{O}(n) = \mathcal{O}(n^3) \end{aligned}$$

Property 4.86 The asymptotic time complexity of function *BuildTree* to construct a syntax tree for a non-ambiguous grammar (Algorithm 4.83) is cubic in the worst case, i.e., $\mathcal{O}(n^3)$, where n is the length of the string accepted.

The above complexity can be improved if we consider that function *BuildTree* (Algorithm 4.83) just reads the Earley vector E and does not write it. Therefore its 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 pointer j ($0 \leq j \leq n$); 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 becomes

$$\begin{aligned} BT \text{ cost} &= \text{pre-ordering } E + (\text{if-then (a)} + \text{if-then (b)}) \times \# \text{ of tree nodes} \\ &= \mathcal{O}(n^2 \log n) + (\mathcal{O}(\log n) + \mathcal{O}(n \log n)) \times \mathcal{O}(n) = \mathcal{O}(n^2 \log n) \end{aligned}$$

This new complexity of *BuildTree*, for non-ambiguous grammars, is lower than the previous cubic one without pre-ordering (Property 4.86), and approaches that of the Earley algorithm rearranged for non-ambiguous grammars.

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 4.83 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
$\mathcal{C} := \text{BuildTree}(Y, e, h, k) \cdot \mathcal{C}$	Create node Y , call $\text{BuildTree}(Y, \dots)$ 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 some library for the management of graph structures, as many are available. The worst-case time complexity of this variant of *BuildTree* is still cubic.

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 deserve interest in the processing of natural languages. 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 [35] for how to proceed in such more general cases.

4.11 Parallel Local 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.

First and foremost, the deterministic sequential parsing algorithms cannot 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 (hence $LR(1)$) language

$$L = \overbrace{\{c^*oc^*a^n b^n \mid n \geq 1\}}^{L_1} \cup \overbrace{\{c^*tc^*a^n b^{2n} \mid n \geq 1\}}^{L_2}$$

such that the presence of letter o (respectively t) announces that each later letter a will be followed by *one* letter b (respectively *two* b 's). Imagine to parse the following text subdivided into two segments, as shown here:

$\overbrace{cc \dots ct}^{\text{segment 1}} \overbrace{cc \dots caa \dots abb \dots b}^{\text{segment 2}}$

The second parser does not know whether the first one has encountered a letter o or t , and so faces the task of analyzing language $\{a^n b^n \mid n \geq 1\} \cup \{a^n b^{2n} \mid n \geq 1\}$, which is non-deterministic (as explained on p. 156 for a similar case). Notice that an arbitrary distance separates the positions of substrings o (or t) and abb , where the choice of the reduction $ab \rightsquigarrow \dots$ or $abb \rightsquigarrow \dots$ is due. This example shows that, if we want to be able to deterministically parse separate text segments, we must exclude the deterministic languages that exhibit 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 state 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 parsing algorithm that bases its parsing decisions on the local properties of the text, thereby it lends itself to parallel execution.

4.11.1 Floyd's Operator-Precedence Grammars and Parsers

Floyd [12, 14], 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. Floyd's approach assumes that the grammar has been normalized in the *operator form* (see Sect. 2.5.13.7 on p. 62); 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: $S \rightarrow bSeS \mid bSe \mid be$. 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 consider the language of the arithmetic expressions with the operators plus “+” (addition) and times “ \times ” (multiplication). Since traditionally times takes precedence over plus, these two relations hold: $\times \gg +$ and $+ \ll \times$. The former relation means that in an expression such as $5 \times 3 + 7$, the first reduction to apply is $5 \times 3 \rightsquigarrow \dots$. Similarly the fact that plus yields precedence to times (latter relation), says that in an expression such as $5 + 3 \times 7$, the first reduction to apply is $3 \times 7 \rightsquigarrow \dots$.

We are going to extend this traditional sense of precedence from the arithmetic operators to any terminal characters. Moreover we introduce the “equal in precedence” relation \doteq , to mean that the two related characters are adjacent in a grammar rule: thus the left and the right parentheses are related by “ $(\cdot\doteq\cdot)$ ” in the rule $E \rightarrow (F)$ for parenthesized arithmetic expressions.

A word of warning: none of the relations enjoys the mathematical properties of reflexivity, symmetry, and transitivity, of the numerical order relations $<$, $>$ and $=$, notwithstanding the similarity of the signs.

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. To formalize precedences we introduce two sets, 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.

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

$$\text{if } A \xrightarrow{*} \beta \text{ then } \beta \in \Sigma^* (V_N \Sigma^+)^* \cup \Sigma^* (V_N \Sigma^+)^* V_N$$

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

$$a \in \Sigma \quad \text{or} \quad A \in V \quad \text{or} \quad B\sigma C$$

where B, C are nonterminals possibly null, and the string σ in operator form starts and ends with a terminal character.

Definition 4.87 The *left* and *right terminal sets*, denoted \mathcal{L}, \mathcal{R} , of a string in the operator form are recursively defined by

$$\begin{aligned} \mathcal{L}(a) &= \{a\} & \mathcal{R}(a) &= \{a\} \\ \mathcal{L}(A) &= \bigcup_{\forall \text{rule } A \rightarrow \alpha} \mathcal{L}(\alpha) & \mathcal{R}(A) &= \bigcup_{\forall \text{rule } A \rightarrow \alpha} \mathcal{R}(\alpha) \\ \mathcal{L}(A\sigma C) &= \text{Init}(\sigma) \cup \mathcal{L}(A) & \mathcal{R}(A\sigma C) &= \text{Fin}(\sigma) \cup \mathcal{R}(C) \end{aligned} \tag{4.18}$$

where *Init* and *Fin* denote the initial and final character of a string.

Fig. 4.54 Floyd grammar for certain arithmetic expressions and its operator-precedence matrix (read a matrix entry as *row rel. col*, e.g., $n \triangleright +$, etc.)

grammar G			operator precedence matrix M		
$S \rightarrow A \mid B$			n	\triangleright	\triangleright
$A \rightarrow A + B \mid B + B$			$+$	\triangleleft	\triangleright
$B \rightarrow B \times n \mid n$			\times	\doteq	

Table 4.7 Some left and right terminal sets for the grammar in Fig. 4.54

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	$\{n, \times, +\}$	$\{n, +\}$

Example 4.88 (Grammar of arithmetic expressions) A grammar G for certain arithmetic expressions is shown in Fig. 4.54, left.

The left/right terminal sets of the strings that occur in the grammar as left- or right-hand sides, computed by Eq. (4.18), are listed in Table 4.7.

To compute the precedence relations we need the following definition.

Definition 4.89 (Precedence relations) There are three operator-precedence (*OP*) relations over $\Sigma \times \Sigma$:

- takes precedence: $a \triangleright b$ iff some rule contains Db , $D \in V$ and $a \in \mathcal{R}(D)$
- yields precedence: $a \triangleleft b$ iff some rule contains aD , $D \in V$ and $b \in \mathcal{L}(D)$
- equal in precedence: $a \doteq b$ iff some rule contains aBb , $B \in V \cup \epsilon$

The *operator-precedence matrix* (*OPM*) M is a square array of size $|\Sigma| \geq 1$ that associates to each ordered pair (a, b) the set $M_{a,b}$ of *OP* relations holding between terminals a and b .

A grammar has the *operator-precedence property* (we also say it is a Floyd operator-precedence grammar) if, and only if, each matrix entry contains one relation at most. In that case, the matrix qualifies as *conflict-free*.

To illustrate the definitions of the relations, we refer the reader to Fig. 4.55.

As said, the precedence relations are not necessarily symmetric, in the sense that relation $a \triangleright b$ does not imply that relation $b \triangleleft a$ holds; similarly relation $a \doteq b$ does not imply that relation $b \doteq a$ holds.

For the current Example 4.88 the *OPM* in Fig. 4.54, right, is computed by inspecting the rules and the left/right terminal sets listed in Table 4.7.

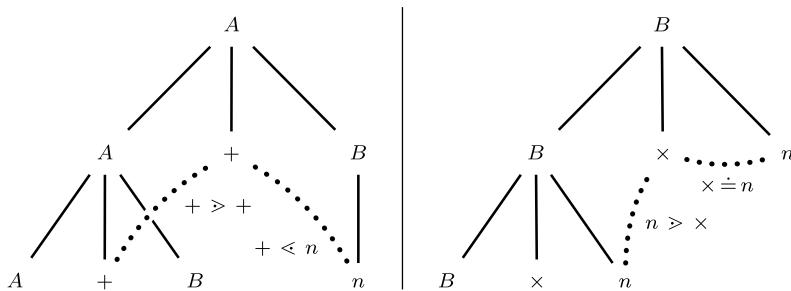


Fig. 4.55 Precedence relations made apparent in the derivations of Example 4.88

Rule	Terminal set	Precedence relations
$\dots \rightarrow A + \dots$	$\mathcal{R}(A) = \{n, +\}$	$n > +$ and $+ > +$
$\dots \rightarrow B + \dots$	$\mathcal{R}(B) = \{n\}$	$n > +$
$\dots \rightarrow \dots + B$	$\mathcal{L}(B) = \{n, \times\}$	$+ \lessdot n$ and $+ \lessdot \times$
$\dots \rightarrow \dots \times n$		$\times \doteq n$

The grammar has the operator-precedence property since its *OPM* is conflict-free.

We comment the meaning of a few relations. The *OPM* entries $+ \lessdot n$ and $n > +$ indicate that, when parsing a string containing the pattern $\dots + n + \dots$, the right part n of rule $B \rightarrow n$ 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 $\dots n An \dots$ or $\dots n Bn \dots$; 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 $\dots \times + \dots$, $\dots \times A + \dots$ and $\dots \times B + \dots$ ever occurs in a sentential form.

The next example refers to a real language, which turns out to have a syntax ideal for precedence parsing.

Example 4.90 (Grammar of JSON) 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 and precedence matrix of *JSON* are shown in Fig. 4.56. The symbol *number* is a numeric constant and the symbol *bool* stands for “true”, “false” or “null”. The symbol *char* denotes any 8 bit character, except those listed in the matrix. 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.

4.11.1.1 Comparison with Deterministic Languages

We present a few examples to compare the languages of operator-precedence grammars and those of *LL(1)* and *LR(1)* grammars.

S → OBJECT

OBJECT → { } | { MEMBERS }

MEMBERS → PAIR | PAIR , MEMBERS

PAIR → STRING : VALUE

VALUE → STRING | OBJECT | ARRAY | *number* | *bool*

STRING → " " | " CHARS "

ARRAY → [] | [ELEMENTS]

ELEMENTS → VALUE | VALUE , ELEMENTS

CHARS → CHAR | CHAR CHARS || CHARS → *char* | *char* CHARS

CHAR → *char*

operator precedence matrix

	{	}	,	:	number	bool	"	char	[]
{	≡	≤	≤				≤			
}		≥	≥							≥
,	≤	≥	≥	≤	≤	≤	≤	≤	≤	≥
:	≤	≥	≥		≤	≤	≤	≤	≤	≤
number		≥	≥							≥
bool		≥	≥							≥
"		≥	≥	≥			≡	≤	≤	≥
char							≥	≥		
[≤		≤		≤	≤	≤	≤	≤	≡
]		≥	≥							≥

Fig. 4.56 Official JSON grammar. *Uppercase* (respectively *lowercase*) words denote nonterminals and terminals; in the *OPM* void cases are left empty. The only rule modified to cast the grammar into operator form is *underlined*

A grammar that is deterministic but not *OP*, and that generates the language $\{a^n b a^n \mid n \geq 0\}$, is

$$S \rightarrow a S a \mid b \quad \text{with matrix} \quad \begin{array}{c|cc}
& a & b \\
\hline a & \leq > \equiv & \leq \\
b & > &
\end{array}$$

There are conflicts in the matrix entry $[a, a]$, yet the grammar meets the *LL(1)* and therefore also the *LR(1)* condition.

The *OP* grammars have 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” relations. To illustrate we list another grammar G :

$$G \left\{ \begin{array}{l} S \rightarrow Xb \\ X \rightarrow aXb \mid ab \end{array} \right. \quad \text{with } OPM : \quad \begin{array}{c} a \quad b \\ \begin{array}{|c|c|} \hline & \leq & \div \\ \leq & \boxed{} & \boxed{} \\ \hline & \geq & \div \\ \geq & \boxed{} & \boxed{} \\ \hline \end{array} \\ b \end{array}$$

The reversed language $(L(G))^R$ is generated by the OP grammar G_R :

$$G_R \left\{ \begin{array}{l} S \rightarrow bX \\ X \rightarrow bXa \mid ba \end{array} \right. \quad \text{with } OPM : \quad \begin{array}{c} a \quad b \\ \begin{array}{|c|c|} \hline & \geq & \\ \geq & \boxed{} & \boxed{} \\ \hline & \div & \leq \\ \div & \boxed{} & \boxed{} \\ \hline \end{array} \\ b \end{array}$$

On the other hand, in general neither a $LL(1)$ nor a $LR(1)$ grammar remains such when is reversed. For instance grammar G has the $LL(1)$ property, but the reversed grammar G_R does not. The case of a language L that ceases to be deterministic upon reversal is illustrated by

$$L = \{oa^n b^n \mid n \geq 1\} \cup \{ta^{2n} b^n \mid n \geq 1\}$$

Here the first character (o or t) determines whether the string contains one a per b or two a 's per b . This property is lost in the reversed language: L^R contains sentences of two types, say, b^4a^4o and b^4a^8t . A deterministic pushdown machine cannot decide when to pop a letter b upon reading an a .

The next statement generalizes the preceding findings.

Property 4.91 (Comparison of OP languages)

- The family of languages defined by Floyd OP grammars is strictly included in the family of deterministic ($LR(1)$) languages.
- Every OP grammar has the property that the mirror reversed grammar is also an OP grammar (so the mirror language remains deterministic).
- The families of languages defined by $LL(1)$ and by OP grammars are not comparable.

In practice OP grammars can be used to define quite a few existing technical languages, if their grammar has been converted into operator form. But there are languages where the use of the same terminal character in different constructs causes precedence conflicts, which may or may not be removed by suitable grammar transformation.

4.11.2 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 left part non-

terminal. This test is very efficient and, unlike the conditions examined by $LR(1)$ parsers, does not rely on long-distance information. However, a minor difficulty remains, if the grammar includes two rules such as $A \rightarrow x$ and $B \rightarrow x$ with a repeated right part: how to choose between nonterminals A and B when string x has been selected for reduction. This uncertainty could be propagated until just one choice remains, but to simplify matters, we assume that the grammar does not have any repeated right parts, i.e., it is *invertible*, a property that can be always satisfied by the transformation presented in Chap. 2, p. 59.

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 can be very efficient and are easy to construct. As the parallel algorithm stems directly from the sequential one, we start from the latter.

The key idea driving the algorithm is simple: wherever a series of \doteq precedence relations enclosed by the relation pair $<, >$ is found between consecutive terminals, the enclosed substring is a reduction handle. To find handles, the parser uses the *OPM* and a pushdown stack S to discover the substring to be reduced, as soon as the next $>$ relation occurs.

It is convenient to assume that the input string is enclosed by the special character $\#$, such that a $\#$ yields precedence to every other character and every character takes precedence over $\#$: this is tantamount to adding the new axiom S_0 and the axiomatic rule $S_0 \rightarrow \#S\#$.

To prepare for its parallel use, we enable the parsing algorithm to analyze also 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 text segments.

The algorithm uses a stack S containing symbols that are couples of type (x, p) , where $x \in \Sigma \cup V$ and p is one of the precedence relations $\{<, \doteq, >\}$ or is undefined (denoted by \perp). The second component is used to encode the precedence relation existing between two consecutive symbols; conventionally it is always $p = \perp$ for nonterminals. We assume that the stack grows rightwards.

Algorithm 4.92 (Operator-precedence sequential parser)²⁸

OPparser($S, u, head, end$)

S : stack of couples u : string to analyze $head, end$: pointers in u

conventions and settings

denote $u = u_1 u_2 \dots u_m$ and $s = u_2 \dots u_{m-1}$ 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 $S = (a, \perp)$

set the initial pointers $head := 2$ and $end := m$

²⁸See [4, 5] for further reading.

```

begin
1. let  $x = u_{\text{head}}$  be the symbol currently pointed by  $\text{head}$  and consider the precedence relation between  $x$  and the top-most terminal  $y$  in the stack  $\mathcal{S}$ 
2. if  $x$  is a nonterminal then
   push  $(x, \perp)$   $\text{head}++$ 
3. else if  $y \lessdot x$  then
   push  $(x, \lessdot)$   $\text{head}++$ 
4. else if  $y \doteq x$  then
   push  $(x, \doteq)$   $\text{head}++$ 
5. else if  $y > x$  then
   a. if stack  $\mathcal{S}$  does not contain  $\lessdot$  then
      push  $(x, \gtrdot)$   $\text{head}++$ 
   b. else
      let the stack  $\mathcal{S}$  be  $(z_0, p_0) \dots (z_{i-1}, p_{i-1})(z_i, \lessdot) \dots (z_n, p_n)$  where  $z_i$  (more precisely  $p_i$ ) is the top-most  $\lessdot$  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  $A \rightarrow z_{i-1}z_i \dots z_n$  then
            replace  $(z_{i-1}, p_{i-1})(z_i, \lessdot) \dots (z_n, p_n)$  in  $\mathcal{S}$  with  $(A, \perp)$ 
         ii. else if  $z_{i-1}$  is a terminal or  $\#$  and  $\exists$  rule  $A \rightarrow z_i \dots z_n$  then
            replace  $(z_i, \lessdot) \dots (z_n, p_n)$  in  $\mathcal{S}$  with  $(A, \perp)$ 
         iii. else
            start an error recovery procedure29
         end if
      end if
   end if
6. if  $\text{head} < \text{end}$  or ( $\text{head} = \text{end}$  and  $\mathcal{S} \neq (a, \perp)(B, \perp)$ ) for any nonterminal  $B$  then goto step (1) else return  $\mathcal{S}$  end if
end

```

Upon the normal termination of the algorithm, its configuration can be no longer changed either by shift or by reduce actions, and we say 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 $\mathcal{S} = (\#, \perp)$ (so $a = \#$), never performs step (2), never pushes (x, \gtrdot) onto the stack, and accepts the input string only if the stack is $\mathcal{S} = (\#, \perp)(S, \perp)$.

We comment a few steps of the parsing trace shown in Fig. 4.57. Step 1 shifts token n on the stack, then step 2 detects the pattern $\langle n \rangle$ and reduces the handle n to B . At step 3 the top-most 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 $B \times n$

²⁹An error is detected either when the handle, i.e., the portion of string included between relations \lessdot and \gtrdot , does not match any r.h.s., or when no precedence relation holds between two consecutive terminal characters.

stack	input string	relation or reduction	step
(#, ⊥)	# ^{head} n + n × n ^{end} #	# ≪ n	1
(#, ⊥)(n, ≪)	+ ^{head} n × n #	n > + n ↠ B	2
(#, ⊥)(B, ⊥)	+ ^{head} n × n #	# ≪ +	3
(#, ⊥)(B, ⊥)(+, ≪)	^{head} n × n #	+ ≪ n	4
(#, ⊥)(B, ⊥)(+, ≪)(n, ≪)	^{head} × n #	n > × n ↠ B	5
(#, ⊥)(B, ⊥)(+, ≪)(B, ⊥)	^{head} × n #	+ ≪ ×	6
(#, ⊥)(B, ⊥)(+, ≪)(B, ⊥)(×, ≈)	^{head} n #	× ≈ n	7
(#, ⊥)(B, ⊥)(+, ≪)(B, ⊥)(×, ≈)(n, ≈)	^{head} #	n > # B × n ↠ B	8
(#, ⊥)(B, ⊥)(+, ≪)(B, ⊥)	^{head} #	+ > # B + B ↠ A	9
(#, ⊥)(A, ⊥)	^{head} end #		10

Fig. 4.57 Steps of a Floyd sequential parser for sentence $a + a \times a$ (grammar and OPM in Fig. 4.54)

in the pattern $\triangleleft B \times \dot{n} \triangleright$; notice that the handle includes a nonterminal B , which occurs between the terminal characters related by $+ \triangleleft \times$.

4.11.3 Parallel Parsing Algorithm

At the level of abstraction that is usually adopted in parsing, 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 over 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 before parallel parsing starts.

Thus 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 spe-

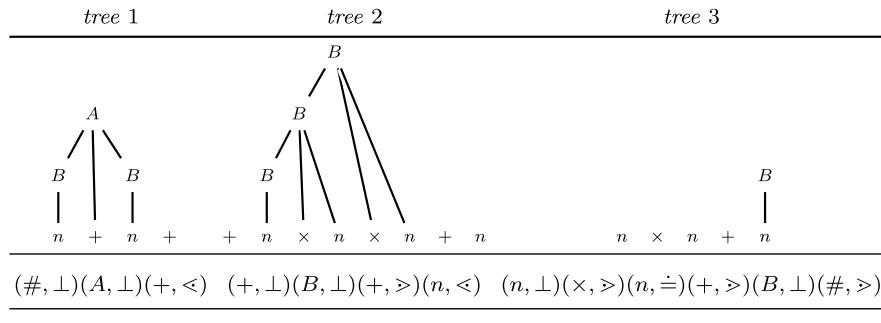


Fig. 4.58 Partial trees and corresponding stacks after the first parsing phase of text $n + n + n \times n \times n + n \times n + n$ by using three workers

cific positions where a token (such as `begin`) occurs, that parts a large syntactic constituent of the language.

Then we apply Algorithm 4.92 to each substring, 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 set of reductions operated by each parser makes 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 overlap by one token any two consecutive segments, i.e., the last terminal of a segment coincides with the first terminal of the next segment.

Example 4.93 (Parallel parsing) Reconsider the grammar of Example 4.88, p. 271, and the source text

$\#n + n + n \times n \times n + n \times n + n\#$

Assume that there are $k = 3$ workers and the segmented text is as

$$\overbrace{n + n + n}^1 \times \overbrace{n \times n + n}^2 \times \overbrace{n + n}^3$$

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 existing k 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 a few 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 reuses the sequential algorithm. 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, and thus causes a 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 4.92 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 $\mathcal{S}_2^L = (+, \perp)(B, \perp)(+, \gtrdot)$ 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 4.92 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 $\mathcal{S}'^L\mathcal{S}'^R$. 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:

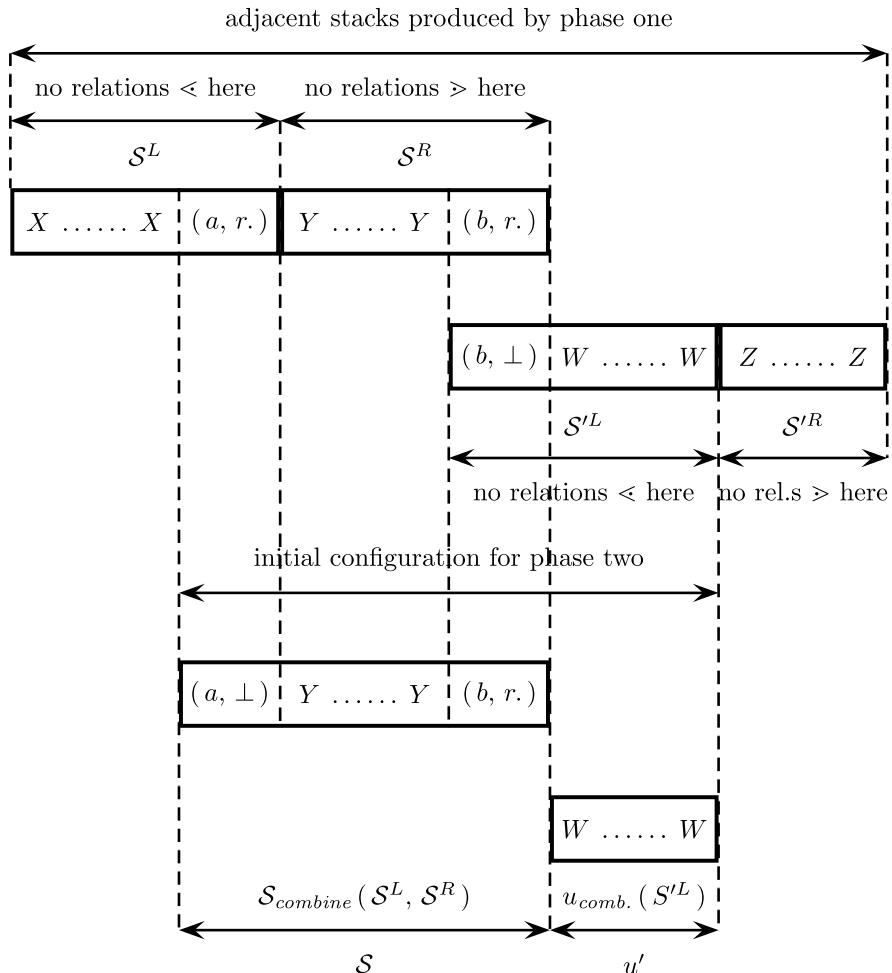


Fig. 4.59 Scheme of the initial configuration of a worker of the next phase: the stack is \mathcal{S} and the input string is u'

1. Stack initialization:

$$\mathcal{S}_{\text{combine}}(\mathcal{S}^L, \mathcal{S}^R) = (a, \perp)\mathcal{S}^R \quad \text{where } a \text{ is the top symbol of } \mathcal{S}^L$$

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 $u_{\text{combine}}(\mathcal{S}'^L)$ 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 $\dot{\equiv}$ precedence relations, the number of stacks can be reduced. Since in the neighboring stack, part \mathcal{S}^R too contains only \lessdot and $\dot{\equiv}$ relations, 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 $\mathcal{S} = (a, \perp)(A, \perp)(b, \succ)$ or $\mathcal{S} = (a, \perp)(A, \perp)(b, \dot{\equiv})$ 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 4.94 (Operator-precedence parallel parser)

1. **split** the input string u into k substrings: $\#u_1 u_2 \dots u_k \#$
2. **launch** k instances of Algorithm 4.92 where for each index i with $1 \leq i \leq k$ the parameters of the i th instance are as

denote as a the last symbol of u_{i-1}

denote as b the first symbol of u_{i+1}

denote $u_0 = u_{k+1} = \#$

$\mathcal{S} := (a, \perp)$

$u := u_i b$

$head := |u_1 u_2 \dots u_{i-1}|$

$end := |u_1 u_2 \dots u_i| + 1$

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 4.92 with these parameters:

$\mathcal{S} := \mathcal{S}_{\text{combine}}(\mathcal{S}_i^L, \mathcal{S}_i^R)$

$s := u_{\text{combine}}(\mathcal{S}_{i+1}^L)$

$head := 1$

$end := |u|$

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 4.95 (Parallel parsing steps) We resume from the configuration shown in Fig. 4.58, where three bipartite stacks, computed by three workers, are present.

$(\#, \perp) (A, \perp) (+, \lessdot)$	\mathcal{S}_1
$(+, \perp) (B, \perp) (+, \gg) \overbrace{(n, \lessdot)}^{\mathcal{S}_2^R}$	\mathcal{S}_2
$(n, \perp) (\times, \gg) (n, \doteq) (+, \gg) (B, \perp) (\#, \gg)$	$\mathcal{S}_3 = \mathcal{S}_3^L$

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, **repeat**, launches Algorithm 4.92 on every initial configuration constructed as shown in Fig. 4.59. Here there is just one instance, namely the parser that combines the outcome of workers 2 and 3, starting from the initial configuration:

$$\overbrace{(+, \perp)(n, \lessdot)}^{\mathcal{S}_{\text{combine}}(\mathcal{S}_2^L, \mathcal{S}_2^R)} \quad \overbrace{\times n + B\#}^{u_{\text{combine}}(\mathcal{S}_3^L)}$$

In the order, the following reductions are performed:

$$n \rightsquigarrow B \quad B \times n \rightsquigarrow B$$

thus obtaining the stack: $\mathcal{S} = (+, \perp)(B, \perp)(+, \gg)(B, \perp)(\#, \gg)$. The situation before the final round is

$(\#, \perp) (A, \perp) (+, \lessdot) (B, \perp) (+, \gg)$	$\mathcal{S}_1 \mathcal{S}_2^L$
$(+, \perp) (B, \perp) (+, \gg) (B, \perp) (\#, \gg)$	\mathcal{S}

The parser can use as input string simply a # and the stack obtained by concatenating the existing stacks:

$$(\#, \perp)(A, \perp)(+, \lessdot)(B, \perp)(+, \gg)(B, \perp)(+, \gg)(B, \perp)$$

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.

4.11.3.1 Complexity

Parallel parsing techniques are mainly motivated by possible speed gains 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 speed-up 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)(a) of Algorithm 4.94, takes $\mathcal{O}(1)$ time, hence $\mathcal{O}(k)$ overall time for k workers. This goal is easily achieved by maintaining at runtime 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 \triangleright 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 [4] indicate that parsing large texts in parallel on the current multi-processor and multi-core computers, achieves a significant speed-up.

4.12 Managing Syntactic Errors and Changes

In this section we address two additional features that are sometimes requested from a parser: the capability to manage user errors in a text, and the ability to efficiently re-parse 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 hand-written 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 runtime in rather unpredictable ways, yet it cannot be caught 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 runtime, such as memory errors (segmentation, overflow, etc.) or nontermination. In such cases we say the program violates the *dynamic* (or runtime) *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 runtime 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 [17].

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 push-down 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, then to be called *erroneous*. The current token, for which the move is undefined, is named the *error token*. From a subjective point of view, the human 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 that terminates 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 4.96 (Human error and error token) In the *FORTRAN* language a loop such as DO110J = 9, 20 is defined by the following rule:

$$\langle \text{iterative instruction} \rangle \rightarrow \text{DO} \langle \text{label} \rangle '=' \langle \text{initial value} \rangle ',' \langle \text{step} \rangle$$

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 110J = 9, 20.
delay

A more abstract example from the regular language

$$L = db^* aa^* \cup eb^* cc^*$$

happens with the illegal string $x = dbb\dots b\overset{\downarrow}{cccccc}$, where the first error token is marked by the arrow, whereas it is likely that the human error was to type d for e , because for the two ways to correct x into $x' = e \underbrace{bb\dots b}_{\text{delay}} ccccc$ or $x'' = dbb\dots baaaaaa$, 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; *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.

A Parser with Basic Error Recovery In Fig. 4.60 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 n”. 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 to A_n . For the current instruction two cases are possible: the instruction scans a token e ; or the instruction invokes a ma-

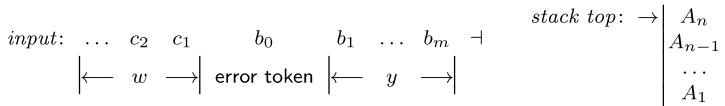


Fig. 4.60 PDA parser configuration when an error is detected. The input string is $z = wb_0y \dashv$, character b_0 is the error token, string w is the already scanned prefix, and string $y = b_1 \dots b_m$ is the suffix still to be scanned. The pushdown stack contains $\alpha = A_1 A_2 \dots A_n$, where A_n is the top symbol

chine/nonterminal E with guide set $Gui(E)$. Then the message says: “at line n 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, \dots, \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 4.97 (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 $call\ i(i + i, i + i)$. We illustrate the panic mode error recovery for the following (erroneous) text:

$$\overbrace{call\ i}^{\text{scanned text}} \ \overbrace{=}^{\text{error}} \overbrace{i + i \dots + i \dashv}^{\text{panic-skipped text}} \quad (4.19)$$

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 $i = i + i \dots + i$; and substituting token ‘ $=$ ’ with ‘(’ and then inserting token ‘)’ at the end before ‘ \dashv ’, yields the invocation *call i(i + i … + 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 (4.19), it suffices to let the parser *insert* tokens, according to its grammatical expectations. In fact, starting again from text (4.19), 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:

$$\text{call } i \stackrel{\text{inserted}}{(} = i + i \dots + i \dashv$$

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:

$$\text{call } i (\stackrel{\text{error}}{=} i + i \dots + i \dashv$$

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 $i + \dots + i$ since this piece fits as actual parameter for the procedure, without any further errors as far as the end-of-text ‘ \dashv ’, which becomes the new error token:

$$\text{call } i (\stackrel{\text{skipped}}{=} \stackrel{\text{(deleted)}}{=} i + i \dots + i \stackrel{\text{error}}{\dashv}$$

The last action is to insert a right parenthesis ‘)’ before the end-of-text ‘ \dashv ’, as expected (since now the syntactic class is that of a parameter list):

$$\text{call } i (i + i \dots + i) \stackrel{\text{inserted}}{\dashv}$$

Then the analysis concludes correctly and recovers a procedure call that has an expression as its actual parameter: *call i(i + i … + 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 [10].

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 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.³⁰

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.³¹ We notice that the local parsability property of Floyd's Operator-Precedence grammars (exploited in Sect. 4.11.1 for parallel parsing), makes incremental parsing too straightforward.

³⁰For this problem see [19].

³¹The main ideas on incremental parsing can be found in [16, 27].

References

1. A. Aho, M. Lam, R. Sethi, J. Ullman, *Compilers: Principles, Techniques and Tools* (Prentice-Hall, Englewood Cliffs, 2006)
2. J. Aycock, A. Borsotti, Early action in an Earley parser. *Acta Inform.* **46**(8), 549–559 (2009)
3. J. Aycock, R. Horspool, Practical Earley parsing. *Comput. J.* **45**(6), 620–630 (2002)
4. A. Barenghi, E. Viviani, S. Crespi Reghizzi, D. Mandrioli, M. Pradella, PAPAGENO: a parallel parser generator for operator precedence grammars, in *SLE*, ed. by K. Czarnecki, G. Hedin. Lecture Notes in Computer Science, vol. 7745 (Springer, Berlin, 2012), pp. 264–274
5. A. Barenghi, S. Crespi Reghizzi, D. Mandrioli, M. Pradella, Parallel parsing of operator precedence grammars. *Inf. Process. Lett.* **113**(7), 245–249 (2013)
6. J.C. Beatty, Two iteration theorems for the $LL(k)$ languages. *Theor. Comput. Sci.* **12**, 193–228 (1980)
7. J.C. Beatty, On the relationship between the $LL(1)$ and $LR(1)$ grammars. *J. ACM* **29**(4), 1007–1022 (1982)
8. J. Bersel, L. Boasson, Formal properties of XML grammars and languages. *Acta Inform.* **38**, 115–125 (2002)
9. L. Breveglieri, S. Crespi Reghizzi, A. Morzenti, Parsing methods streamlined. CoRR, [arXiv:1309.7584](https://arxiv.org/abs/1309.7584) [cs.FL] (2013)
10. M.G. Burke, G.A. Fisher, A practical method for LR and LL syntactic error diagnosis. *ACM Trans. Program. Lang. Syst.* **9**(2), 164–197 (1987)
11. N.P. Chapman, *LR Parsing: Theory and Practice* (Cambridge University Press, Cambridge, 1987)
12. S. Crespi Reghizzi, D. Mandrioli, D.F. Martin, Algebraic properties of operator precedence languages. *Inf. Control* **37**(2), 115–133 (1978)
13. J. Earley, An efficient context-free parsing algorithm. *Commun. ACM* **13**(2), 94–102 (1970)
14. R.W. Floyd, Syntactic analysis and operator precedence. *J. ACM* **10**(3), 316–333 (1963)
15. R. Floyd, R. Beigel, *The Language of Machines: An Introduction to Computability and Formal Languages* (Computer Science Press, New York, 1994)
16. C. Ghezzi, D. Mandrioli, Incremental parsing. *ACM Trans. Program. Lang. Syst.* **1**(1), 58–70 (1979)
17. D. Grune, C. Jacobs, *Parsing Techniques: A Practical Guide*, 2nd edn. (Springer, London, 2009)
18. M. Harrison, *Introduction to Formal Language Theory* (Addison Wesley, Reading, 1978)
19. J. Heering, P. Klint, J. Rekers, Incremental generation of parsers. *IEEE Trans. Softw. Eng.* **16**(12), 1344–1351 (1990)
20. S. Heilbrunner, On the definition of $ELR(k)$ and $ELL(k)$ grammars. *Acta Inform.* **11**, 169–176 (1979)
21. S. Heilbrunner, A direct complement construction for $LR(1)$ grammars. *Acta Inform.* **33**(8), 781–797 (1996)
22. K. Hemerik, Towards a taxonomy for ECFG and RRG parsing, in *Language and Automata Theory and Applications, Third Int. Conf., LATA 2009*, ed. by A.H. Dediu, A.-M. Ionescu, C. Martín-Vide. Lecture Notes in Computer Science, vol. 5457 (Springer, Berlin, 2009), pp. 410–421
23. J. Hopcroft, J. Ullman, *Formal Languages and Their Relation to Automata* (Addison-Wesley, Reading, 1969)
24. J. Hopcroft, J. Ullman, *Introduction to Automata Theory, Languages, and Computation* (Addison-Wesley, Reading, 1979)
25. D.E. Knuth, On the translation of languages from left to right. *Inf. Control* **8**, 607–639 (1965)
26. D.E. Knuth, Top-down syntax analysis. *Acta Inform.* **1**, 79–110 (1971)
27. J. Larchevêque, Optimal incremental parsing. *ACM Trans. Program. Lang. Syst.* **17**(1), 1–15 (1995)

28. J. Lewi, K. De Vlaminck, J. Huens, M. Huybrechts, *A Programming Methodology in Compiler Construction, I and II* (North-Holland, Amsterdam, 1979)
29. D. Mandrioli, C. Ghezzi, *Theoretical Foundations of Computer Science* (Wiley, New York, 1987)
30. R. Quong, T. Parr, ANTLR: a predicated- $LL(k)$ parser generator. *Softw. Pract. Exp.* **25**, 789–810 (1995)
31. G. Révész, *Introduction to Formal Languages* (Dover, New York, 1991)
32. S. Rodger, T.W. Finley, *JFLAP: An Interactive Formal Language and Automata Package* (Jones and Bartlett, Sudbury, 2006)
33. D.J. Rosenkrantz, R.E. Stearns, Properties of deterministic top-down parsing. *Inf. Control* **17**(3), 226–256 (1970)
34. A. Salomaa, *Formal Languages* (Academic Press, New York, 1973)
35. E. Scott, SPPF-style parsing from Earley recognizers. *Electron. Notes Theor. Comput. Sci.* **203**(2), 53–67 (2008)
36. G. Senizergues, $L(A) = L(B)$? A simplified decidability proof. *Theor. Comput. Sci.* **281**, 555–608 (2002)
37. D. Simovici, R. Tenney, *Theory of Formal Languages with Applications* (World Scientific, Singapore, 1999)
38. S. Sippu, E. Soisalon-Soininen, *Parsing Theory, vol. 1: Languages and Parsing* (Springer, Berlin, 1988)
39. S. Sippu, E. Soisalon-Soininen, *Parsing Theory, vol. 2: LR(k) and LL(k)* (Springer, Berlin, 1990)
40. M. Tomita, *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems* (Kluwer Academic, Boston, 1986)
41. N. Wirth, *Algorithms + Data Structures = Programs* (Prentice-Hall, Englewood Cliffs, 1975)

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 termed (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 typical compilers. What is lacking 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 method for designing well-structured translators by taking advantage of the syntactic modularity of grammar rules.

We are going 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 or the message to be communicated. In linguistics the two terms have often been taken as emblems representing forms and contents, respectively, but this reference to the studies in the human sciences does not make the distinction any more precise or formal.

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 used 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 a priori limited: 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 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 and 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 a linear time. This and similar operations are performed by a compiler subsystem usually referred to as a semantic analyzer.

Going deeper into 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 the one 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 the practice a Turing machine is too complicated to be programmed for any realistic problem, let compilation alone.

Years of attempts at inventing grammars or automata that would allow some of the usual 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. 86). In other words, practical syntactic methods are limited to the context-free domain.

5.1.1 Chapter Outline

The word *translation* signifies a correspondence between two texts that have the same meaning, but that are written in different languages. Many cases of translation 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 termed the *source*, and the other text and language are the *target*.

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 the parsing algorithms; and the finite transducer, which is a finite automaton enriched with an output function.

The purely syntactic translation models fall short of the requirements of compilation, as various typical transformations to be performed cannot be expressed with such methods. Nevertheless, the syntactic translation models are important as a conceptual foundation of 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 those 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 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 the 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 are 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

¹A rigorous presentation can be found in Berstel [5] and in Sakarovitch [12].

Σ and Δ , respectively. A translation is a correspondence between source and target strings, to be formalized as a binary relation between the universal languages Σ^* and Δ^* , that is, as a subset of the cartesian product $\Sigma^* \times \Delta^*$ between the source and target universal languages.

A *translation relation* ρ is a set of pairs of strings (x, y) , with $x \in \Sigma^*$ and $y \in \Delta^*$, as follows:

$$\rho = \{(x, y), \dots\} \subseteq \Sigma^* \times \Delta^*$$

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:

$$\begin{aligned} L_1 &= \{x \in \Sigma^* \mid \text{for some string } y \text{ such that } (x, y) \in \rho\} \\ L_2 &= \{y \in \Delta^* \mid \text{for some string } x \text{ such that } (x, y) \in \rho\} \end{aligned}$$

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 :

$$\tau: \Sigma^* \rightarrow \wp(\Delta^*) \quad \tau(x) = \{y \in \Delta^* \mid (x, y) \in \rho\}$$

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:

$$L_2 = \tau(L_1) = \bigcup_{x \in \Sigma^*} \tau(x)$$

In general a translation function is not total, i.e., it is partial: for some strings over the source alphabet the function may be undefined. A simple expedient for making it 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 most important case, occurs when every source string has no more than one image, and in this case the translation function is $\tau: \Sigma^* \rightarrow \Delta^*$.

The *inverse translation* τ^{-1} is a function that maps a target string on the set of the corresponding source strings.³

$$\tau^{-1}: \Delta^* \rightarrow \wp(\Sigma^*) \quad \tau^{-1}(y) = \{x \in \Sigma^* \mid y \in \tau(x)\}$$

²Remember that given a set X , symbol $\wp(X)$ denotes the power set of X , i.e., the set of all the subsets of X .

³Given a translation relation ρ , the inverse translation relation ρ^{-1} is obtained by swapping 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} .

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 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 a 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, 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 naive way to specify a text transformation 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, p. 79. Each source character is transliterated to a target character or more generally to a string.

Let us read Example 2.87 on p. 80 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:

$$h^{-1}(\square) = \{\alpha, \dots, \omega\}$$

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 a letter in the position i of the Latin alphabetical ordering with 26 letters ($i = 0, \dots, 25$), by the letter in the position $(i + k) \bmod 26$ at alphabetical distance k , where constant k is the ciphering secret key ($1 \leq k \leq 25$).

To finish we stress that transliteration transforms a letter into another one and totally ignores 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 component. The latter are then assembled into the target syntax tree that represents the translation. Such a structural translation is now formalized as a mapping scheme, by relating the source and target grammar rules.

Definition 5.1 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 $h_\Sigma: C \rightarrow \Sigma$ and $h_\Delta: C \rightarrow \Delta$ 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 as of Definition 5.1.

Example 5.2 (Translation grammar for string reversal) Take for instance string aab . Its translation is the mirror string baa . The translation grammar G_τ for string reversal is as follows:

$$G_\tau : S \rightarrow \frac{a}{\varepsilon} S \frac{\varepsilon}{a} \mid \frac{b}{\varepsilon} S \frac{\varepsilon}{b} \mid \frac{\varepsilon}{\varepsilon}$$

Equivalently, the translation grammar G_τ can be replaced by the following translation scheme (G_1, G_2):

<i>source grammar</i> G_1	<i>target grammar</i> G_2
$S \rightarrow aS$	$S \rightarrow Sa$
$S \rightarrow bS$	$S \rightarrow Sb$
$S \rightarrow \varepsilon$	$S \rightarrow \varepsilon$

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:

$$h_\Sigma \left(S \rightarrow \frac{b}{\varepsilon} S \frac{\varepsilon}{b} \right) = S \rightarrow bS \quad h_\Delta \left(S \rightarrow \frac{b}{\varepsilon} S \frac{\varepsilon}{b} \right) = S \rightarrow Sb$$

To obtain a pair of strings that correspond in the translation relation, we construct a derivation, like

$$S \Rightarrow \frac{a}{\varepsilon} S \frac{\varepsilon}{a} \Rightarrow \frac{a}{\varepsilon} \frac{a}{\varepsilon} S \frac{\varepsilon}{a} \frac{\varepsilon}{a} \Rightarrow \frac{a}{\varepsilon} \frac{a}{\varepsilon} \frac{b}{\varepsilon} S \frac{\varepsilon}{b} \frac{\varepsilon}{a} \frac{\varepsilon}{a} \Rightarrow \frac{a}{\varepsilon} \frac{a}{\varepsilon} \frac{b}{\varepsilon} \frac{\varepsilon}{b} \frac{\varepsilon}{a} \frac{\varepsilon}{a} = z$$

and then we project the sentence z of $L(G_\tau)$ on the two alphabets, this way:

$$h_\Sigma(z) = aab \quad h_\Delta(z) = baa$$

Otherwise, using the translation scheme, we generate a source string by a derivation of G_1 , and its image by a derivation of G_2 ; and we pay attention to use corresponding rules at each step.

⁴Another historical name for such translations is *simple syntax-directed translations*.

The reader may have noticed that the preceding translation grammar G_τ of Example 5.2 is almost identical to the 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 :

$$G_p: S \rightarrow aSa' \mid bSb' \mid \varepsilon$$

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 exist an alphabet Ω , a context-free language L over Ω , and two alphabetic homomorphisms (transliterations) $h_1: \Omega \rightarrow \Sigma \cup \{\varepsilon\}$ and $h_2: \Omega \rightarrow \Delta \cup \{\varepsilon\}$, such that

$$\rho_{G_\tau} = \{(h_1(z), h_2(z)) \mid z \in L\}$$

Example 5.4 (String reversal continued from Example 5.2) We illustrate with the translation of string $x \in (a \mid b)^*$ to its mirror. From the condition (2) of Property 5.3, the translation ρ_{G_τ} can be expressed by using the alphabet $\Omega = \{a, b, a', b'\}$, and the following context-free language L , quite similar to the palindromes:

$$L = \{u(u^R)' \mid u \in (a \mid b)^*\} = \{\varepsilon, aa', \dots, abbb'b'a', \dots\}$$

where string $(v)'$ is the primed copy of string v . 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 $abb'a' \in L$ is transliterated to the two strings

$$(h_1(abb'a'), h_2(abb'a')) = (ab, ba)$$

that belong to the translation relation ρ_{G_τ} .

5.4.1 Infix and Polish Notations

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.

The *degree of an operator* is the number of arguments or operands the operator may have. The degree can be fixed or variable, and in the latter case the operator is called *variadic*. Having a degree two, i.e., using *binary* operators, is the most common case. For instance the comparison operators such as equality ‘=’ and difference ‘≠’ are binary. 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 noncommutative binary operation, whereas a change of sign (as in $-x$) is an example of *unary* operation. If the same sign ‘-’ is used to denote both operations, the operator becomes variadic with degree one or two.

Examining now the relative positions of signs 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 notion 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:

$$o_0 \ arg_1 \ o_1 \ arg_2 \dots o_{n-1} \ arg_n \ o_n$$

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 of many programming languages is mixfix with degree two, or three if the *else* clause is present:⁵

$$\text{if } arg_1 \text{ then } arg_2 \text{ [else } arg_3\text{]}$$

Because of the varying degree, this representation is ambiguous if the second argument can be in turn a conditional operator (as seen on p. 52). To remove ambiguity, 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

$$\text{jump_if_false } arg_1, arg_2$$

Considering 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*.

⁵Some languages may have a closing mark *end_if* as well.

A representation is called *polish*⁶ if it does not use parentheses, and if the operators are either all prefix or all postfix. The elementary grammar of polish expressions is printed on p. 48.

The next example (Example 5.5) shows a frequent transformation performed by compilers to eliminate parentheses, through converting an arithmetic expression from infix to polish notation. For simplicity we prefer to use disjoint source/target alphabets, and thus to avoid the need of fractions in the rules.

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 (or a number). 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:

$$\Sigma = \{+, \times, '(', ')', i\} \quad \Delta = \{\text{add}, \text{mult}, i'\}$$

Translation grammar (axiom E):

$$G_\tau \quad \begin{cases} E \rightarrow \text{add } T + E \mid T \\ T \rightarrow \text{mult } F \times T \mid F \\ F \rightarrow '(' E ')' \mid ii' \end{cases}$$

Notice that rule $E \rightarrow \text{add } T + E$ abridges rule $E \rightarrow \frac{\varepsilon}{\text{add}} T \frac{+}{\varepsilon} E$, with no danger of confusion because the source/target alphabets are disjoint.

The equivalent translation scheme is the following:

<i>source grammar G_1</i>	<i>target grammar G_2</i>
$E \rightarrow T + E$	$E \rightarrow \text{add } TE$
$E \rightarrow T$	$E \rightarrow T$
$T \rightarrow F \times T$	$T \rightarrow \text{mult } FT$
$T \rightarrow F$	$T \rightarrow F$
$F \rightarrow '(' E ')' \mid ii'$	$F \rightarrow E$
$F \rightarrow i$	$F \rightarrow i'$

An example of translation is the 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 $\text{mult add } i'i'i'$.

Construction of Target Syntax Tree In a translation scheme the rules of the two grammars are in one-to-one correspondence. We observe that corresponding rules

⁶From the nationality of the logician Jan Lukasiewicz, who proposed its use for compacting and normalizing logical formulas.

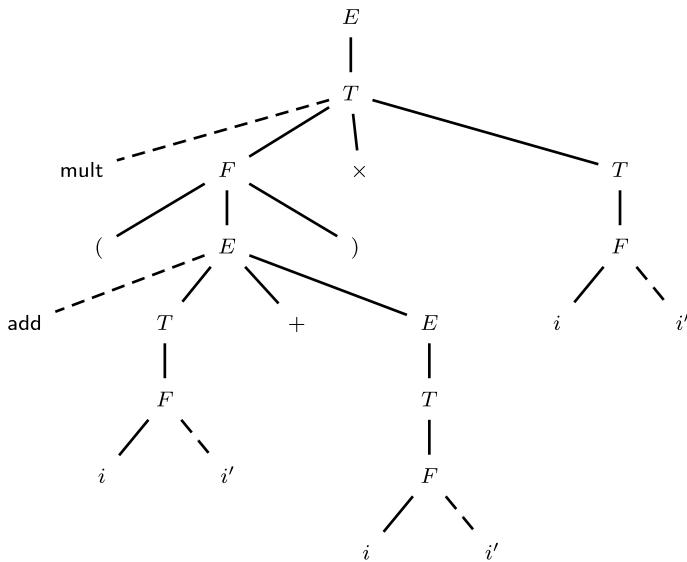


Fig. 5.1 Source-target tree generated by the translation grammar of Example 5.5

have identical left parts and that the nonterminal symbols occur in the same order in their right parts.

Given a translation grammar, we do the following to compute the image of a source sentence x . First we parse the string with 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 traverse tree t_x in some suitable order, such as the pre-order. At each step, if the current node of the tree has a certain grammar rule of G_1 , then we apply the corresponding rule of target grammar G_2 and thus we append a few child nodes to the target tree. At the end of the 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 for the later stages of compilation, and to reformat the tree as needed. This transformation is an instance of language abstraction (p. 24). 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 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, then 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. 302. Therefore it suffices to interchange source and target grammars, and thus to obtain

source grammar G_1	target grammar G_2
$E \rightarrow \text{add } TE$	$E \rightarrow T + E$
$E \rightarrow T$	$E \rightarrow T$
$T \rightarrow \text{mult } FT$	$T \rightarrow F \times T$
$T \rightarrow F$	$T \rightarrow F$
$F \rightarrow E$	$F \rightarrow '(E)'$
$F \rightarrow i'$	$F \rightarrow i$

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, consider the following multiple derivations of source string i' :

$$E \Rightarrow T \Rightarrow F \Rightarrow i' \quad E \Rightarrow T \Rightarrow F \Rightarrow E \Rightarrow T \Rightarrow F \Rightarrow i' \quad \dots$$

each one of which produces a distinct image, as follows:

$$E \Rightarrow T \Rightarrow F \Rightarrow i \quad E \Rightarrow T \Rightarrow F \Rightarrow (E) \Rightarrow (T) \Rightarrow (F) \Rightarrow (i) \quad \dots$$

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 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:

$$S \rightarrow \frac{a}{b} S \quad S \rightarrow \frac{a}{c} S \quad S \rightarrow \frac{a}{d}$$

where the source grammar $S \rightarrow aS \mid a$ is unambiguous, yet the translation $\tau(aa) = \{bd, cd\}$ is not single valued, because the first two rules of the translation grammar correspond to the same source rule $S \rightarrow aS$.

The next property (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, and
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 ambiguity, we have not considered the ambiguity of the translation grammar G_τ itself, but only of its source/target components separately, because it is not relevant to ensure the single-valued translation property. The next example (Example 5.8) shows that an unambiguous translation grammar may have an ambiguous source grammar, and thus cause the translation to be ambiguous.

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

$$S \rightarrow \frac{\text{if } c \text{ then}}{\text{if } c \text{ then}} S \xrightarrow{\varepsilon} \mid \frac{\text{if } c \text{ then}}{\text{if } c \text{ then}} S \xrightarrow{\text{else}} S \xrightarrow{\varepsilon} \mid a$$

is unambiguous. Yet the underlying source grammar, which defines the conditional instruction without end mark, is a typical case of ambiguity (p. 52). The translation produces two images of the same source string, as follows:

$$\text{if } c \text{ then if } c \text{ then } a \quad \begin{matrix} \downarrow & \end{matrix} \quad \begin{matrix} \text{end_if} \\ \text{else} \end{matrix} \quad \begin{matrix} \downarrow & \end{matrix} \quad \begin{matrix} \text{end_if} \\ \uparrow \end{matrix} \quad \begin{matrix} \end{matrix}$$

$$\text{end_if end_if}$$

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 cause the translation to become multi-valued. Parser construction tools help because they routinely check the source grammar for determinism, which excludes ambiguity.

5.4.3 Translation Grammars and Pushdown Transducers

Much as the recognizer of a context-free language, context-free transducers too need a pushdown or *LIFO* store.

A *pushdown transducer* or *IO-automaton* is like a pushdown automaton, 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

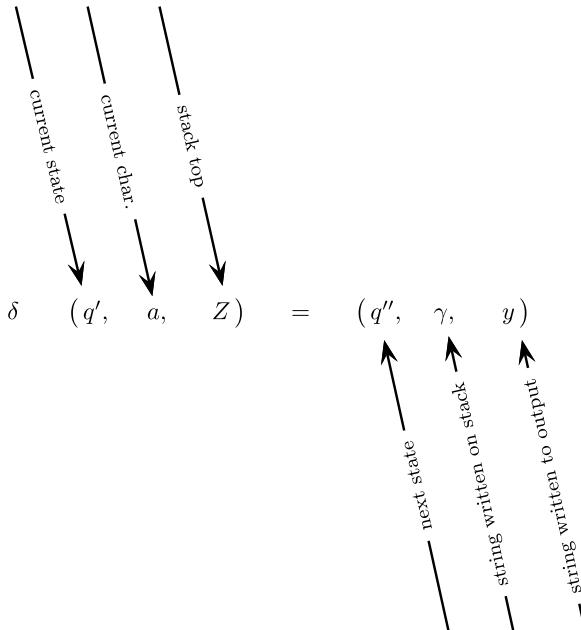


Fig. 5.2 Scheme of the move of a pushdown transducer

- 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.⁷ 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 we have $\delta(q', a, Z) = (q'', \gamma, y)$, 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 function meaning 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, as stated in Sect. 4.2.2, p. 146. The instantaneous configuration of the pushdown transducer is defined as a 4-tuple $(q, y, \eta, z) \in (Q \times \Sigma^* \times \Gamma^* \times \Delta^*)$, includ-

⁷It is also possible to specify the output by a separate output function (usually denoted λ). We skip the formalization of the case of nondeterministic transducer: the range of function δ would become the powerset \wp of the preceding cartesian product.

ing:

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 \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
$(q, ax, \eta Z, z)$	$(p, x, \eta\gamma, zy)$	Reading move $\delta(q, a, Z) = (p, \gamma, y)$
$(q, ax, \eta Z, z)$	$(p, ax, \eta\gamma, zy)$	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):

$$\tau(x) = z \iff (q_0, x, Z_0, \varepsilon) \xrightarrow{*} (q, \varepsilon, \lambda, z) \quad \text{with } q \in F \text{ and } \lambda \in \Gamma^*$$

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, the latter is procedural and helps in compilation. Their equivalence is stated next (Property 5.9).

Property 5.9 (Equivalence of translation relations and automata) 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 as is used to construct the pushdown recognizer of language $L(G_\tau)$ (Table 4.1 on p. 144). 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 without loss of generality, it helps to reorganize the source and target strings that occur in a grammar rule, in such a way that the first character is a source one, where possible.

More precisely, we make the following hypotheses on the form of the source/target pairs $\frac{u}{v}$ that occur in the rules,⁸ where $u \in \Sigma^*$ and $v \in \Delta^*$:

1. For any pair $\frac{u}{v}$ we have $|u| \leq 1$, 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 $\frac{a_1}{v} \frac{a_2}{\epsilon}$ without affecting the translation.
2. No rule may contain the following substrings:

$$\frac{\epsilon}{v_1} \frac{a}{v_2} \quad \text{or} \quad \frac{\epsilon}{v_1} \frac{\epsilon}{v_2} \quad \text{where } v_1, v_2 \in \Delta^*$$

Should such combinations be present in a rule, then they can be, respectively, replaced by the equivalent pairs $\frac{a}{v_1 v_2}$ or $\frac{\epsilon}{v_1 v_2}$.

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 $\frac{\epsilon}{v}$ with $v \in \Delta^+$, and of type $\frac{b}{w}$ with $b \in \Sigma$ and $w \in \Delta^*$, occurring in some grammar rule. The transducer moves are constructed as described 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 a computed by a nondeterministic pushdown transducer.

Example 5.11 (Nondeterministic pushdown transducer) Consider the source language L ,

$$L = \{a^* a^m b^m \mid m > 0\}$$

⁸For completeness we mention that sometimes in the scientific literature on formal languages, the source/target fractions $\frac{u}{v}$ in the grammar rules are denoted as $u\{v\}$, e.g., $X \rightarrow \alpha u\{v\}\beta$ instead of $X \rightarrow \alpha \frac{u}{v} \beta$; but such a notation is not used in this book.

Table 5.1 Correspondence between translation grammar rules and pushdown translator moves (if $n = 0$ then the sequence $A_1 \dots A_n$ is absent)

#	Rule	Move	Comment
1	$A \rightarrow \frac{\varepsilon}{v} BA_1 \dots A_n$ $n \geq 0$ $v \in \Delta^+$ $B \in V$ $A_i \in (C \cup V)$	If $top = A$ then write (v); pop; push ($A_n \dots A_1 B$)	Emit the target string v and push on stack the prediction string $BA_1 \dots A_n$
2	$A \rightarrow \frac{b}{w} A_1 \dots A_n$ $n \geq 0$ $b \in \Sigma$ $w \in \Delta^*$ $A_i \in (C \cup V)$	If $cc = b \wedge top = A$ then write (w); pop; push ($A_n \dots A_1$); advance the reading head	Char b was the next expected and has been read; emit the target string w ; push the prediction string $A_1 \dots A_n$
3	$A \rightarrow BA_1 \dots A_n$ $n \geq 0$ $B \in V$ $A_i \in (C \cup V)$	If $top = A$ then pop; push ($A_n \dots A_1 B$)	Push the prediction string $BA_1 \dots A_n$
4	$A \rightarrow \frac{\varepsilon}{v}$ $v \in \Delta^+$	If $top = A$ then write (v); pop	Emit the target string v
5	$A \rightarrow \varepsilon$	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

Table 5.2 Moves of a nondeterministic pushdown transducer

#	Rule	Move
1	$S \rightarrow \frac{a}{\varepsilon} S \frac{\varepsilon}{c}$	If $cc = a \wedge top = S$ then pop; push ($\frac{\varepsilon}{c} S$); advance the reading head
2	$S \rightarrow A$	If $top = S$ then pop; push (A)
3	$A \rightarrow \frac{a}{d} A \frac{b}{\varepsilon}$	If $cc = a \wedge top = A$ then pop; write (d); push ($\frac{b}{\varepsilon} A$); advance the reading head
4	$A \rightarrow \frac{a}{d} \frac{b}{\varepsilon}$	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

and define the following translation τ on L :

$$\tau(a^k a^m b^m) = d^m c^k \quad \text{where } k \geq 0 \text{ and } m > 0$$

The translation τ first changes letter b to letter d , 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.

Example 5.12 (Context-free nondeterministic translation) The translation function τ ,

$$\tau(u) = u^R u \quad \text{where } u \in \{a, b\}^*$$

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
$S \rightarrow \frac{\epsilon}{a} S \frac{a}{a}$	$S \rightarrow Sa$	$S \rightarrow aSa$
$S \rightarrow \frac{\epsilon}{b} S \frac{b}{b}$	$S \rightarrow Sb$	$S \rightarrow bSb$
$S \rightarrow \frac{\epsilon}{\epsilon}$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$

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.⁹ The only way to reverse a string by means of a pushdown device is to store the string in the stack and to output it in the order it is popped. But popping destroys the stored string and makes it unavailable for copying to the output at a later moment.

Nondeterministic algorithms are used infrequently in compilation. In the next section we develop translator construction methods suitable for use with the widespread deterministic parsers.

5.4.4 Syntax Analysis with Online Translation

Given a context-free translation scheme, the previous construction produces a pushdown transducer that is often nondeterministic and generally unpractical to be used 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

⁹For a proof see [1, 2].

for deterministic parsing. To compute the image of a string, the parser emits the corresponding translation as it completes a syntactic subtree.

We know that bottom-up and top-down parsers differ in the construction order of the syntax tree. A question to be investigated is how the order interferes with the possibility of correctly producing the target image. The main result will be that the top-down parsing order is fully compatible, whereas the bottom-up order places some restrictions on the translation.

5.4.5 Top-Down Deterministic Translation by Recursive Procedures

We recall there are two techniques for building efficient top-down deterministic parsers: as pushdown automata or as recursive descent procedures. Both techniques can be simply extended toward the construction of translators. 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. For each nonterminal symbol, a procedure has the task of recognizing the substrings 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 5.13) should be enough to explain such a straightforward modification of a recursive descent parser.

Example 5.13 (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 by

$$v \times (v + v) \quad \Rightarrow \quad vvv \text{ add mult}$$

The source language is defined by the extended BNF grammar G_1 (axiom E):

$$G_1 \quad \left\{ \begin{array}{l} E \rightarrow T(+T \mid -T)^* \\ T \rightarrow F(\times F \mid \div F)^* \\ F \rightarrow v \mid ('E') \end{array} \right.$$

Figure 5.3 (left) represents the machine for nonterminal E : the machine meets the $ELL(1)$ condition and the relevant guide sets are written in braces.

The recursive descent procedure for nonterminal E is immediately derived and shown in Fig. 5.4 (left); we recall that function *next* returns the current input token; and for simplicity, the check of the error cases is omitted.

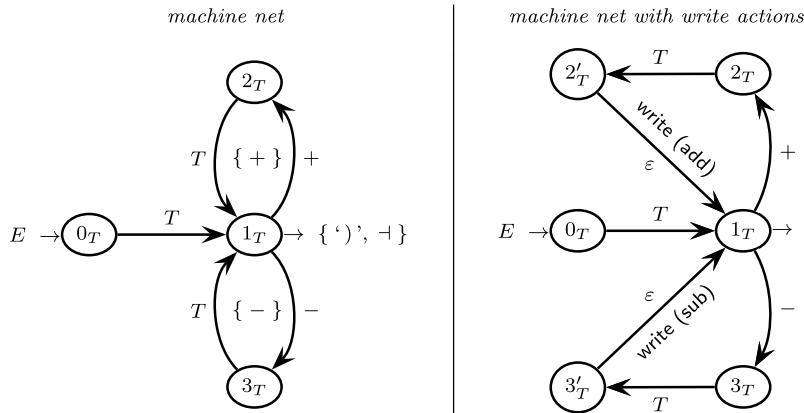


Fig. 5.3 Machine M_E that represents the EBNF axiomatic rule $E \rightarrow T(+T \mid -T)^*$ of the source grammar in Example 5.13 (left), and the same machine M_E augmented with write actions and states where necessary (right)

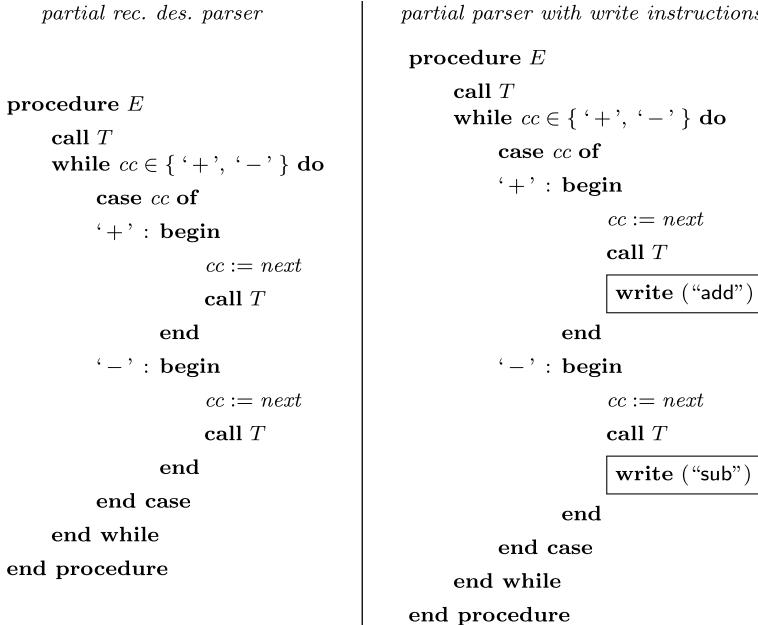


Fig. 5.4 Recursive descent syntactic procedures for nonterminal E , without (left) and with (right) write instructions, which are framed

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 with write instructions is easily derived and is also shown in Fig. 5.4 (right). Similarly the procedures for nonterminals T and F can be augmented to write the other target symbols, but for brevity here they are omitted.

The machines and procedures that compute the complete translation correspond to the following extended BNF translation grammar G_τ (axiom E):

$$G_\tau \quad \left\{ \begin{array}{l} E \rightarrow T \left(\frac{+}{\varepsilon} T \frac{\varepsilon}{\text{add}} \mid \frac{-}{\varepsilon} T \frac{\varepsilon}{\text{sub}} \right)^* \\ T \rightarrow F \left(\frac{\times}{\varepsilon} F \frac{\varepsilon}{\text{mult}} \mid \frac{\div}{\varepsilon} F \frac{\varepsilon}{\text{div}} \right)^* \\ F \rightarrow \frac{v}{v} \mid \frac{(')}{\varepsilon} E \frac{')}{\varepsilon} \end{array} \right.$$

where the source and target alphabets Σ and Δ are as follows:

$$\Sigma = \{ '+', '- ', ' \times ', ' \div ', v, '(', ')' \}$$

$$\Delta = \{\text{add}, \text{sub}, \text{mult}, \text{div}, v\}$$

and the round parentheses do not appear translated in the target alphabet Δ , since they are unnecessary in the postfix polish form.

In the grammar rules, the elements of the target string, i.e., the characters of alphabet Δ , appear in the denominators of the fractions.

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

Consider again a context-free translation scheme and assume the source grammar is suitable for bottom-up deterministic parsing, by the *ELR(1)* condition, p. 185 (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 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

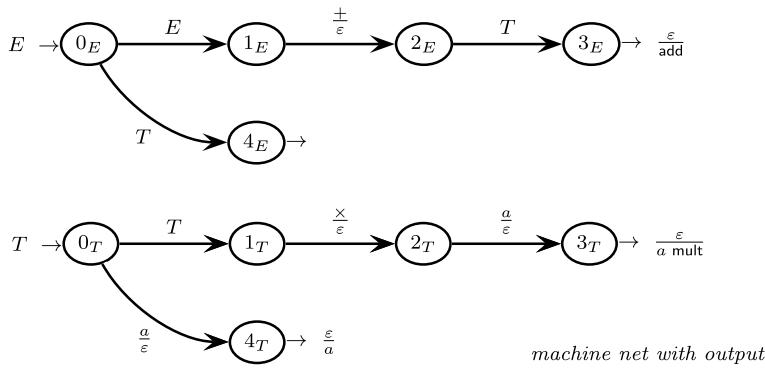


Fig. 5.5 Machine net of the BNF translation grammar of Example 5.14

two different and contradictory write actions, which is impossible for a deterministic transducer. This reasoning explains the impediment to emit the output during a shift move of the translator.

On the other hand, when a reduction applies, exactly one source grammar rule has been recognized, i.e., the final state of the corresponding machine has been entered. 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 5.14) where the translation grammar (not extended) has rules that contain output symbols at the rule end, so that the pilot automaton can be enriched with write actions in the reduction m-states.

Example 5.14 (Translation of expressions to postfix notation) The BNF translation grammar below, which specifies certain formulas that use two infix signs¹⁰ to be translated to postfix operators, is represented as a machine network (axiom E) in Fig. 5.5:

$$E \rightarrow E \frac{+}{\varepsilon} T \frac{\varepsilon}{\text{add}} \mid T \quad T \rightarrow T \frac{\times a}{\varepsilon} \frac{\varepsilon}{a \text{ mult}} \mid \frac{a \varepsilon}{\varepsilon a}$$

Notice that care has been taken to position all the target characters as suffix at the rule end (some rule may have none at all). 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.

On the contrary, in the next case (Example 5.15) the presence of write actions on shift moves makes the translation incorrect.

¹⁰More precisely, two-level infix arithmetic expressions of type sum-of-product, with variables a but without parentheses.

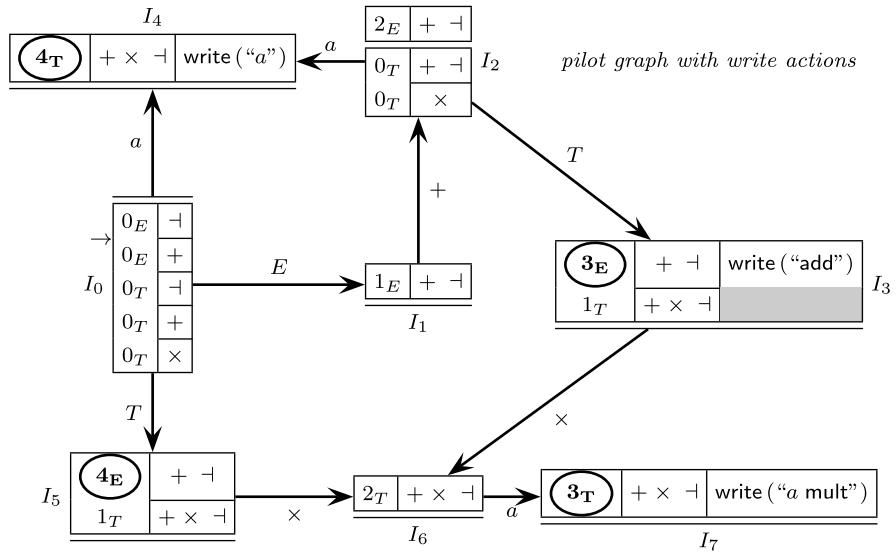


Fig. 5.6 Pilot of the translator of Example 5.14 with write actions in the reduction m-states. The closure of m-states I_0 and I_2 is shown step-by-step

Example 5.15 (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. We have

$$G_\tau : S \rightarrow \frac{a}{b} S \frac{c}{e} S \mid \frac{a}{\varepsilon} \frac{c}{\varepsilon} \quad \begin{aligned} \tau(aaccac) &= be \\ \tau(ac) &= \varepsilon \end{aligned}$$

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. Yet the attempt aborts 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 necessary to put the translation grammar into a form (Definition 5.16) that confines write actions within reduction moves.

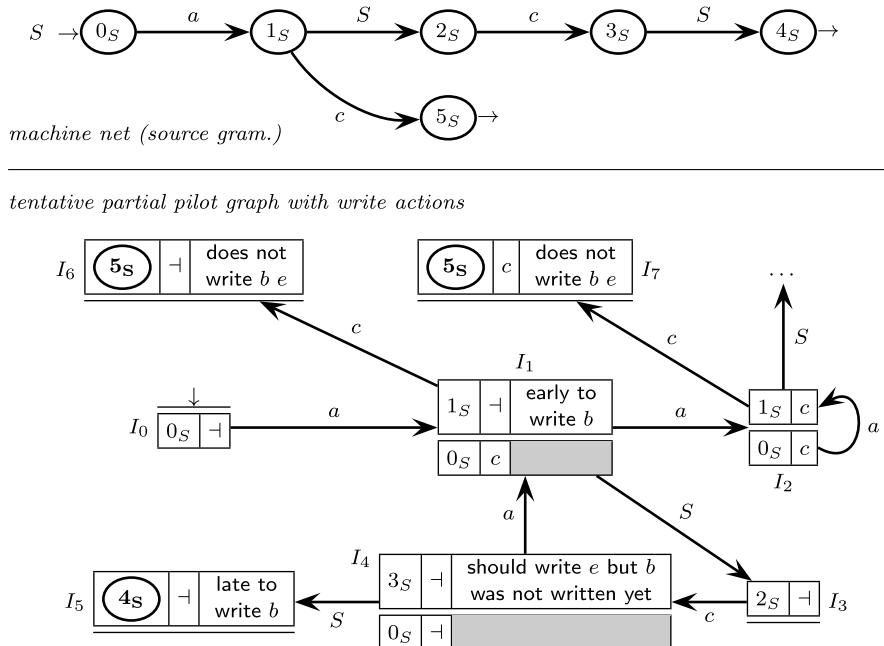


Fig. 5.7 Machine net and tentative partial pilot with write actions (Example 5.15)

Definition 5.16 (Postfix translation grammar) A translation grammar or scheme is in the *postfix normal form* if every target grammar rule has the form $A \rightarrow \gamma w$, 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. 299) is in the postfix form, while Examples 5.12 (p. 310) and 5.15 (p. 315) are not.

One may wonder what loss, 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 5.17 we explain the transformation of a generic translation grammar into postfix normal form.

Algorithm 5.17 (Converting a translation grammar to postfix form) Consider in turn each rule $A \rightarrow \alpha$ of the given translation grammar. If the rule violates the postfix condition, find the longest target string $v \in \Delta^+$ that occurs in the rightmost position in α , and transcribe the rule as

$$A \rightarrow \gamma \frac{\varepsilon}{v} \eta$$

where γ is any string, and η is a nonempty string devoid of target characters. Replace this rule with the next ones:

$$A \rightarrow \gamma Y \eta \quad Y \rightarrow \frac{\varepsilon}{v}$$

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 rightmost target string within γ and repeat the transformation. 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 is in the postfix normal form.

The next example (Example 5.18) illustrates the transformation to postfix, and it should convince the reader that the original and transformed grammars define the same translation.

Example 5.18 (Grammar transformation to postfix normal form) Examining the translation grammar $G_\tau : S \rightarrow \frac{a}{b} S \frac{c}{e} S \mid ac$ of the previous example (Example 5.15), we notice that in the target grammar G_2 the pointed letters of the axiomatic rule

$$\begin{array}{c} \downarrow \quad \downarrow \\ S \rightarrow b \ S \ e \ S \end{array}$$

violate the postfix form. We apply the normalization algorithm (Algorithm 5.17) and we introduce two new nonterminals: A to represent the left parenthesis b ; and B for the right one e ; as follows:

$$\begin{array}{lll} G_\tau: & S \rightarrow A S C S \mid ac & A \rightarrow \frac{a}{b} \quad C \rightarrow \frac{c}{e} \\ & & \\ G_{2\text{postfix}}: & S \rightarrow A S C S \mid \varepsilon & A \rightarrow b \quad C \rightarrow e \end{array}$$

It is easy to check that the target grammar $G_{2\text{postfix}}$ defines the same translation as the original target grammar G_2 does. 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 the axiomatic machine M_S may end at the same one final state 4_S . The pilot of the postfix grammar, partially shown in Fig. 5.8, writes the translation output only at reduction.

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 non-terminal 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 5.19) for being deterministic.

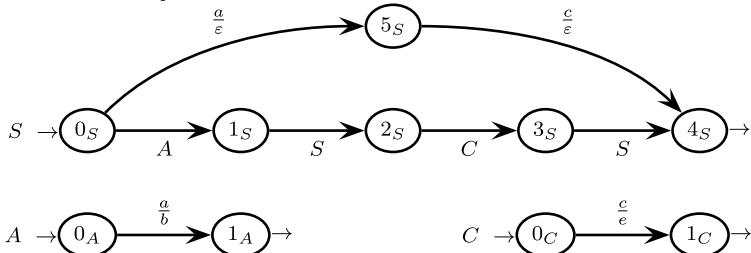
BNF translation grammar in postfix form

$$G_\tau: \quad S \rightarrow A \ S \ C \ S \mid a \ c$$

$$A \rightarrow \frac{a}{b}$$

$$C \rightarrow \frac{c}{e}$$

machine net with output



partial pilot graph with write actions

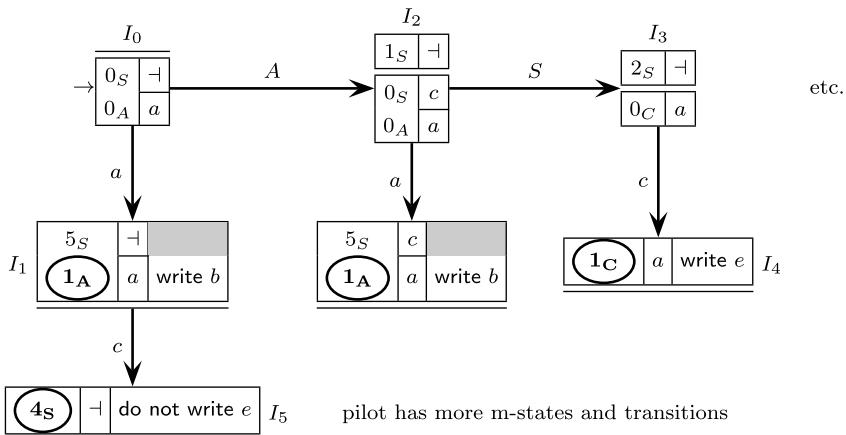


Fig. 5.8 Translation grammar G_τ in the postfix form, machine net with output, and partial pilot with write actions (Example 5.18)

Property 5.19 (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 only writes on the output at reduction moves.

In essence the postfix form allows the parser to defer its write actions until it reaches a state where the action is uniquely identified.

This method has some inconveniences. The introduction of new nonterminal symbols, such as A and B in Example 5.18, makes the new grammar less readable. Another nuisance may come from rule normalization, when empty rules such as $Y \rightarrow \varepsilon$ are added to the source grammar. Empty rules tend to increase the length k of the look-ahead needed for parsing; and in some cases the $LR(k)$ property may be lost, as the next example (Example 5.20) shows.

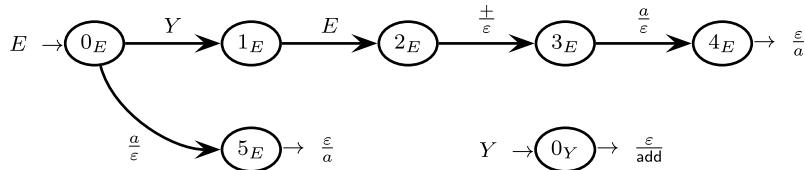


Fig. 5.9 Machine net with output of a BNF translation grammar normalized in the postfix form (Example 5.20)

Example 5.20 (Loss of LR(1) property caused by normalization) The translation of an infix expression¹¹ to prefix form is specified by the grammar G_τ below. By looking 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
$E \rightarrow \frac{\varepsilon}{\text{add}} E \frac{+a}{a}$	$E \rightarrow E + a$	$E \rightarrow \text{add } Ea$
$E \rightarrow \frac{a}{a}$	$E \rightarrow a$	$E \rightarrow a$

Here is the postfix form G'_τ of grammar G_τ , obtained by applying the normalization algorithm (Algorithm 5.17):

G'_τ postfix	G'_1	G'_2
$E \rightarrow YE \frac{+a}{\varepsilon} \frac{\varepsilon}{a}$	$E \rightarrow YE + a$	$E \rightarrow YEa$
$E \rightarrow \frac{a}{\varepsilon} \frac{\varepsilon}{a}$	$E \rightarrow a$	$E \rightarrow a$
$Y \rightarrow \frac{\varepsilon}{\text{add}}$	$Y \rightarrow \varepsilon$	$Y \rightarrow \text{add}$

The machine net of the postfix version G'_τ is in Fig. 5.9. It is easy to check that grammar G'_τ defines the same translation as the original one G_τ . However, while the original source grammar G_1 satisfies the LR(1) condition, the new rule $Y \rightarrow \varepsilon$ of the postfix source grammar G'_1 , i.e., the fact that state 0_Y is both initial and final, 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 utilization 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. 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

¹¹The expression is simply a summation $a + a + \dots + a$, of one or more terms.

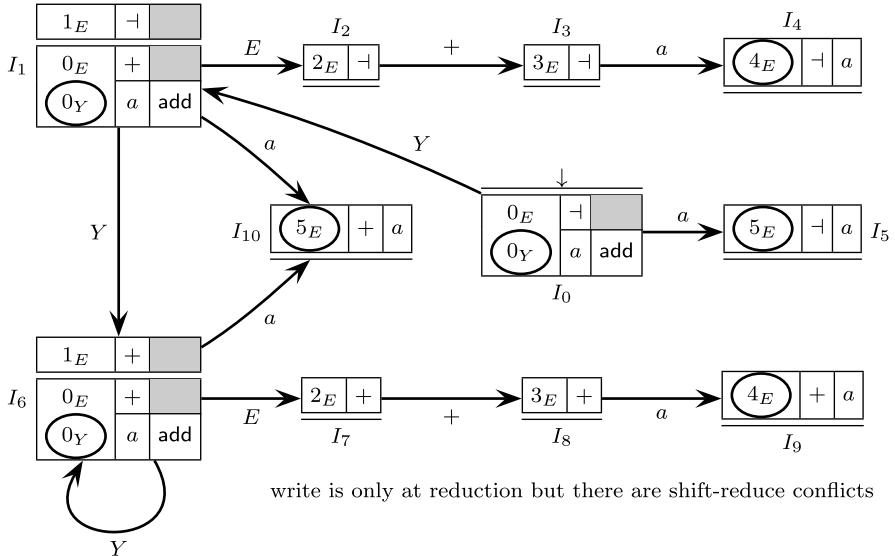


Fig. 5.10 Pilot $LR(1)$ of the trans. grammar of Fig. 5.9, with shift–reduce conflicts in the m-states I_0 , I_1 and I_6 (a few m-states have write actions)

outputting the sequence of labels of the source rules, in the order they occur in the tree derivation.

Given a source grammar with rules labeled for reference, the next syntactic translation scheme produces the label sequence of a derivation:

Label	Original translation rule	Modified rule
r_i	$A \rightarrow \alpha$	$A \rightarrow \alpha \frac{\varepsilon}{r_i}$

The image produced by the above translation is exactly the sequence of rule labels used by the rightmost derivation. Since by hypothesis the source grammar is $LR(1)$ and the scheme is postfix, the parser enriched with write actions easily computes this translation.

5.4.7 Comparisons

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 Translations

Just like context-free grammars have as a special case the right-linear grammars, which in turn have regular expressions and finite state 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 state transducers.

Consider for instance the following translation τ :

$$\begin{cases} a^{2n} \xrightarrow{\tau} b^{2n} & n \geq 0 \\ a^{2n+1} \xrightarrow{\tau} c^{2n+1} & n \geq 0 \end{cases}$$

Translation τ can be specified by the following right-linear translation grammar G_τ (axiom A_0):

$$G_\tau = \left\{ \begin{array}{l} A_0 \rightarrow \frac{a}{c} A_1 \mid \frac{a}{c} \mid \frac{a}{b} A_3 \mid \varepsilon \\ A_1 \rightarrow \frac{a}{c} A_2 \mid \varepsilon \\ A_2 \rightarrow \frac{a}{c} A_1 \\ A_3 \rightarrow \frac{a}{b} A_4 \\ A_4 \rightarrow \frac{a}{b} A_3 \mid \varepsilon \end{array} \right.$$

The well-known equivalence between right-linear grammars and finite state automata can be extended by defining regular translation expressions.

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. Then a sentence generated by such a regular expression 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. In this manner, such a regular expression model defines a translation relation to be called *regular* or *rational*.

In this way the above translation 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

$$\frac{a}{c} \cdot \left(\frac{a^2}{c^2}\right)^2 = \frac{a}{c} \cdot \frac{a^2}{c^2} \cdot \frac{a^2}{c^2} = \frac{a^5}{c^5} \in L(e_\tau)$$

then corresponds to the pair (a^5, c^5) in the translation relation ρ_τ defined by the expression e_τ .

Example 5.21 (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:

$$\Sigma = \{1, ‘/’\} \quad \Delta = \{1, ‘÷’, ‘:’\}$$

The source strings have the form $c/(c)^*$, where c stands for any unary number denoted by 1^+ . Two valid translations are the following:

$$(3/5/2, 3 : 5 : 2) \quad (3/5/2, 3 \div 5 \div 2)$$

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. Incidentally the inverse translation is an alphabetic homomorphism. The translation is defined by the r.e.

$$(1, 1)^+((‘/’, ‘:’)(1, 1)^+)^* \cup (1, 1)^+((‘/’, ‘÷’)(1, 1)^+)^*$$

or using the more readable fractional notation, by the following r.e.:

$$\left(\frac{1}{1}\right)^+ \left(\frac{/}{:} \left(\frac{1}{1}\right)^+\right)^* \cup \left(\frac{1}{1}\right)^+ \left(\frac{/}{\div} \left(\frac{1}{1}\right)^+\right)^*$$

The terms produced by applying a derivation are strings of fractions, i.e., string pairs. Consider the following derived string:

$$\left(\frac{1}{1}\right)^1 \left(\frac{/}{\div} \left(\frac{1}{1}\right)^2\right)^1 = \frac{1}{1} / \frac{1}{1} \frac{1}{1}$$

project it on the top and bottom components, and thus obtain the pair of source/target strings $(1/11, 1 \div 11)$.

We can summarize the previous discussion and examples (Example 5.21) about regular translations, by the next definition (Definition 5.22).

Definition 5.22 (Regular (or rational) translation) A *regular* or *rational translation expression*, for short 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 component, 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. But 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 following example (Example 5.23).

Example 5.23 (Consistent transliteration of an operator (Example 5.21) continued) The recognizer of the regular translation relation is shown in Fig. 5.11.

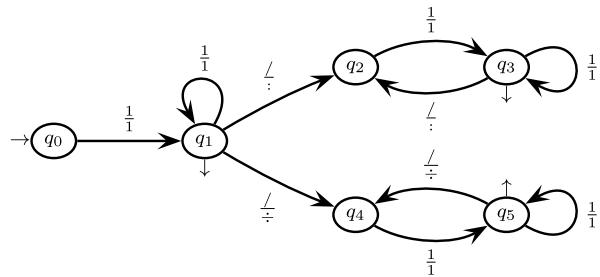
This automaton can be viewed as a machine with *two* read-only input tapes, in short a $2I$ -machine, each one with an independent reading head, respectively containing the source string x and the target string y . Initially the heads are positioned on the first characters and the machine is in the start 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 “ \div ” 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.¹²

This automaton can check that two strings, such as

$$(11' / '1, 1' \div '1) \equiv \frac{11/1}{1 \div 1}$$

¹²This model is known as a Rabin and Scott machine. For greater generality such a machine may be equipped with more than two tapes, in order to define a relation between more than two languages.

Fig. 5.11 $2I$ -automaton of the r.t.e. of Examples 5.21 and 5.23



do not correspond in the translation, because the following computation:

$$q_0 \xrightarrow{\frac{1}{1}} q_1$$

admits no continuation with the next pair, i.e., fraction $\frac{1}{\div}$.

It is sometimes convenient to assume 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 the wrong idealizations for modeling a compiler, because in compilation the target string is not given, but 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.

5.5.1.1 Forms of Two-Input Automata

When designing a two-input recognizer, we can assume without loss of generality that each move reads exactly one character or nothing from the source tape, while it may read one string of any length ≥ 0 (so including null) from the target tape; the following definition (Definition 5.24) formalizes the concept.

Definition 5.24 (Two-input automaton or $2I$ -automaton) A finite automaton with two inputs or $2I$ -automaton is defined as a usual finite automaton (Sect. 3.5.2, p. 107) by a set of states Q , an initial state q_0 and a set $F \subseteq Q$ of final states. The transition function δ is as follows:¹³

$$\delta: Q \times (\Sigma \cup \{\varepsilon\}) \times \Delta^* \rightarrow \wp(Q)$$

The $2I$ -automaton has a move that can behave as follows:

- on the first tape: if we have $q' \in \delta(q, a, v)$, the move reads character $a \in \Sigma$; and if it holds or $q' \in \delta(q, \varepsilon, v)$, the move does not read anything
- on the second tape: the move reads string $v \in \Delta^*$, possibly null
- and the move enters the next state q'

The recognition condition is that a computation reaches a final state.

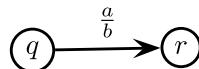
¹³Remember that symbol \wp is the powerset, i.e., the set of all the subsets.

Imagine now to project the arc labels of a $2I$ -automaton on the first component. The resulting automaton has just one input tape with symbols from the source alphabet Σ , and is termed the input automaton *subjacent* to the original machine; it recognizes the source language.

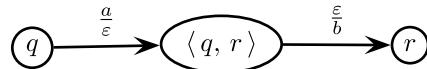
Sometimes another *normal* form of a $2I$ -automaton is used, characterized by the fact that each move reads exactly one character either from the source tape or from the target tape, but not from both. More precisely the arc labels are of the following two types:

- label $\frac{a}{\varepsilon}$ with $a \in \Sigma$, i.e., read one character from source, or
- label $\frac{\varepsilon}{b}$ with $b \in \Delta$, i.e., read one character from target

This means that a machine in the normal form shifts only one head per move, and of 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:



is replaced by a cascade of normalized moves, like



where symbol $\langle q, r \rangle$ is a new state.¹⁴

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 remark on notation: in an arc label we usually drop a component when it 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 expression says that a sequence of a 's, if followed by b , is translated to d ; and if it is followed by c , it is translated to e .

¹⁴According to Definition 5.24, a move of a $2I$ -automaton may read nothing from both tapes, i.e., the arc may be labeled by $\frac{\varepsilon}{\varepsilon}$; such strictly null moves can always be eliminated, too; just think that they are analogous to the spontaneous moves of recognizer automata, and that they can be eliminated in a similar way; here we do not go into such subtle details.

5.5.1.2 Equivalence of Models

In accordance with the well-known equivalence of regular expressions and finite state automata, we state a similar property for translations (Property 5.25).

Property 5.25 (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 $(u, v) = \frac{u}{v}$, where we have $u \in \Sigma^*$ 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 5.26 (Nivat theorem) The following four conditions are equivalent:

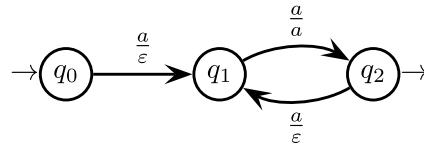
1. The translation relation ρ_τ is defined by a right-linear (or left-linear) translation grammar G_τ .
2. The translation relation ρ_τ is defined by a $2I$ -automaton.
3. The translation relation $\rho_\tau \subseteq \Sigma^* \times \Delta^*$ is regular.
4. There exist an alphabet Ω , a regular language R over Ω , and two alphabetic homomorphisms $h_1: \Omega \rightarrow \Sigma \cup \{\epsilon\}$ and $h_2: \Omega \rightarrow \Delta \cup \{\epsilon\}$, such that

$$\rho_\tau = \{(h_1(z), h_2(z)) \mid z \in R\}$$

Example 5.27 (Division by two) The image string is the halved source string. The translation relation $\{(a^{2n}, a^n) \mid n \geq 1\}$ is defined by the following r.t.e.:

$$\left(\frac{aa}{a}\right)^+$$

An equivalent $2I$ -automaton A is shown here:



To apply the Nivat theorem (clause (4) of Property 5.26), we derive from automaton A the following r.t.e.:

$$\left(\frac{aa}{\epsilon a}\right)^+$$

and we rename for clarity the pairs

$$\frac{a}{\epsilon} = c \quad \frac{a}{a} = d$$

Next consider the alphabet $\Omega = \{c, d\}$. The r.t.e. defines the regular language $R = (cd)^+$, 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 $z = cdc \in R$ we have $h_1(z) = aaaa$ and $h_2(z) = aa$.

To illustrate Nivat theorem (clause (1) of Property 5.26), we apply the well-known equivalence of finite automata and right-linear grammars (p. 103), to obtain the following equivalent right-linear translation grammar:

$$S \rightarrow \frac{a}{\varepsilon} Q_1 \quad Q_1 \rightarrow \frac{a}{a} Q_2 \quad Q_2 \rightarrow \frac{a}{\varepsilon} Q_1 \mid \varepsilon$$

Each grammar rule corresponds to a move of the $2I$ -automaton. Rule $Q_2 \rightarrow \varepsilon$ is a short notation for $Q_2 \rightarrow \frac{\varepsilon}{\varepsilon}$.

We recall from Sect. 5.4 that the notation based on a translation grammar instead of a $2I$ -automaton or a r.t.e. has to be used for the syntactic translations, which have a context-free grammar as their support, because such translations require a pushdown stack and cannot be described by a simple finite automaton.

Several properties of regular languages, but not all their properties, have an analogous formulation for regular translation relations.¹⁵ 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 those relations that remain regular, it is possible to formulate a pumping lemma similar to the one for the regular languages on p. 73.

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. Consider for instance the following languages L_1 , L_2 and translation τ :

$$L_1 = (a \mid b)^* \quad L_2 = (a \mid b)^* \quad \tau(x) = x^R \quad \text{with } x \in L_1$$

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.

5.5.2 Translation Functions and Finite Transducers

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 trans-*

¹⁵See the books [5, 12].

ducer 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 a nondeterministic example (Example 5.28).

Example 5.28 (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 ,

$$\rho_\tau = \{(a^{2n}, b^{2n}) \mid n \geq 0\} \cup \{(a^{2n+1}, c^{2n+1}) \mid n \geq 0\}$$

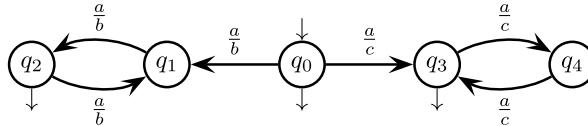
defines the following translation function τ :

$$\tau(a^n) = \begin{cases} b^n & \text{for } n \geq 0 \text{ even } (n=0 \text{ is even}) \\ c^n & \text{for } n \geq 1 \text{ odd} \end{cases}$$

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 target labels of such arcs are different.¹⁶ Therefore the $2I$ -machine can deterministically decide if two strings memorized on the two tapes, such as $\frac{aaaa}{bbbb}$, correspond to each other in the translation relation.

Although the transducer or *IO-automaton* has the same state-transition graph as the $2I$ -automaton, the meaning of an arc, e.g., $q_0 \xrightarrow{\frac{a}{b}} q_1$, is entirely different: in the state q_0 it reads character a from the input, writes character b to the output and enters state q_1 . We observe that 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 aa :

$$q_0 \rightarrow q_1 \rightarrow q_2 \quad q_0 \rightarrow q_3 \rightarrow q_4$$

but only the former succeeds in reaching a final state, namely q_2 , and thus only its output is considered: $\tau(aa) = bb$. Notice that the transducer nondeterminism shows up in the input automaton subjacent to the transducer, which is nondeterministic.

¹⁶If 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 [12].

Moreover, it should be intuitively clear that the requested translation cannot be computed by a 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 5.28), 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.

5.5.2.1 Sequential Transducers

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. Finally, 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.¹⁷ The next definition formalizes the concept (Definition 5.29).

Definition 5.29 (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

The domains and images of these three functions are the following:

$$\delta: Q \times \Sigma \rightarrow Q \quad \eta: Q \times \Sigma \rightarrow \Delta^* \quad \varphi: F \times \{\dashv\} \rightarrow \Delta^*$$

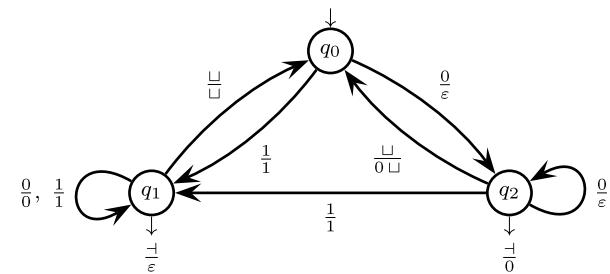
In the graphical presentation as a state-transition graph, the two functions $\delta(q, a) = r$ and $\eta(q, a) = u$ are represented by the arc $q \xrightarrow{\frac{a}{u}} r$, which means: in the state q , when reading character a , emit string u and move to the next state r . The final function $\varphi(r, \dashv) = v$ 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:

$$\tau(x) = \left\{ yz \in \Delta^* \mid \exists \text{ a labeled computation } \frac{x}{y} \text{ that ends in} \right. \\ \left. \text{the state } r \in F \text{ and we have } z = \varphi(r, \dashv) \right\}$$

¹⁷This is the terminology of [12]; others [5] call subsequential the same model. In electrical engineering a sequential machine is a quite similar logical device that computes a binary output that is a function of the sequence of input bits.

Fig. 5.12 Sequential transducer (*IO*-machine) of Example 5.30. Notice the graphical representation of the final function φ by means of the fractions $\frac{1}{\varepsilon}$ and $\frac{0}{0}$ that label the final darts that tag states q_1 and q_2 , respectively



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 be 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 5.29. The next example (Example 5.30) shows a simple use of a sequential translation, inspired to arithmetic.

Example 5.30 (Meaningless zeros) The source text is a list of binary integers, i.e., a list of bit sequences, separated by a blank represented by symbol “ \sqcup ”. The single-valued translation deletes all the meaningless zeros, i.e., the leading ones (and an integer of all zeroes is translated to only one zero). This translation is defined by the following r.t.e.:

$$\left(\left(\frac{0^+}{0} \mid \left(\frac{0}{\varepsilon} \right)^* 1 \left(\frac{0}{0} \mid \frac{1}{1} \right)^* \right) \sqcup \right)^* \left(\frac{0^+}{0} \mid \left(\frac{0}{\varepsilon} \right)^* 1 \left(\frac{0}{0} \mid \frac{1}{1} \right)^* \right) \dashv$$

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 .

Examples of computation. When translating the source string $00 \sqcup 01$, 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 00 , the machine traverses the state sequence $q_0 q_2 q_2$, writes the target string $\varepsilon \cdot \varepsilon$ and finally writes the target character 0.

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 composition of two sequential transducers can be replaced by just one transducer of the same kind, though typically larger.

5.5.2.2 Two Opposite Passes

Given a single-valued translation specified by a regular translation expression or by a $2I$ -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 which scan the string in opposite directions. In the first pass a sequential transducer scans from left to right and converts the source string to 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. The next example (Example 5.31) shows a simple case.

Example 5.31 (Regular translation by two one-way passes in opposite directions (Example 5.28 on p. 328 continued)) We recall that the translation of string a^n to b^n if n is even or to 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 get known just at the end; but then it would be too late for writing the output, because number n is unbounded and thus exceeds the 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} ,

$$e_{\tau_1} = \left(\frac{a}{a'} \frac{a}{a''} \right)^* \left[\frac{a}{a'} \right]$$

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} ,

$$e_{\tau_2} = \left(\frac{a''}{b} \frac{a'}{b} \right)^* \cup \left(\frac{a'}{c} \frac{a''}{c} \right)^* \frac{a'}{c}$$

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 aa we have the following:

$$\tau_2((\tau_1(aa))^R) = \tau_2((a'a'')^R) = \tau_2(a''a') = bb$$

In fact a cascade of 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. The sequential translator model with look-ahead is implemented by compiler construction tools of widespread application, such as *Lex* and *Flex*.¹⁸

5.5.3 Closure Properties of Translations

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 ,

$$\tau(L) = \{y \in \Delta^* \mid y = \tau(x) \wedge x \in L\}$$

The question is the following: what type is the language family of the image language? 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 $\in CF$

¹⁸For a formalization of look-ahead sequential transducers, we refer to Yang [15].

Cases (1) and (3) descend from the Nivat theorem (Property 5.26 on p. 326) and from the fact that both language families REG and CF are closed under intersection with regular languages (p. 153). 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 obtained machine 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 is a pushdown automaton. 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 uu^R , which clearly belongs to a context-free language.

Case (4) is different because, as we know from Table 2.8 on p. 78, the intersection of two context-free languages L and L_1 is not always in the family CF . Therefore there is not any guarantee that a pushdown automaton will be able to recognize the image of L computed by a pushdown translator. The next example (Example 5.32) shows a situation of this kind.

Example 5.32 (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 (Example 2.84 on p. 76):

$$\tau(L) = \{a^n b^n c^n \mid n \geq 0\}$$

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:

$$G_\tau : \quad S \rightarrow \left(\frac{a}{a}\right)^* X \quad X \rightarrow \frac{b}{b} X \frac{c}{c} \mid \varepsilon$$

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 Translations

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, p. 86) 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 syntactic 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.¹⁹

A syntax-directed compiler performs two cascaded phases:

1. parsing or syntax analysis
2. semantic evaluation or analysis

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*.

¹⁹Formal semantic methods are needed if one has to prove that a compiler is correct, i.e., that for any source text the corresponding image expresses the intended meaning. For an introduction to formal semantics, see for instance [13, 14].

Table 5.3 Attribute grammar of Example 5.33

#	Syntax	Semantic functions		Comment
1	$N \rightarrow D \bullet D$	$v_0 := v_1 + v_2 \times 2^{-l_2}$		Add integer to fractional value divide by weight 2^{l_2}
2	$D \rightarrow DB$	$v_0 := 2 \times v_1 + v_2$	$l_0 := l_1 + 1$	Compute value and length
3	$D \rightarrow B$	$v_0 := v_1$	$l_0 := 1$	
4	$B \rightarrow 0$	$v_0 := 0$		Value initialization
5	$B \rightarrow 1$	$v_0 := 1$		

5.6.1.1 Introductory Example

Attribute grammar concepts are now introduced on a running example.

Example 5.33 (Converting a fractionary binary number to base 10 (Knuth²⁰)) 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 $1101 \bullet 01$ 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 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 rule, but 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.

Notice 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 stipulation:²¹

²⁰This historical example by Knuth introduced [8] attribute grammars as a systematization of compiler design techniques used by practitioners.

²¹Alternatively, a more verbose style is used in other texts, e.g., for the first function: v of N instead of v_0 .

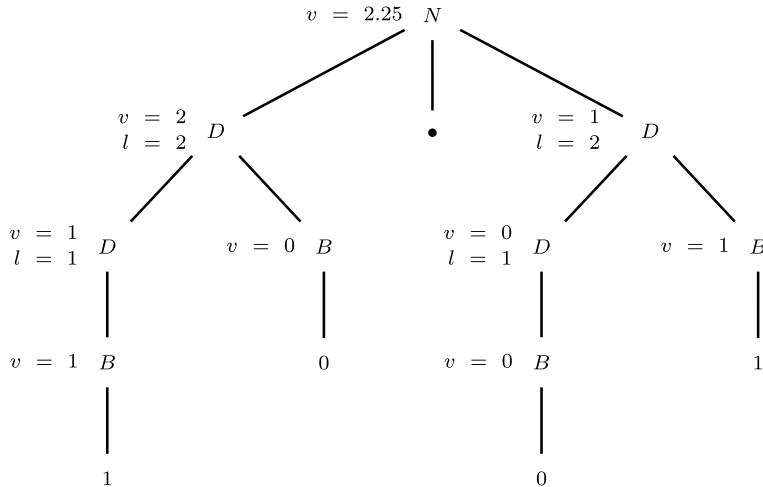


Fig. 5.13 Decorated syntax tree of Example 5.33

$$\underbrace{N}_0 \rightarrow \underbrace{D}_1 \bullet \underbrace{D}_2$$

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.

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 a 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 $10 \bullet 01$, 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 an 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 $10 \bullet 01$.

5.6.2 Left and Right Attributes

In the grammar of Example 5.33, attribute computation essentially flows from bottom to top because an attribute of the left part (father or parent) of a production rule is defined by a function having as arguments some attributes of the right part (children). But 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*²²) 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 we show a grammar featuring both left and right attributes.

Example 5.34 (Breaking a text into lines (Reps)) 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 .

²²The word *inherited* is used by object-oriented languages in a totally unrelated sense.

Table 5.4 Attribute grammar of Example 5.34

#	Syntax	Right attributes	Left attributes
1	$S_0 \rightarrow T_1$	$prec_1 := -1$	
2	$T_0 \rightarrow T_1 \perp T_2$	$prec_1 := prec_0$ $prec_2 := last_1$	$last_0 := last_2$
3	$T_0 \rightarrow V_1$		$last_0 := \text{if } (prec_0 + 1 + length_1) \leq W$ then $(prec_0 + 1 + length_1)$ else $length_1$ end if
4	$V_0 \rightarrow cV_1$		$length_0 := length_1 + 1$
5	$V_0 \rightarrow c$		$length_0 := 1$

Attribute computation is expressed by the rules of the attribute grammar in Table 5.4. Two remarks on the syntax: first, the subscripts of nonterminal symbols are added as reference for 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 $T \rightarrow T \perp T$, which is bilaterally recursive. 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.

In order to choose a feasible attribute evaluation schedule, let us examine the dependencies between the assignment statements for a specific syntax tree. 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 $last(2) \rightarrow prec(4)$: it represents a dependence of the latter attribute from the former, induced by function $prec_2 := last_1$, which is supported by production rule 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.

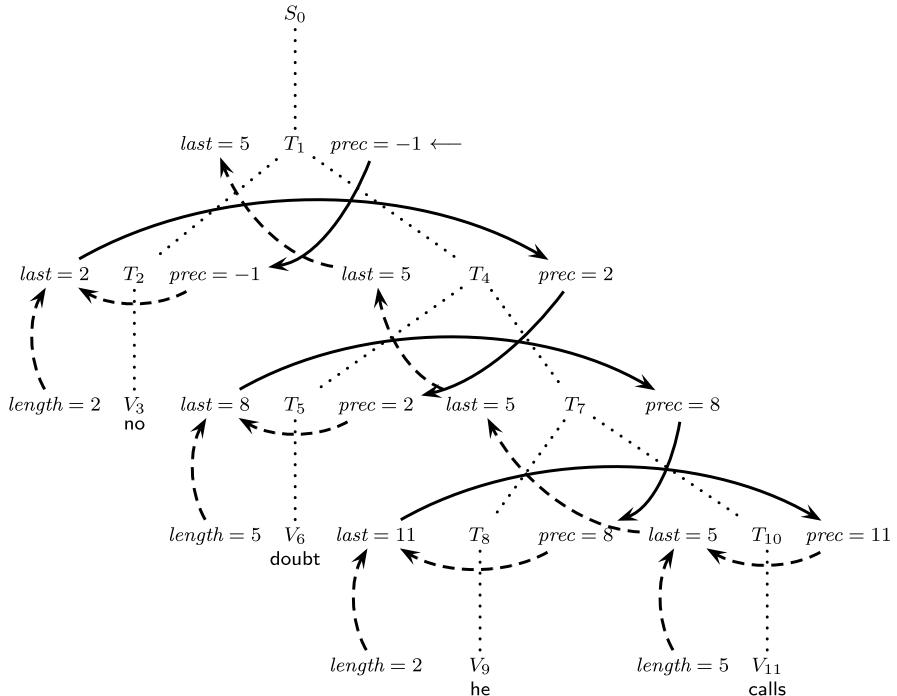


Fig. 5.14 Decorated tree with dependence graph for Example 5.34

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 This grammar 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 first example of attribute grammar)? And then, is it convenient to do so?

The first answer is yes, since we show that the position of the last letter of a word can be computed by a different approach. Initially compute the left attribute $length$, then construct a new left attribute $list$, having as domain a list of integers representing word lengths. 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 subplot 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 required at the root of the tree has essentially the same organization and complexity as the original text segmentation 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 previous grammar.

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 5.35).

Definition 5.35 (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:

$$p : D_0 \rightarrow D_1 D_2 \dots D_r \quad r \geq 0$$

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

$$\sigma_k := f(attr(\{D_0, D_1, \dots, D_r\}) \setminus \{\sigma_k\})$$

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 $\sigma_0 := f(\dots)$ 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 $\sigma_k := f(\dots)$ 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.²³

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 5.36 (Example 5.34, p. 338, continued) We refer again to the grammar of Example 5.34 on p. 338, 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 $prec_0$, $last_0$, and $last_2$ (attribute $length$ is not pertinent to production 2)

Then we have

$$attr(T) = \{prec, last\} \quad attr(V) = \{length\} \quad attr(S) = \emptyset$$

²³In 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.

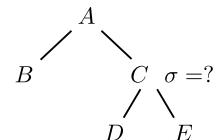
A caution against misuse of not local attributes. Item 4 expresses a sort of *principle of locality* of semantic functions: it is an error to designate as argument or result of a semantic function supported by p , an attribute which is not pertinent to production p . An instance of such error occurs in the modified rule (2):

#	Syntax	Semantic functions
1	$S_0 \rightarrow T_1$...
2	$T_0 \rightarrow T_1 \perp T_2$	$prec_1 := prec_0 + \underbrace{length_0}_{\text{non-pertinent attr.}}$
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 be disjoint sets is discussed next. Each attribute of a node of the syntax tree must be defined by exactly one assignment, otherwise it may 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	$A \rightarrow BC$	$\sigma_C := f_1(attr(A, B))$
2	$C \rightarrow DE$	$\sigma_C := f_2(attr(D, E))$



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

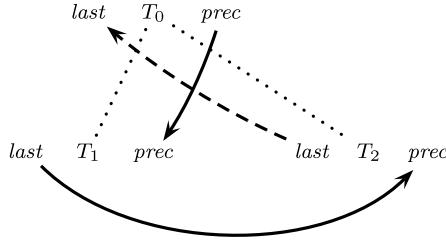
An advantage of a grammar as a specification of a translation is that it does not get involved with the details of the tree traversing procedures. In fact, the attribute evaluation program can be automatically constructed, starting from the functional dependencies between attributes, knowing of course the bodies of the semantic functions.

To prepare for that, we introduce the *dependence graph of a semantic function*: the nodes of this directed graph are the arguments and result, and there is an arc from each argument to the result. Collecting the dependence graphs for all the functions supported by the same production, we obtain the *dependence graph of a production*. The next example (Example 5.37) shows a case.

syntax support and semantic functions

#	<i>syntax</i>	<i>right attributes</i>	<i>left attributes</i>
1	$S_0 \rightarrow T_1$	$prec_1 := -1$	
2	$T_0 \rightarrow T_1 \perp T_2$	$prec_1 := prec_0$ $prec_2 := last_1$	$last_0 := last_2$
3	$T_0 \rightarrow V_1$		$last_0 := \text{if } (prec_0 + 1 + length_1) \leq W$ $\quad \text{then } (prec_0 + 1 + length_1)$ $\quad \text{else } length_1$ $\quad \text{end if}$
4	$V_0 \rightarrow c\ V_1$		$length_0 := length_1 + 1$
5	$V_0 \rightarrow c$		$length_0 := 1$

dependence graph of production rule 2



dependence graphs of the remaining productions

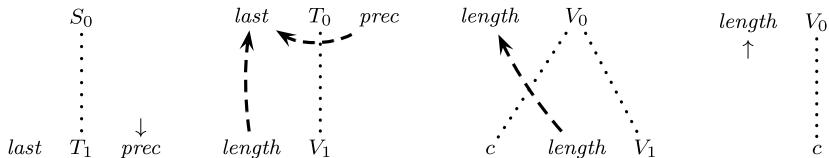


Fig. 5.15 Grammar of Example 5.34 and dependence graphs of productions

Example 5.37 (Dependence graph of productions) We reproduce from p. 339 in Fig. 5.15 the grammar of Example 5.34. 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 that a node with (respectively without) incoming arcs is an attribute of type internal (respectively external).

The *dependence graph of a (decorated) syntax tree*, already introduced, is obtained by pasting together the graphs of the individual productions used in the tree nodes. For example, look back at Fig. 5.14 on p. 340.

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 the values are computed by assignments and 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

$$\sigma_1 \rightarrow \sigma_2 \rightarrow \dots \rightarrow \sigma_{j-1} \rightarrow \sigma_j \quad \text{with } j > 1$$

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:

$$\begin{aligned}\sigma_j &= f_j(\dots, \sigma_{j-1}, \dots) \\ \sigma_{j-1} &= f_{j-1}(\dots, \sigma_{j-2}, \dots) \\ &\dots \\ \sigma_2 &= f_2(\dots, \sigma_1, \dots)\end{aligned}$$

since the result of a function is an argument of the next one.

For instance, in Fig. 5.14 on p. 340 one such path is the following:

$$prec(T_1) \rightarrow prec(T_2) \rightarrow last(T_2) \rightarrow prec(T_4) \rightarrow prec(T_5) \rightarrow \dots$$

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 tree²⁴ is acyclic. The next property (Property 5.38) states when a grammar can be considered correct.

Property 5.38 (Correct attribute grammar) Given an attribute grammar satisfying the conditions of Definition 5.35, consider a 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.

²⁴We assume the parser returns exactly one syntax tree per sentence.

To prove the property, we show that, under the acyclicity condition, the equations can be ordered in such a way that any 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 $V = \{1, 2, \dots, |V|\}$. The next algorithm (Algorithm 5.39) computes a total order of nodes, called *topological*. The result $ord[1 \dots |V|]$ is the vector of sorted nodes: it gives the identifier of the node that has been assigned to the i th position ($1 \leq i \leq |V|$) in the ordering.

Algorithm 5.39 (Topological sorting)

```

begin
     $i := 0$                                 - - initialize node counter  $i$ 
    while  $V \neq \emptyset$  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.  $i$ 
         $V := V \setminus \{n\}$                 - - remove node  $n$  from set  $V$ 
        - - remove the dangling arcs from set  $E$ 
         $E := E \setminus \{\text{arcs going out from node } n\}$ 
    end while
end
```

In general many different topological orders are possible, because the dependence graph typically does not enforce a total order relation.

Example 5.40 (Topological sorting) Applying the algorithm to the graph of Fig. 5.14 on p. 340, we obtain a topological order:

$length_3, length_6, length_9, length_{11}, prec_1, prec_2, last_2, prec_4,$
 $prec_5, last_5, prec_7, prec_8, last_8, prec_{10}, last_{10}, last_7, last_4, last_1$

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:

$$\sigma_1 \rightarrow \sigma_2 \rightarrow \cdots \rightarrow \sigma_{j-1} \rightarrow \sigma_j \quad j > 1$$

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, i.e., 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,²⁵ 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.

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 order of visit 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, \dots, N_r its children; denote by t_i the subtree rooted in node N_i .

A depth-first visit algorithm first visits the root of the tree. 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, \dots, t_r , in an order, yet not necessarily coincident with the natural one $1, 2, \dots, r$, i.e., according to some permutation of $1, 2, \dots, r$.

When a node is visited, the local attributes are computed. 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

²⁵See [8, 9]. The asymptotic time complexity is NP -complete with respect to the size of the attribute grammar.

conditions for one-sweep evaluation are simple to state and to check on the dependence graph dip_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 Grammars

For each production rule p :

$$p : D_0 \rightarrow D_1 D_2 \dots D_r \quad r \geq 0$$

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 $\{D_1, D_2, \dots, D_r\}$ of the production. The sibling graph has an arc

$$D_i \rightarrow D_j \quad \text{with } i \neq j \text{ and } i, j \geq 1$$

if in the dependence graph dip_p there is an arc $\sigma_i \rightarrow \delta_j$ 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 the nodes of the dependence graph of the production: the former are syntactical symbols, the latter are attributes. Clearly all attributes of dip_p having the same subscript j are coalesced into a 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 (Definition 5.41) states the conditions for having a one-sweep grammar.

Definition 5.41 (One-sweep grammar) A grammar satisfies the one-sweep condition if, for each production p :

$$p : D_0 \rightarrow D_1 D_2 \dots D_r \quad r \geq 0$$

having dependence graph dip_p , if the following clauses hold:

1. graph dip_p contains no circuit
2. graph dip_p does not contain a path:

$$\lambda_i \rightarrow \dots \rightarrow \rho_i \quad i \geq 1$$

- 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 dip_p contains no arc $\lambda_0 \rightarrow \rho_i$, with $i \geq 1$, from a left attribute of the father node D_0 to a right attribute of a sibling 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 $\lambda_i \rightarrow \dots \rightarrow \rho_i$, 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 children, i.e., the subtrees t_1, \dots, t_r , and to schedule their visit in an order consistent with the precedences expressed by graph dip_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 5.42 (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 :

$$p: D_0 \rightarrow D_1 D_2 \dots D_r \quad r \geq 0$$

1. Choose a topological order, denoted TOS , of the nonterminals D_1, D_2, \dots, 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 dip_p .
3. Choose a topological order, denoted TOL , of the left attributes of symbol D_0 , with respect to the dependence graph dip_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 5.43).

Example 5.43 (One-sweep semantic procedure) For brevity we consider a grammar fragment containing just one production and we leave the semantic functions unspecified. Production $D \rightarrow ABC$ has the dependence graph dip shown in Fig. 5.16.

It is straightforward to check the graph satisfies conditions (1), (2), and (3) of Definition 5.41, 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 attribute of A , B or C

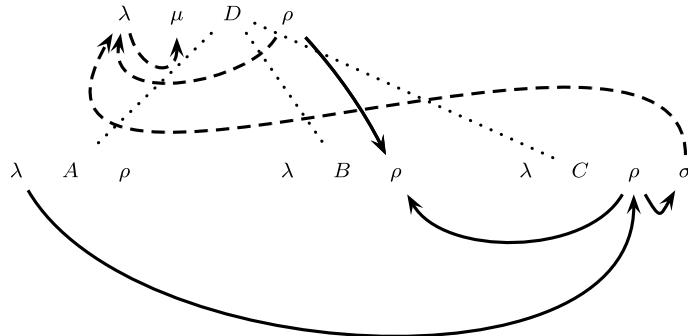
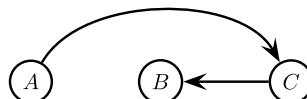


Fig. 5.16 Dependence graph of the production rule of Example 5.43

4. the sibling graph $sibl$, below, is acyclic:



We explain where its arcs come from:

- $A \rightarrow C$ from dependence $\lambda_A \rightarrow \rho_C$
- $C \rightarrow B$ from dependence $\rho_C \rightarrow \rho_B$

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 .

procedure D (**in** t, ρ_D ; **out** λ_D, μ_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 which prevent its use. More general classes of evaluators and corresponding grammar conditions are available, which we do not discuss.²⁶ 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, operating on the same syntax tree.

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 5.35 (p. 341) as well as the one-sweep condition 5.41 (p. 348). 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. After the execution of phase one, all the attributes in $Attr_1$ 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 the 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 $Attr_2$ 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

²⁶Among 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 grammar conditions is in Engelfriet [7].

one sweep. Notice that not all the attribute values computed in stage one have to be stored in the decorated tree produced by phase one, but only those used as arguments by semantic functions 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 [6]. Partitioning the 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 difficult 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, comments, etc. 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 (Example 2.27) defined by the regular expression on p. 27. A lexeme of 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 its terminal alphabet. Moreover, the 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, including for instance:

```
{begin, end, if, then, else, do, . . . , 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 (*IO*-automaton) that divides the source text into lexemes, assigning 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 into the target string each lexeme as a pair of elements: the name, i.e., the encoding of the lexical class, and a semantic attribute, termed *lexical*.

Lexical attributes change from a class to another and are altogether missing from certain classes. Some typical cases are:

- *decimal constant*: 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 *beta237* can be divided in many ways into valid lexemes: a lexeme *beta* of class *identifier* followed by 237 of class *integer*; or an identifier *beta2* followed by the integer 37, and so on.

In practice, this sort of concatenation ambiguity (p. 49) is often cured by imposing to the scanner the *longest prefix rule*. The rule tells the scanner to segment a string $x = uv$ into the lexemes, say, $u \in \text{identifier}$ and $v \in \text{integer}$, 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 *beta237* 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 *beta237* is just a string “*beta237*” (or a pointer thereto). Then the finite transducer returns the translation:

$$\langle \text{class} = \text{identifier}, \text{ss} = \text{'beta237'} \rangle$$

upon recognizing lexeme *beta237*. 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 the one-sweep algorithm (p. 348) visits in depth-first order the syntax tree, traversing the subtrees t_1, \dots, t_r , for the current production $D_0 \rightarrow D_1 \dots D_r$ 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 $1, \dots, r$.

On the other hand, we know a parser constructs the tree in the natural order, i.e., subtree t_j is constructed after subtrees t_1, \dots, 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 5.44).

Definition 5.44 (*L* condition) A grammar satisfies condition *L*²⁷ if, for each production $p: D_0 \rightarrow D_1 \dots D_r$, we have:

1. the one-sweep condition 5.41 (p. 348) is satisfied, and
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 $1, \dots, r$ is a topological sort of the sibling graph and can be applied to visit the sibling subtrees. The next property (Property 5.45) relates the *L* condition and deterministic parsing.

²⁷The letter *L* stands for left to right.

Table 5.5 Grammar for converting fractional numbers (Example 5.46)

Grammar				
Syntax	Left attributes		Right attributes	
$N_0 \rightarrow \bullet D_1$	$v_0 := v_1$		$l_1 := 1$	
$D_0 \rightarrow B_1 D_2$	$v_0 := v_1 + v_2$		$l_1 := l_0$	$l_2 := l_0 + 1$
$D_0 \rightarrow B_1$	$v_0 := v_1$		$l_1 := l_0$	
$B_0 \rightarrow 0$	$v_0 := 0$			
$B_0 \rightarrow 1$	$v_0 := 2^{-l_0}$			

Attributes				
Attribute	Meaning	Domain	Type	Assoc. symbols
v	Value	Real	Left	N, D, B
l	Length	Integer	Right	D, B

Property 5.45 (Attribute grammar and deterministic parsing) Let a grammar be such that:

- the syntax satisfies the $LL(k)$ condition, and
- 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 5.46), is a straightforward combination of a recursive descent parser and a one-sweep recursive evaluator.

Example 5.46 (Recursive descent parser with attribute evaluation) Revising Example 5.33 (p. 336), 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

$$L = \bullet(0 \mid 1)^*$$

The meaning of a string such as $\bullet01$ is decimal number 0.25. The grammar is listed in Table 5.5. Notice the value of a bit is weighted by a negative exponent, equal to its distance from the fractional point.

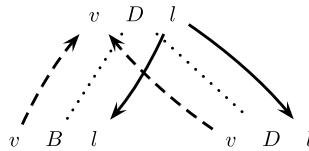
The syntax is deterministic $LL(2)$, as can be checked.

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

$D \rightarrow BD$ 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 left attribute v of father to 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. 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  $v_0$ )
  if  $cc1 = '•'$  then
    read
  else
    error
  end if
   $l_1 := 1$            -- initialize a local var. with right attribute of  $D$ 
   $D(l_1, v_0)$       -- call  $D$  to construct a subtree and compute  $v_0$ 
end procedure
  
```

```

procedure D (in  $l_0$ ; out  $v_0$ )
  case  $cc2$  of
    '0', '1' : begin
       $B(l_0, v_1)$ 
       $l_2 := l_0 + 1$ 
       $D(l_2, v_2)$ 
       $v_0 := v_1 + v_2$ 
    end
    '-' : begin
       $B(l_0, v_1)$ 
       $v_0 := v_1$ 
    end
  end
  
```

```

otherwise error
end case
end procedure

procedure B (in  $l_0$ ; out  $v_0$ )
  case cc1 of
    ‘0’:  $v_0 := 0$            -- case of rule  $B \rightarrow 0$ 
    ‘1’:  $v_0 := 2^{-l_0}$       -- case of rule  $B \rightarrow 1$ 
    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 the bottom-up syntax tree construction with attribute evaluation. Some problems have to be addressed: how to ensure that the 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 (which 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 performs a reduction, when the parser is in a macro-state containing the marked production $D_0 \rightarrow D_1 \dots D_r \bullet$. 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 right part. Assuming the values of all the attributes of $D_1 \dots D_r$ are available, the algorithm can invoke the functions and return the values of the left attributes of D_0 .

But a difficulty comes from the evaluation of the right attributes of $D_1 \dots D_r$. Imagine the algorithm is about to construct and decorate the subtree of D_1 . 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.

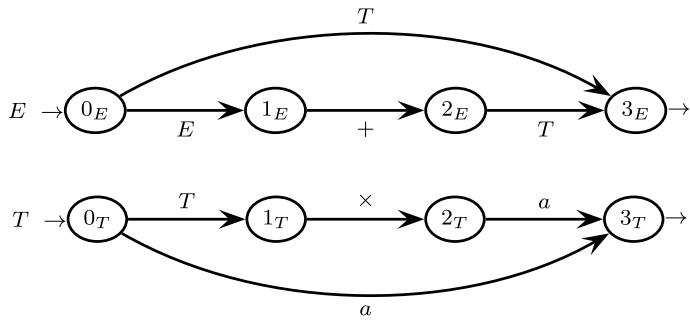


Fig. 5.17 Machine net for the arithmetic expressions (Example 5.47)

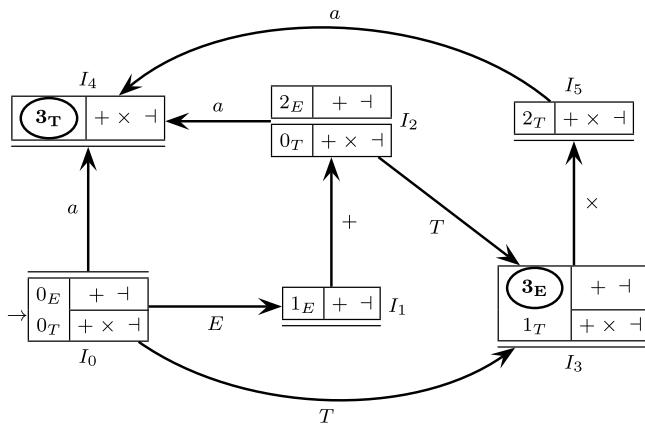


Fig. 5.18 Pilot graph of the arithmetic expressions (Example 5.47, Fig. 5.17)

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 fields and one or more semantic fields containing attribute values (or pointers to values stored elsewhere); see the next example (Example 5.47).

Example 5.47 (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 in Fig. 5.18. In the 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 the LR(1) condition.²⁸

²⁸See also Example 5.14, p. 314, where the source grammar is the same, though there the machines are drawn as trees by splitting the final states per rule, and thus the pilot (Fig. 5.6, p. 315) has more m-states; but its behavior as a recognizer is the same as here.

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
$E_0 \rightarrow E_1 + T_2$	$o_0 := o_1 \text{ or } (v_1 + v_2 > \text{maxint})$ $v_0 := \text{if } o_0 \text{ then nil else } (v_1 + v_2)$
$E_0 \rightarrow T_1$	$o_0 := o_1$ $v_0 := v_1$
$T_0 \rightarrow T_1 \times a$	$o_0 := o_1 \text{ or } (v_1 \times \text{value}(a) > \text{maxint})$ $v_0 := \text{if } o_0 \text{ then nil else } (v_1 \times \text{value}(a))$
$T_0 \rightarrow a$	$o_0 := \text{false}$ $v_0 := \text{value}(a)$

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 tree.

Right Attributes Independent of Father Actually, the 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.

Grammar expressivity improves if right attributes are readmitted, although with the following limitation on their dependencies (Definition 5.48).

Definition 5.48 (Condition A ²⁹ for bottom-up evaluation) For each production $p: D_0 \rightarrow D_1 \dots D_r$, the following must hold:

1. the L condition (p. 354) for top-down evaluation is satisfied, and
2. no right attribute ρ_{D_k} of a sibling depends on a right attribute σ_{D_0} of the father, with $1 \leq k \leq r$.

Positively stated, the same condition becomes: a right attribute ρ_{D_k} may only depend on the right or left attributes of symbols $D_1 \dots D_{k-1}$, with $1 \leq k \leq r$.

If a grammar meets the A condition, the left attributes of nonterminal symbols D_1, \dots, 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 $1, 2, \dots, r$
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.

²⁹The letter A stands for ascending order.

<i>stack</i>		<i>string</i>			
I_0	a_3		+	a_5	\dashv
I_0	a_3	I_4	+	a_5	\dashv
	T				
I_0	$v = 3$	I_3	+	a_5	\dashv
	$o = \text{false}$				
	E				
I_0	$v = 3$	I_1	+	a_5	\dashv
	$o = \text{false}$				
	E				
I_0	$v = 3$	I_1	+	I_2	a_5
	$o = \text{false}$				\dashv
	E				
I_0	$v = 3$	I_1	+	I_2	a_5
	$o = \text{false}$				I_4
	E				
I_0	$v = 3$	I_1	+	I_2	$v = 5$
	$o = \text{false}$				I_3
	E				
I_0	$v = 3 + 5$				
	$= 8$				
	$o = \text{false}$				\dashv

Fig. 5.19 Shift-reduce parsing trace with attribute evaluation (Example 5.47)

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. 348), similarly to the one-sweep evaluators.

5.6.8 Typical Applications of Attribute Grammars

Syntax-directed translation is widely applied in compiler design. Attribute grammars provide a convenient and 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 Checks

The formal language L_F defined by the syntax is just a gross approximation by excess to the actual programming (or 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 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 functionality of the semantic predicates depends 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 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 5.49) schematizes the creation of a symbol table and its use for checking the variables that appear in the assignments.

Example 5.49 (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 that the following semantic prescriptions have to be enforced:

1. a variable may not be multiply 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

Table 5.6 Grammar for checking the declaration of variables versus their use in the assignment statements (Example 5.49)

Syntax	Semantic functions	Comment
$S \rightarrow P$	$t_1 := \emptyset$	Initialize tab. to empty
$P \rightarrow DP$	$t_1 := t_0$ $t_2 := \text{insert}(t_0, n_1, \text{descr}_1)$	Propag. tab. to subtree add description to tab.
$P \rightarrow AP$	$t_1 := t_0$ $t_2 := t_0$	Propag. tab. to subtrees
$P \rightarrow \varepsilon$		No functions used here
$D \rightarrow \text{id}$	$dd_0 := \text{present}(t_0, n_{\text{id}})$ if $\neg dd_0$ then $\text{descr}_0 := \text{'sca'}$ end if $n_0 := n_{\text{id}}$	Declare scalar variable
$D \rightarrow \text{id}[\text{const}]$	$dd_0 := \text{present}(t_0, n_{\text{id}})$ if $\neg dd_0$ then $\text{descr}_0 := (\text{'vect'}, v_{\text{const}})$ end if $n_0 := n_{\text{id}}$	Declare vector variable
$A \rightarrow L := R$	$t_1 := t_0$ $t_2 := t_0$ $ai_0 := \neg \langle \text{descr}_1 \text{ is compatible with } \text{descr}_2 \rangle$	Propag. tab. to subtrees
$L \rightarrow \text{id}$	$\text{descr}_0 := \langle \text{type of } n_{\text{id}} \text{ in } t_0 \rangle$	
$L \rightarrow \text{id}[\text{id}]$	if $\left(\begin{array}{l} \langle \text{type of } n_{\text{id}_1} \text{ in } t_0 \rangle = \text{'vect'} \text{ and} \\ \langle \text{type of } n_{\text{id}_2} \text{ in } t_0 \rangle = \text{'sca'} \end{array} \right) \text{ then}$ $\text{descr}_0 := \text{'sca'}$ else error end if	Use indexed variable
$R \rightarrow \text{id}$	$\text{descr}_0 := \langle \text{type of } n_{\text{id}} \text{ in } t_0 \rangle$	Use sca./vect. variable
$R \rightarrow \text{const}$	$\text{descr}_0 := \text{'sca'}$	Use constant
$R \rightarrow \text{id}[\text{id}]$	if $\left(\begin{array}{l} \langle \text{type of } n_{\text{id}_1} \text{ in } t_0 \rangle = \text{'vect'} \text{ and} \\ \langle \text{type of } n_{\text{id}_2} \text{ in } t_0 \rangle = \text{'sca'} \end{array} \right) \text{ then}$ $\text{descr}_0 := \text{'sca'}$ else error end if	Use indexed variable

The attribute grammar is shown in Table 5.6. The syntax is in a rather 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 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	id
v	Constant value	Number	Left	const
dd	Double declaration	Boolean	Left	D
ai	Left/right parts incompatible	Boolean	Left	D
$descr$	Variable descriptor	Record	Left	D, L, R
t	Symbol table	Array of records	Right	A, P

The semantic analyzer processes a declaration D , and if the declared variable is already present in the symbol table, then it sets predicate dd to true. Otherwise the variable descriptor is constructed and passed to the father (or parent) 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, then 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.

For instance, in the syntactically correct text

$$\overbrace{a[10]}^{D_1} \quad \overbrace{i}^{D_2} \quad \overbrace{b}^{D_3} \quad \overbrace{i := 4}^{A_4} \quad \overbrace{c := a[i]}^{A_5: ai=true} \quad \overbrace{c[30]}^{D_6} \quad \overbrace{i}^{\overbrace{D_7: dd=true}^{A_8: ai=true}} \quad \overbrace{a := c}^{A_8: ai=true}$$

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, wrong dimension, etc.

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 towards 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 expres-

sion; 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 *runtime errors* when executed. See for instance the following program fragment:

```
array a[10]; ... read (i); a[i] := ...
```

The **read** instruction may assign to variable i a value that falls out of the interval $1 \dots 10$, 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. 300) 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.³⁰ 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*.

³⁰Code 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., [3] or [4].

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 towards 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 requires a fresh label that differs from the already used labels: 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 \bullet) 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*.

In the next examples (Examples 5.50 and 5.51), we, respectively, illustrate these issues with conditional and iterative instructions.

Example 5.50 (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 (<i>a</i> > <i>b</i>) then <i>a</i> := <i>a</i> - 1 else <i>a</i> := <i>b</i> end if ...	<i>tr(a > b)</i> jump-if-false <i>rc</i> , <i>e_7</i> <i>tr(a := a - 1)</i> jump <i>f_7</i> <i>e_7:</i> <i>tr(a := b)</i> <i>f_7:</i> ...
	-- rest of the program

Remember that *tr(...)* 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 (Example 5.51) we show the translation of a loop with initial condition.

Table 5.7 Grammar for translating conditional *if then else* instructions to conditional jumps with labels (Example 5.50)

Syntax	Semantic functions
$F \rightarrow I$	$n_1 := \text{fresh}$
$I \rightarrow \text{if } (\text{cond})$	$tr_0 := tr_{\text{cond}} \bullet$
then	jump-if-false rc_{cond} , $e_n_0 \bullet$
L_1	$tr_{L_1} \bullet$ jump $f_n_0 \bullet$
else	$e_n_0 : \bullet$
L_2	$tr_{L_2} \bullet$
end if	$f_n_0 :$

Table 5.8 Grammar for translating iterative *while do* instructions to conditional jumps with labels (Example 5.51)

Syntax	Semantic functions
$F \rightarrow W$	$n_1 := \text{fresh}$
$W \rightarrow \text{while } (\text{cond})$	$tr_0 := i_n_0 : \bullet tr_{\text{cond}} \bullet$
do	jump-if-false rc_{cond} , $f_n_0 \bullet$
L	$tr_L \bullet$ jump $i_n_0 \bullet$
end while	$f_n_0 :$

Example 5.51 (Iterative instruction) The grammar for a *while do* statement, shown in Table 5.8, is quite similar to that of Example 5.50 and does not need further comments.

It suffices to display the translation of a program fragment (assuming 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, loops with final conditions, etc., 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.³¹ 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. But 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 the

³¹The optimizer is by far the most complex and expensive part of a modern compiler; the reader is referred to, e.g., [3, 4, 10].

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., that 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. 227) 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. 351. The whole set of semantic attributes is divided in 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. 354). 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 $D_0 \rightarrow D_1 \dots D_i \dots D_r$, 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 $D_1 \dots D_{i-1}$ is then available.

Following condition *L*, 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 (Example 5.52) illustrates the use of guide predicates.

Example 5.52 (A language without punctuation marks) The syntax of the historical Pascal-like language *PLZ-SYS*³² 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.

³²Designed in the 70's for an 8-bit microprocessor with minimal memory resources.

Procedure P contains five identifiers in its parameter list, which can be interpreted in three ways:

$$P \text{ proc } (XYT1ZT2) \quad \left\{ \begin{array}{l} 1. X \text{ has type } Y \text{ and } T1, Z \text{ have type } T2 \\ 2. X, Y \text{ have type } T1 \text{ and } Z \text{ has type } T2 \\ 3. X, Y, T1, Z \text{ have type } T2 \end{array} \right.$$

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:

type $T1 = \text{record} \dots \text{end}$ **type $T2 = \text{record} \dots \text{end}$**

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 distributed towards the subsequent parts of the program, and in particular to the procedure heading declarations. For downward and rightward propagation, the left attribute t is (conceptually) 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; and 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 the var_id nonterminals expand to a generic identifier id , both alternative rules of nonterminal V start with the string id id , because:

$$\begin{aligned} V &\rightarrow var_id \ V \quad \text{-- starts with } var_id \text{ followed by } var_id \\ V &\rightarrow var_id \quad \text{-- starts with } var_id \text{ followed by } type_id \end{aligned}$$

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 5.52)

Syntax	Semantic functions
-- declarative part	-- symbol table is copied from T to I
$D \rightarrow T I$	$td_I := t_T$
-- type declaration	-- descriptor is inserted in table
$T \rightarrow \text{type id} = \text{record} \dots \text{end } T$	$t_0 := \text{insert}(t_2, n_{\text{id}}, \text{'type'})$
$T \rightarrow \varepsilon$	$t_0 := \emptyset$
-- procedure heading	-- table is passed to L and to $I \equiv 3$
$I \rightarrow \text{id proc } (L) I$	$td_L := td_0 \quad td_3 := td_0$
$I \rightarrow \varepsilon$	
-- parameter list	-- table is passed to V and to $L \equiv 3$
$L \rightarrow V \text{ type_id } L$	$td_V := td_0 \quad td_3 := td_0$
$L \rightarrow \varepsilon$	
-- variable list (of same type)	-- table is passed to var_id and to $V \equiv 2$
$V \rightarrow var_id V$	$td_1 := td_0 \quad td_2 := td_0$
$V \rightarrow var_id$	$td_1 := td_0$
$type_id \rightarrow id$	
$var_id \rightarrow id$	

Let cc_1 and cc_2 , respectively, be the current terminal character (or rather lexeme) and the next one. The guide predicates for each alternative rule are listed here:

#	Production rule	Guide predicate
1	$V \rightarrow var_id V$	$\langle \text{the descr. of } cc_2 \text{ in table } td_0 \rangle \neq \text{'type'}$ and
1'		$\langle \text{the descr. of } cc_1 \text{ in table } td_0 \rangle \neq \text{'type'}$
2	$V \rightarrow var_id$	$\langle \text{the descr. of } cc_2 \text{ in table } td_0 \rangle = \text{'type'}$ and
2'		$\langle \text{the descr. of } cc_1 \text{ in table } td_0 \rangle \neq \text{'type'}$

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, we can add a semantic predicate to production rule $L \rightarrow V \text{ type_id } L$, in order to check whether the sort of $type_id \equiv cc_2$ in the table is equal to “type”.

In this manner the top-down parser gets help from the values of the available semantic attributes and 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 compilers translating a programming language, and also by many software engineering tools.

Imagine 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 a particular 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 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 that 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 of a write instruction, we say the statement *uses* (or makes reference to) that variable
 - thus a node representing a statement p , in the graph is associated with two sets: the set $\text{def}(p)$ of defined variables and the set $\text{use}(p)$ of used variables

Notice 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:

$$\text{def}(p) = \{a\} \quad \text{use}(p) = \{a, b\}$$

In this abstract model the statements **read**(a) and $a := 7$ are undistinguishable, as they carry the same associate information: $\text{def} = \{a\}$ and $\text{use} = \emptyset$.

In order to clarify the concepts and to describe some applications of the method, we present a more complete example (Example 5.53).

Example 5.53 (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 $\text{use}(1)$ and $\text{def}(5)$ 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 $\langle \text{label}, \text{definedvariables}, \text{usedvariables} \rangle$ such

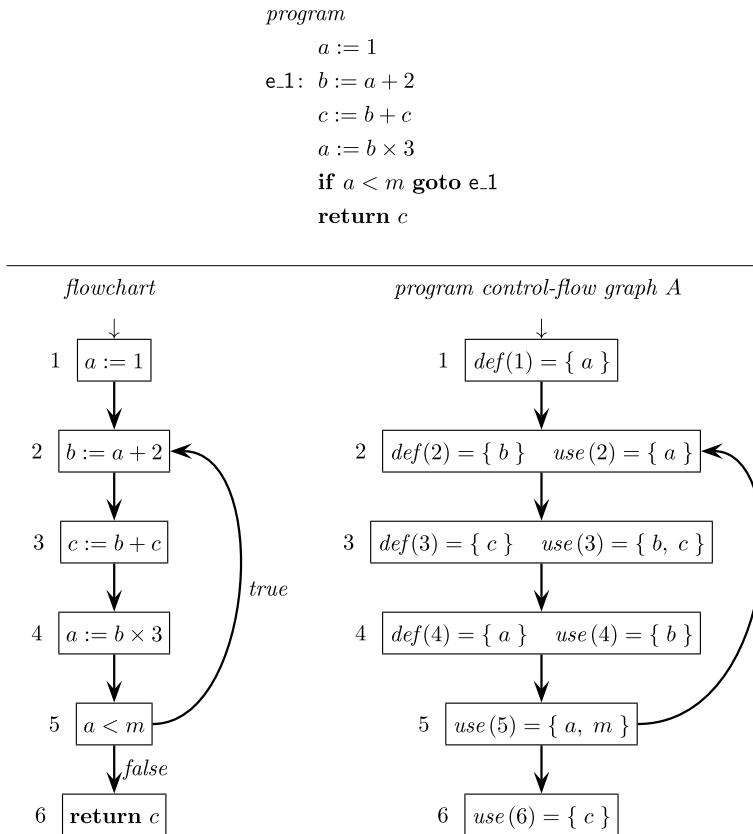


Fig. 5.20 Program, flowchart and abstract control-flow graph of Example 5.53

as

$$\langle 2, \text{def}(2) = \{b\}, \text{use}(2) = \{a\} \rangle$$

For the sake of brevity, we often denote such instruction by the first component, 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 terminal characters are not written on the arcs but inside the nodes, as we did with the local automata studied on p. 122. Clearly all the arcs entering the same node “read” the same character. The states of the automaton (as in the syntax charts on p. 169) are not given an explicit name, since they are anyhow identified by the instruction label. The initial state is marked by an entering arrow (dart). The final states are those without successor.

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.

Clearly 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, 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:

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \equiv 1234523456$$

The set of such recognized paths is the language $L(A) = 1(2345)^+6$.

Conservative Approximations 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: if a * *2 ≥ 0 then 2: istr2 else 3: istr3
```

where the formal language accepted by the automaton contains two paths {12, 13}, but path 13 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.

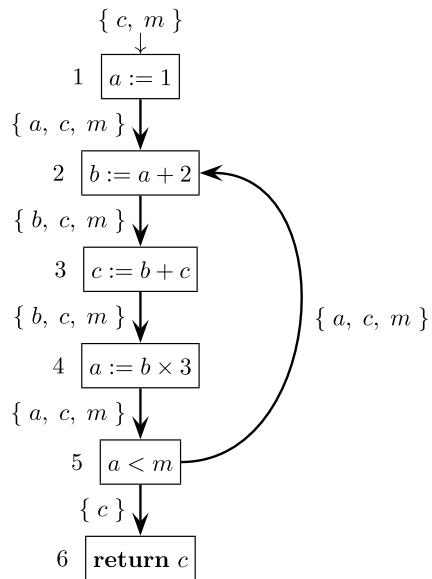
In conclusion, the decision to examine all the recognized paths (from the initial node to a final one) is a *conservative approximation* to program analysis, which may cause the diagnosis of nonexisting errors or the prudential assignment of unnecessary resources, but it never misses real error conditions or real resource requirements.

An usual hypothesis in static analysis is that the automaton is clean (p. 99), 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 Variables

A professional compiler performs several analysis passes over the intermediate representations of a program in order to improve it. A very interesting analysis, which

Fig. 5.21 Control-flow graph of the program with the sets of variables live out of the nodes (Example 5.53)



allows a variety of profitable optimizations, is the study of the liveness intervals of program variables (Definition 5.54).

Definition 5.54 (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 $a \in \text{def}(r)$, and
- instruction q uses variable a , i.e., $a \in \text{use}(q)$

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 5.55) illustrates the situation.

Example 5.55 (Example 5.53 continued) For the example of Fig. 5.20 on p. 372, 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 123 such that $c \in \text{use}(3)$, 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) 12 and 452; it is not live in the intervals 234 and 56, and so on.

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, 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, p. 372) it is $D(b) = \{2\}$ and $U(b) = \{3, 4\}$. The liveness condition will be first expressed in terms of formal language operations, 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 $x = upvqw$, 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: s

$$u, w \in I^* \wedge p \in I \wedge v \in (I \setminus D(a))^* \wedge q \in U(a) \quad (5.1)$$

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 (5.1), denoted as L_p , is a sublanguage of $L(A)$, i.e., $L_p \subseteq L(A)$. Moreover, language L_p is regular, because it can be defined by the following intersection:

$$L_p = L(A) \cap R_p \quad (5.2)$$

where language R_p is regular and is defined by the extended (i.e., with a set difference) regular expression R_p ,

$$R_p = I^* p (I \setminus D(a))^* U(a) I^* \quad (5.3)$$

Formulas (5.2) and (5.3) 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 (5.2), as explained in Chap. 3 on p. 134. 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 $\text{live}_{\text{out}}(p)$ and those live-in $\text{live}_{\text{in}}(p)$. A second equation expresses the relation between the variables live-out of a node and those live-in for its successors.

We denote by $\text{succ}(p)$ the set of the (immediate) successors of node p , and by $\text{var}(A)$ the set of all the variables of program A .

Data-Flow Equations For each final (exit) node p :

$$\text{live}_{\text{out}}(p) = \emptyset \quad (5.4)$$

For any other node p :

$$\text{live}_{\text{in}}(p) = \text{use}(p) \cup (\text{live}_{\text{out}}(p) \setminus \text{def}(p)) \quad (5.5)$$

$$\text{live}_{\text{out}}(p) = \bigcup_{q \in \text{succ}(p)} \text{live}_{\text{in}}(q) \quad (5.6)$$

Comments:

- In Eq. (5.4) no variable is live out 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. (5.5) 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. 372). 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; 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. (5.6), 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. (5.5) and (5.6) 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., $\text{live}_{\text{in}}(p)$ and $\text{live}_{\text{out}}(p)$ 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:

$$\forall p \in I : \quad \text{live}_{\text{in}}(p) = \emptyset \quad \text{live}_{\text{out}}(p) = \emptyset$$

Let $i \geq 0$ be the current iteration. In each equation of the system (5.5), (5.6), we replace the unknowns occurring in the right hand sides with the values of the current iteration, and thus we obtain the values of the 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 the one of the preceding iteration.

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 $\text{live}_{\text{in}}(p)$ and $\text{live}_{\text{out}}(p)$ is bounded by the number of program variables
- 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 (Example 5.56) on the running example.

Example 5.56 (Running example (Example 5.53) 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. 372) are written in Table 5.10. The names of the unknowns are shortened to $\text{in}(p)$ and $\text{out}(p)$ instead of $\text{live}_{\text{in}}(p)$ and $\text{live}_{\text{out}}(p)$, 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 5.56)

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.³³ 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.

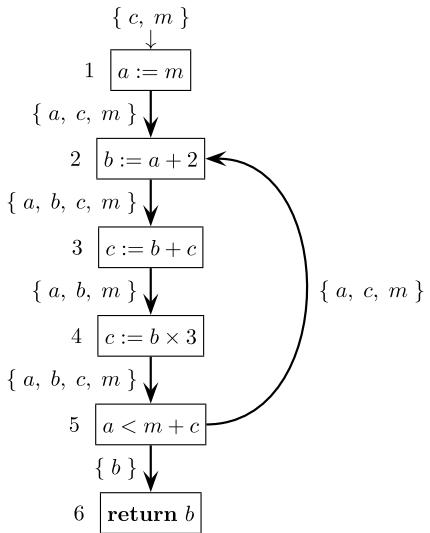
Memory Allocation Liveness analysis is best applied to decide if two variables can reside in the same memory cell (or in the same machine register). It is evident that if two variables are live in the same program point, both values must be present in the memory when execution reaches that point, because 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.

Example 5.57 (Interference and register assignment) In the control-flow graph of Fig. 5.21 on p. 374, 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) ,

³³For a proof refer for instance to any of [3, 4, 10].

Fig. 5.22 Control-flow graph with live sets applied to detect the useless definitions in the program (Example 5.58)



(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 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 Definitions 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. 372 does not have any useless definitions, in contrast with the next example (Example 5.58).

Example 5.58 (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 Definitions

Another basic and widely applied type of static analysis is the search for the 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 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 $p : a := b \oplus c$ 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 (Definition 5.59) formalizes the concept.

Definition 5.59 (Reaching definition) We say that the *definition* 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 .

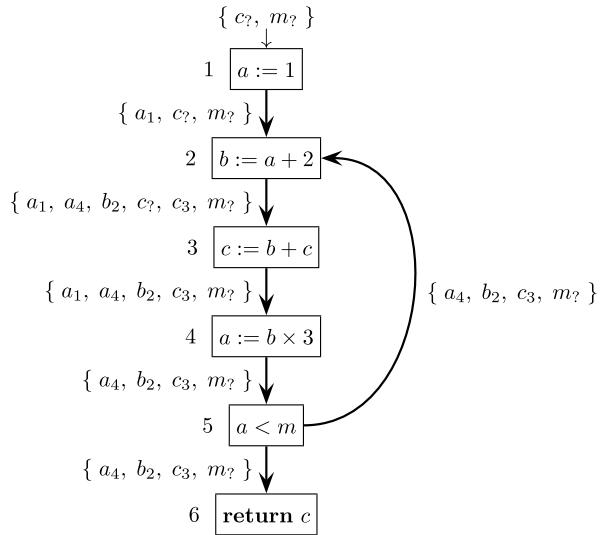
When this happens, instruction p may use the value of variable a computed in the instruction q .

Referring to automaton A , i.e., to the control-flow graph of the subprogram, the condition above (Definition 5.59) can be restated more precisely as follows. Definition a_q reaches instruction p , if language $L(A)$ contains a sentence of the form $x = uqvwpw$, 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:

$$u, w \in I^* \wedge q \in D(a) \wedge v \in (I \setminus D(a))^* \wedge p \in I \quad (5.7)$$

Notice that the instructions p and q may coincide.

Fig. 5.23 Control-flow graph with reaching definitions (Example 5.60)



Looking again at the program reproduced in Fig. 5.23 (identical to the one on p. 372), 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 Definitions 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:

$$\begin{cases} sup(p) = \{a_q \mid q \in I \wedge q \neq p \wedge \\ \quad a \in def(q) \wedge a \in def(p)\} & \text{if } def(p) \neq \emptyset \\ sup(p) = \emptyset & \text{if } def(p) = \emptyset \end{cases}$$

Notice the set $def(p)$ may contain more than one name, in the case of the instructions that define 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:

$$in(1) = \emptyset \tag{5.8}$$

Table 5.11 Data-flow equations for the reaching definitions in the program (Example 5.60 in Fig. 5.23)

$in(1) = \{c?, m?\}$
$out(1) = \{a_1\} \cup (in(1) \setminus \{a_4\})$
$in(2) = out(1) \cup out(5)$
$out(2) = \{b_2\} \cup (in(2) \setminus \emptyset) = \{b_2\} \cup in(2)$
$in(3) = out(2)$
$out(3) = \{c_3\} \cup (in(3) \setminus \{c_7\})$
$in(4) = out(3)$
$out(4) = \{a_4\} \cup (in(4) \setminus \{a_1\})$
$in(5) = out(4)$
$out(5) = \emptyset \cup (in(5) \setminus \emptyset) = in(5)$
$in(6) = out(5)$
$out(6) = \emptyset \cup (in(6) \setminus \emptyset) = in(6)$

For any other node $p \in I$:

$$out(p) = def(p) \cup (in(p) \setminus sup(p)) \quad (5.9)$$

$$in(p) = \bigcup_{q \in pred(p)} out(q) \quad (5.10)$$

Comments:

- Equation (5.8) 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.
- Equation (5.9) 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 (5.10) 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 5.60) with a program containing a loop.

Example 5.60 (Reaching definitions) Observe in Fig. 5.23 the same program control-flow graph of p. 372, 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.

We list the constant terms occurring in the equations:

Node	Instruction	<i>def</i>	<i>sup</i>
1	$a := 1$	a_1	a_4
2	$b := a + 2$	b_2	\emptyset
3	$c := b + c$	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 5.60 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 legal 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 , and
2. no other definition a_r of variable a reaches the entrance of p , with $r \neq q$

In the next example (Example 5.61), we show some program improvements produced by a constant propagation and by the induced simplifications.

Example 5.61 (Optimization following constant propagation) Figure 5.24 shows a simple program control-flow graph, and lists the reaching definition sets and the live 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 afterwards 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 expression³⁴ $8 \times 4 = 32 \geq 0 = \text{true}$, thus determining which one of the successor legs of statement 2 will be taken, say the one to the left. Then

³⁴Anticipating a computation to compile time is termed *constant folding*.

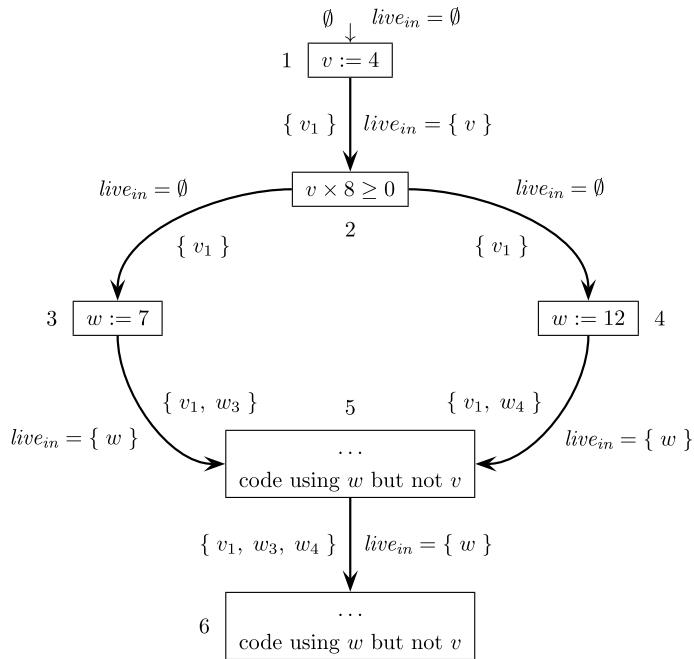
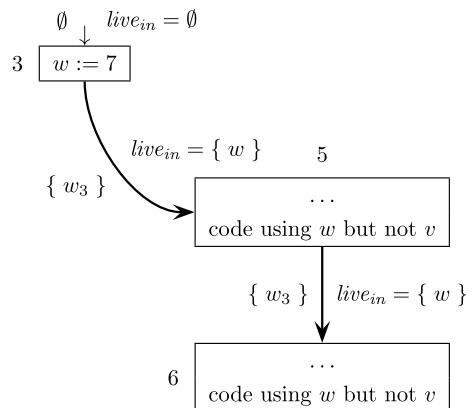


Fig. 5.24 A control-flow graph with reaching definitions and live variables before constant propagation (Example 5.61)

Fig. 5.25 The program control-flow graph of Fig. 5.24 after the optimizations induced by constant propagation



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 legally 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 compile-time error occurs.

Coming back to the program control-flow graph of Fig. 5.23 on p. 381, observe that node 3 uses variable c , but on the path 123 no instruction executed before node 3 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 5.62). For simplicity we assume that the subprogram does not have any input parameters.

Definition 5.62 (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 .

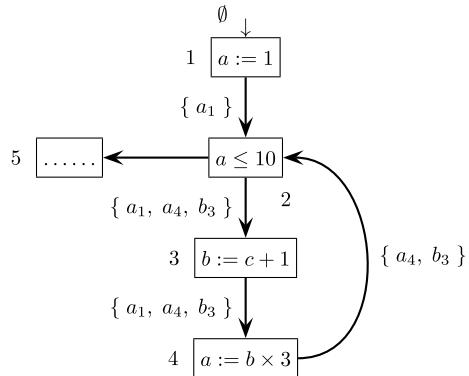
By comparing this notion with the concept of reaching definition introduced on p. 380, we notice a difference in the quantification over paths. If the definition a_q of variable a reaches the entrance of node p , there necessarily exists a path from node 1 to node p that traverses the defining instruction 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

Fig. 5.26 Control-flow graph with the available variables for Example 5.63



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 Variables An instruction p is *not well initialized* if the following predicate holds:

$$\exists q \in pred(p) \text{ such that } use(p) \not\subseteq out'(q) \quad (5.11)$$

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, 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 5.63) illustrates the situation.

Example 5.63 (Detecting uninitialized variables) Observe the program control-flow graph in Fig. 5.26, completed with the sets of reaching definitions.

Condition (5.11) is false in the node 1, since for every predecessor (nodes 1 and 4) the set out' contains a definition of a , which is the only variable used in 1. On the other hand, the condition is true in the node 3 because no definition of variable c reaches the exit from 3. 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 (5.11) 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 runtime errors or raise exceptions during program execution.

To conclude, static analysis³⁵ 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. A. Aho, J. Ullman, *The Theory of Parsing, Translation, and Compiling, vol. 1: Parsing* (Prentice-Hall, Englewood Cliffs, 1972)
2. A. Aho, J. Ullman, *The Theory of Parsing, Translation and Compiling, vol. 2: Compiling* (Prentice-Hall, Englewood Cliffs, 1973)
3. A. Aho, M. Lam, R. Sethi, J. Ullman, *Compilers: Principles, Techniques and Tools* (Prentice-Hall, Englewood Cliffs, 2006)
4. A. Appel, *Modern Compiler Implementation in Java* (Cambridge University Press, Cambridge, 2002)
5. J. Berstel, *Transductions and Context-Free Languages* (Teubner, Stuttgart, 1979)
6. S. Crespi Reghizzi, G. Psaila, Grammar partitioning and modular deterministic parsing. *Comput. Lang.* **24**(4), 197–227 (1998)
7. J. Engelfriet, Attribute grammars: attribute evaluation methods, in *Methods and Tools for Compiler Construction*, ed. by B. Lorho (Cambridge University Press, Cambridge, 1984), pp. 103–138
8. D.E. Knuth, Semantics of context-free languages. *Math. Syst. Theory* **2**(2), 127–145 (1968)
9. D.E. Knuth, Semantics of context-free languages (errata corrigere). *Math. Syst. Theory* **5**(2), 95–99 (1971)
10. S. Muchnick, *Advanced Compiler Design and Implementation* (Morgan Kaufmann, San Mateo, 1997)
11. F. Nielson, H. Nielson, C. Hankin, *Principles of Program Analysis* (Springer, New York, 2005)
12. J. Sakarovitch, *Elements of Automata Theory* (Cambridge University Press, Cambridge, 2009)
13. R.D. Tennent, *Semantics of Programming Languages* (Prentice-Hall, Englewood Cliffs, 1991)
14. G. Winskel, *The Formal Semantics of Programming Languages* (MIT Press, Cambridge, 1993)
15. W. Yang, Mealy machines are a better model of lexical analyzers. *Comput. Lang.* **22**, 27–38 (1996)

³⁵A book on the theory of static program analysis is [11]. For a survey of applications in compilation, see, e.g., [3, 4, 10].

Index

Symbols

2I-automaton, 323, 324
 ε move, 105

A

A-condition for attribute evaluation, 359
Abstract syntax tree, 303, 335
Acceptance
 by empty stack, 144
 with final state, 148
Accepting mode of pushdown machine, 149
Accessible
 state, 99, 177
 subsets, 115
Acyclic attribute grammar, 345, 347
Acyclicity test, 347
Algol 60, 52
Algorithm
 Berry and Sethi, 129
 bottom-up parser construction, *see ELR(1)*
 parser construction
 composition of local automata, 124
 conversion of translation grammar to
 postfix form, 316
 deterministic automaton from a regular
 expression construction, 127
 determinization of finite automaton, 131
 Earley parser, 253
 $ELR(1)$ parser construction, 190
 pointerless, 222
 with vector stack, 202
 finite automaton complementation, 132
 finite automaton determinization, *see*
 Powerset construction
 from grammar to nondeterministic
 pushdown automaton, 143
 from regular expression to deterministic
 finite automaton, *see* Berry and
 Sethi algorithm
 from regular expression to grammar, 67
 local recognizer construction, 122
merge of kernel-equivalent m-states, 219
one-sweep evaluator construction, 349
operator precedence parallel parser
 construction, 281
operator precedence sequential parser
 construction, 275
pilot graph construction, 183
powerset construction, 117
predictive parser construction
 as DPDA, 237
 by recursive procedures, 238
predictive pushdown transducer
 construction, 308
spontaneous move elimination, 115
syntax tree construction, 262
top-down parser construction, *see*
 Predictive parser construction
topological sorting, 346
Alphabet, 7
 unary, 75
Alphabetic homomorphism, 79, 298
Alternative, 18, 31
Ambiguity, 45
 conditional instruction, 52, 301, 305
 degree, 46
 EBNF, 85
 inherent, 53, 119, 160
 of automaton, 111, 160
 of bilateral recursion, 47
 of circular derivation, 304
 of concatenation, 49
 of regular expression, 21, 68
 of translation, 304
 of union, 48
 source grammar, 305
 versus nondeterminism, 160
 $ANTLR$, 242, 248
Aperiodic language, 136
Arden identity, 72
Arithmetic expression, 38, 39, 45, 48, 54, 55,
 59, 85, 311

- Arithmetic expression (*cont.*)
 calculating machine, 358
 machine net, 167
 parenthesis-free, 70
 polish, 302
 syntax chart, 169
- Artificial language, 5
- Attribute, 334, 336
 inherited, *see* Right attribute
 left, *see* Left attribute
 lexical, *see* Lexical attribute
 right, *see* Right attribute
 synthesized, *see* Left attribute
- Attribute grammar, 335, 360
 A-condition, 359
 acyclic, 345
 applications, 360
 definition, 341
 L condition, 354
 multi-sweep, 351, 367
 one-sweep, 348
- Attribute subgrammar, 351
- Automaton
 ambiguous, 111, 160
 clean, 99
 configuration, 94
 equivalent, 95
 finite, 96
 finite deterministic, 97
 for local language, 122
 from regular expression, 119, 121
 generalized, 113
 local, 122
 normalized, 123
 minimization, 101
 nondeterministic, 107
 product, 134
 to regular expression, 112
 two-way, 331
 unidirectional, 94
 with spontaneous moves, 109
- Available variables, 385
- Axiom, 30, 31
- B**
- Back-end compiler, 365
- Base of macro-state, 183
- Berry, 129
- Berry and Sethi, 129
 algorithm, 130
- Berstel, 121, 295
- Binary operator, 301
- Block structure, 29
- BMC, 112
- BNF grammar, 31, 206
- Bottom-up
 attribute evaluation, 359
 deterministic translation, 313
 parser with attribute evaluator, 357
 syntax analysis, 163
- Brzozowski and McCluskey, 112
- BS, *see* Berry and Sethi
- BT, 261
- BuildTree, 261
- C**
- Cancellation rule of Dyck, 42
- Candidate, 174
 identifier, 190
- Cardinality of language, 8
- Cartesian product of automata, 134
- CF family, 68, 74, 78, 81, 83, 151
 closure properties, 44
- Choice of a regular expression, 19
- Chomsky
 hierarchy, 33, 86
 normal form, 33, 61
- Circular derivation, 36, 304
- Clean automaton, 99, 373
- Cleaning
 of automaton, 99
 of grammar, 35
- Cleveland, 89
- Closing mark, 25
- Closure
 function, 174
 in Earley parser, 250
 of macro-state, 183
- Closure properties
 of CF, 44, 78
 of CF and REG, 78
 of DET, 155
 of REG, 23
 under substitution, 81
 under translation, 332
 under transliteration, 81
- Code generation, 364
- Codes and ambiguity, 50
- Coke–Younger–Kasami algorithm, 247
- Compaction of pilot, 218
- Compiler
 decomposition, 351
 language, 334
- Complement
 automaton, 132
 of language, 13
 of regular language, 132
- Completion in Earley parser, 251

- Computation, 94, 107, 147
label, 98, 107
length, 107
of nondeterministic automaton, 107
- Concatenation, 9
of context-free languages, 44, 78
of Dyck languages, 50
of languages, 12
- Condensed skeleton tree, 40
- Conditional instruction, 52, 301, 305
ambiguity, 52, 301, 305
- Configuration of an automaton, 94, 143, 147
- Conflict
convergence, 182
reduce-reduce, 182
shift-reduce, 182
- Conservative approximation, 373
- Constant
folding, 383
propagation, 380, 383
- Context-dependent language, 86
- Context-free
deterministic, 155
grammar
definition, 31
from pushdown automaton, 151
introduction, 29
translation, 299
- Context-sensitive language, 86
- Control instruction translation, 365
- Control-flow graph, 370, 371
automaton, 371
language, 371
- Convergence conflict, 182, 184, 186
- Convergent transitions, 184
- Copy rule, 57
- Copy-free normal form, 57
- Cross operation, 16
- D**
- Dart, 232
- Data-flow equations, 376, 381
liveness, 376
reaching definitions, 381
solution, 376
- Dead code, 384
- Decidable language, 95
- Decimal constant, 96, 99, 109, 111
- Decision algorithm, 92
- Decompiler, 297
- Decorated tree, 337, 339
- Degree of operator, 301
- Dependence graph, 339, 343, 348
of decorated tree, 339, 345
- of semantic function, 343
- Derivation, 33
circular, 36
EBNF, 84
left, 41
of regular expression, 20
right, 41
self-nested, 74
- DET* family, 155, 244
- Deterministic
finite automaton, 97
language, 155
unambiguity, 159
language families comparison, 242
pushdown automaton, 155
subclasses, 160
simple grammar, 161
- Determinization of automaton, 114, 130
- Dictionary, 8
- Difference of languages, 13
- Digrams, 121
- Distance of strings, 286
- Distinctly parenthesized grammar, 43, 162
- Distinguishability of states, 101
- Document type definition, 162
- Double service lemma, 158
- DTD*, 162
- Dyck
cancellation rule, 42
language, 42, 69, 79
concatenation, 50
- Dynamic error, 364
- E**
- Earley
algorithm, 248, 252
closure, 250
introductory example, 249
nonterminal shift, 251
parser, 252
completeness, 257
completion algorithm, 252
computational complexity, 259
correctness, 257
function *BuildTree*, 262
grammar ambiguity, 268
grammar unambiguity, 260
nullable nonterminals, 265
optimization, 260
syntax analysis algorithm, 253
syntax tree construction, 261
syntax tree construction complexity, 266
terminal shift algorithm, 253

- Earley (*cont.*)
 vector, 252
 terminal shift, 250
 vector, 252
- Early scanning, 207
- EBNF* grammar, 82, 165
 ambiguity, 85
 derivation, 84
 translation, 313
- Editing distance, 286
- ELL(1)*
 condition, 214
 direct, 233
 violation, 236
- parser
 direct construction, 231
 step by step construction, 217
- parsing, 211
- PCFG*, 226
 violation remedy, 246
- ELL(k)*
 condition, 240
 parser, 242
- ELR(1)*
 condition, 185
 grammar, 245
 language, 245
 parser
 pointerless, 222
 vector-stack, 201
 parser construction, 183
- ELR(k)*
 grammar, 245
 language, 245
 relation with *ELR(1)*, 245
- Empty string, 9
- Encryption, 298
- End mark, 305
- Engelfriet, 351
- Environment, 361
- Epsilon move, 105
- Equations of unilinear grammar, 71, 72
- Equivalence
 of finite automata, 98, 103
 of grammars
 generalized structural, 56
 strong, 55
 weak, 55
- Equivalent regular expressions, 21
- Error
 dynamic, 284
 objective, 284
 recovery, 286, 287
- panic mode, 287
 panic mode with token insertion, 288
- semantic, 285
- state, 98
- static, 284
- subjective, 284
- syntactic, 285
- treatment, 284
- type, 284
- Expansion of nonterminal, 56
- Extended regular expression, 132, 135
- External attribute, 342
- F**
- FIN* family, 19
- Finals, 121
- Finite
 automaton, 96
 deterministic, 97
 left-linear grammar, 112
 transducer, 328, 329
 opposite passes, 331
 with look-ahead, 332
- Finite-state family, 109
- Flex, 332
- Floyd, 158
 Operator Precedence languages, 274
- Follow set, *see* Set of followers
- Followers, *see* Set of followers
- Formal language, 5
- Free monoid, 14
- Front-end compiler, 364
- G**
- General
 automaton, 93
 parser, 248
- Generalized automaton, 113
- Goal-oriented, *see* Predictive
- Grammar
 ambiguity, 46
 ambiguity from ambiguous r.e., 68
 attribute, 335
 BNF, 31, 206
 Chomsky classification, 86
 clean, 35
 cleaning, 35
 context-free, 29
 context-sensitive, 86
 EBNF, 82, 165
 equivalent, 35
 errors, 35
 extended context-free, 82, 165
 homogeneous, 33, 64

- Grammar (*cont.*)
- invertible, 59
 - left-linear, 69
 - linear, 68
 - marked BNF grammar rule, 169
 - normal form, 33, 56
 - not left-recursive, 63
 - of regular language, 67
 - of Van Wijngarten, 89
 - operator, 62
 - parenthesized, 43, 161
 - representations, 32
 - right linearized, 171
 - right-linear, 69, 103
 - simple deterministic, 161, 242
 - strictly unilinear, 70
 - target, 299
 - translation, 298, 307
 - type 0, 86
 - type 1, 86
 - type 2, 86
 - type 3, 69, 86
 - unilinear, 69
- Graph
- context-sensitive, 87
 - dependence, 339
 - local automaton, 122
 - parser control-flow, 226
 - pilot, 183
 - program control-flow, 370
 - reachability, 36
 - sibling, 348
 - state-transition, 96
 - syntax chart, 169
 - syntax tree, 38
- Greibach normal form, 33, 65, 152
- Guide
- predicate, 367
 - set, 227, 232
- H**
- Handle, 176, 190
- Hierarchical list, 27
- Hierarchy of Chomsky, 86
- Homomorphism
- alphabetic, 79
 - nonalphabetic, 80
- I**
- Incremental
- compilation, 289
 - parser, 289
- Infix operator, 301
- Inherent ambiguity, 53, 119, 160
- Inherited attribute, *see* Right attribute
- Initial of a string, 10
- Initial state uniqueness, 109
- Initials, *see* Set of initials
- Instruction scheduling, 370
- Interference between variables, 378
- Intermediate
- language, 364
 - representation, 364, 370
- Internal attribute, 342
- Intraprocedural static analysis, 370
- Intersection
- of context-free and regular language, 153
 - of context-free languages, 79
 - of regular languages, 133, 134
- Intraprocedural static analysis, 370
- Inverse translation, 296
- Invertible grammar, 59
- IO-automaton, 328
- sequential, 329
- Item, *see* Candidate
- J**
- JavaScript Object Notation, 272
- JFLAP, 152
- JSON, 272
- Jumps, 365
- K**
- Kernel
- equivalent macro-states, 184
 - of macro-state, 183
- Knuth, 176, 197, 211, 244
- L**
- L condition for attribute evaluation, 354
- Language, 8
- abstraction, 24
 - artificial, 5
 - complement, 13
 - context free, 29, 86, 151
 - context sensitive, 87
 - context-free, 35
 - context-sensitive, 86
 - decidable, 95
 - Dyck, 42
 - empty, 8
 - equation, 71
 - finite, 8
 - formal, 5
 - formalized, 5
 - generated, 34
 - infinite, 8
 - recursive derivations, 37
 - local, 122

- Language (*cont.*)
- locally testable, 122
 - nullable, 10
 - recursive, 95
 - recursively enumerable, 95
 - regular, 19
 - replica, 77
 - with center, 88
 - source, 296
 - substitution of, 26
 - target, 81, 296
 - unary alphabet, 75
 - universal, 13
 - with three equal powers, 76, 87
 - with two equal powers, 73
- Least fixed point, 377
- Left
- attribute, 338, 341
 - derivation, 41
 - quotient, 17, 210
 - recursive grammar
 - top-down parsing, 214
 - recursive rule, 63
 - elimination, 63
- Left-linear grammar, 69, 112
- Left-recursion, 33
- elimination, 65
 - immediate, 64
 - LR(1)*, 211
 - nonimmediate, 64
- Left/right terminal set, 270
- Leftmost derivation, *see* Left derivation
- Length of string, 8
- Levenshtein distance, 286
- Lex, 332
- Lexeme, 352
- Lexical
- analysis, 352
 - attribute, 342, 353, 354
 - class, 352, 353
 - closed and open, 353
 - finite and non-, 353
 - level, 352
 - segmentation, 353
- Linear
- grammar, 68
 - language equation, 71
 - regular expression, 124
- Linguistic abstraction, 24
- List
- abstract, 25
 - concrete, 25
 - hierarchical, 27
 - with precedence, 27
- with separators, 25
- Live variable, 374, 383
- Live-in, 375
- Live-out, 375
- Liveness, 374, 376, 378
- equations, 376
 - interval, 373, 375
- LL(1)*
- grammar, 242
 - relation with *LR(1)*, 243
- LL(2)* example, 240
- LL(k)*
- grammar, 244
 - relation with *LR(k)*, 244
- Local
- automaton, 123
 - normalized, 123
 - language, 122, 136, 372
 - automaton, 122
 - composition of, 123
 - set, 125
 - regular expression, 125
 - testability, 122
- Locally testable language, 136
- Longest prefix rule, 353
- Look-ahead
- extending, 205
 - increasing, 240
 - set, 174
- LR(1)*
- family, 244
 - grammar, 244
 - grammar transformation, 205, 208
 - parser
 - relation with *ELR(1)*, 197
 - superfluous features, 201
 - relation with *DET*, 244
- LR(2)* to *LR(1)*, 207, 208
- LR(k)*
- early scanning, 207
 - grammar, 206
 - left quotient, 210
 - transformation, 211
- Lukasiewicz, 302
- M**
- M-state, *see* Macro-state
- Machine net, 165
- Macro-state, 182
- base, 183
 - closure, 183
 - kernel, 183
- Marked BNF grammar rule, 169, 207, 208
- McNaughton, 137

-
- Meaning, 334
 Memory allocation, 378
 Metagrammar, 31
 Metalanguage of regular expression, 30
 Minimal automaton, 100
 Minimization of automaton, 101
 Mirror reflection, 10
 Mixfix operator, 301
 Multi-sweep semantic evaluator, 351
 Multiple transition property, 184
- N**
 Nerode relation, 101
 Nested structure, 29
 Network of finite machines, 165
 Nivat theorem, 326
 Non-deterministic
 union, 158
 Non- $LR(k)$, 211
 Noncounting language, 136
 Nondeterminism motivation, 105
 Nondeterministic
 automaton conversion to deterministic, 114, 130
 finite automaton, 104, 107, 109
 pushdown automaton, 143, 144
 Nonnullable normal form, 57
 Nonterminal, 30
 alphabet, 31
 expansion, 56
 shift, 179
 in Earley parser, 251
 Normal form
 Chomsky, 61
 copy-free, 57
 Greibach, 65
 nonnullable, 57
 real-time, 65
 without repeated right parts, 59
 Normalized local automaton, 123
 Not left-recursive grammar, 63
 Nullable
 language, 10
 nonterminal, 57
 regular expression, 126
 Numbered regular expression, 126
- O**
OAG, 351
 On line machine, 150
 One-sweep
 attribute evaluation, 347
 attribute grammar, 348
 semantic evaluator construction, 349
- Opening mark, 25
 Operator
 binary, 301
 degree, 301
 infix, 301
 mixfix, 301
 postfix, 301
 prefix, 301
 unary, 301
 variadic, 301
 Operator grammar, 62
 normal form, 33, 62
 Operator precedence
 grammar, 269
 languages, 274
 relation, 269, 271
 sequential parser, 275
 Opposite passes, 331
 Optimization of program, 295, 370, 379, 383
- P**
 Palindrome, 30, 43, 162, 300
 nondeterminism, 159
 pushdown machine, 148
 Panic mode error recovery, 287
 Papert, 137
 Parallel parser, 268, 281
 Parenthesis
 language, 41, 161
 redundant, 304
 Parenthesized
 expression, 40
 grammar, 43, 161
 tree, 40, 261
 Parser, 162
 choice criteria, 246
 Coke–Younger–Kasami, 247
 Earley, 252
 general, 248
 local, 268, 269
 parallel, 268, 277, 281
 predictive, 236, 237
 recursive descent, 238, 239
 semantics-directed, 248, 366
 top-down and bottom-up, 163
 top-down deterministic, 237
 with attribute evaluation, 352
 with attribute evaluator, 355
 with translation, 311
 Parser control-flow graph, 226
 direct construction, 232
PCFG, 218, 226
 direct construction, 232

- Pilot
 compaction, 218
 construction, 183
 graph, 183
 machine of translator, 314
- Pin, 121
- PLZ-SYS*, 367
- Pointerless *ELR(1)* parser, 222
- Polish notation, 48, 300, 302
- Portability of compiler, 364
- Postaccessible state, 99
- Postfix
 normal form, 315
 operator, 301
- Power
 of language, 12
 of string, 10
- Powerset construction, 117
- Precedence
 of operators, 11
 relation, 269
- Prediction in Earley parser, 250
- Predictive
 parser, 236
 direct construction, 231
 parser automaton, 237
 pushdown automaton, 143
 pushdown transducer, 308
- Prefix, 10
 operator, 301
- Prefix-free language, 11
- Product
 machine, 134
 of automata, 134
- Production, 31
- Program
 analysis, 369
 optimization, 295, 370, 379, 383
- Projection, 80
- Prospect set, 226, 232
- Pumping property, 73
- Pure syntactic translation, 298
- Pushdown automaton, 95, 142, 143, 146, 151
 accepting modes, 149
 conversion to grammar, 151
 definition, 146
 determinism, 155
 deterministic subclasses, 160
 forms of non-determinism, 155
 time complexity, 146
- Pushdown IO-automaton, 305
- Pushdown transducer, 305, 307, 310
 from translation grammar, 307
 nondeterministic, 310
- Q**
- Quotient
 of grammars, 210
 of languages, 16
- R**
- Rabin and Scott, 323
 machine, 323
- Rational translation, *see* Regular translation
- Reachable
 nonterminal, 35
 state, 99
- Reaching definition, 380, 383
 always, 385
- Real-time normal form, 65
- Recognition algorithm, 92
- Recognize, *see* Accept
- Recursion bilateral, 47
- Recursive
 derivation, 37
 descent, 354
 parser, 213, 238, 239
 parser with attributes, 354
 translator, 311
 machine net, 165
 semantic evaluator, 350
- Reduce move, 176
- Reduce-reduce conflict, 182, 185, 206
- Reduce-shift conflict, *see* Shift-reduce conflict
- Reflection, 10
 of context-free language, 45, 78
 of language, 11
 recognizer of, 106
- REG* family, 19, 68, 74, 78, 82
 closure properties, 23
 included in *CF*, 68
- Register assignment, 378
- Regular expression, 17
 ambiguity, 21, 68, 69
 extended, 22, 135
 from automaton, 112
 language defined, 21
 linear, 124
 metalanguage, 30
 nullability, 126
 numbered, 22, 126
 Thompson method, 119
 with complement, 132
 with intersection, 132
- Regular expression to automaton, 119
 Berry and Sethi, 129
 structural method, 119
 Thompson, 119

- Regular language, 19
intersection, 133
pumping property, 73
- Regular translation, 321
expression, 323
- Repeated right parts, 59
- Replica, 77, 88
- Reps, 338
- Retargeting, 364
- Reversal as translation, 299
- Right
attribute, 338, 340, 341, 357
derivation, 41
- Right-linear grammar, 69, 103
- Right-linearized grammar, 200, 237
- Right-recursion, 33
LR(1), 211
- Rightmost derivation, *see* Right derivation
- Roger, 152
- Rule
Chomsky normal, 33
copy, 33, 57
empty, 33
Greibach normal, 33
homogeneous, 33
homogeneous binary, 61
left-linear, 33
left-recursive, 33
linear, 33
recursive, 33
right-linear, 33
right-recursive, 33
subcategorization, 33
terminal, 33
with operators, 33
- Run-time error, 364
- S**
- Sakarovich, 113, 134, 295
- Scan in Earley parser, 250
- Scanner, 352
- Scheduling of instructions, 370
- Searching a text for a word, 107
- Self-nested derivation, 74
- Self-nesting, 42
- Semantic
analysis, 335
attribute, 334, 338
lexical, 353
check, 361
error, 361
evaluator, 335, 347
multi-sweep, 351
one-sweep, 347
- recursive, 350
function, 336, 341
interpretation, 55
metalanguage, 334, 341
predicate, 361
procedure, 350
rule, 336
translation, 333
- Semantics, 5, 293, 333, 334
- Semantics-directed parser, 248, 366
- Sentence, 8
ambiguous, 21
- Sentential form, 34
- Sequential
function, 330
transducer, 329
- Set
of followers, 129, 173
of initials, 121, 174
- Set operations, 13
- Sethi, 129
- Shift
move, 176
terminal or nonterminal, 182
- Shift-reduce
conflict, 182, 185, 208
parser with attribute evaluator, 357
translator, 313
- Sibling graph, 348
- Simple deterministic grammar, 161, 242
- Single-transition property, 214, 218
- Single-valued translation, 305
- Sink state, *see* Trap state
- Skeleton tree, 40
condensed, 40
- Sms, 190
- Source
grammar, 299
ambiguity, 305
language, 296
- Spontaneous loop in pushdown machine, 150
- Spontaneous move, 105, 109, 147
elimination, 114, 115
- Stack
candidate, 190
m-state, 190
macro-state, 190
- Star
of context-free language, 44, 78
operation, 14
properties, 15
- Star-free language, 136
- State
accessible, 99

- State (*cont.*)
 - distinguishable, 100
 - postaccessible, 99
 - reachable, 99
 - trap, 98
 - useful, 99
 - useless, 99
- State-transition diagram, 96
 - pushdown machine, 148
- Static error, 361
- Static program analysis, 364, 369, 387
 - interprocedural, 370
 - intraprocedural, 370
- STP*, *see* Single-transition property
- Strictly unilinear grammar, 70
- String, 8
 - empty, 9
- String form, 34
- Strong equivalence of grammars, 55
- Structural adequacy, 55, 70
- Subcategorization rule, 57
- Subjacent automaton, 325, 328, 330, 332
 - pushdown, 306
- Substitution, 26, 81
 - closure property, 81
- Substring, 10
- Suffix, 10
- Symbol table, 285, 361
- Syntactic
 - analysis, 162
 - error, 285
 - level, 352
 - procedure, 212, 238
 - semantic analyzer, 355
 - support, 336, 341
 - translation, 298
 - translation scheme, 299
- Syntactic-semantic analysis, 352
- Syntax, 5
 - abstract, 24
 - analysis, 162
 - bottom-up and top-down, 163
 - chart, 169, 372
 - tree, 38
 - abstract, 303, 335
 - as translation, 319
- Syntax-directed
 - compiler, 334
 - translation, 121, 299, 334
- Synthesized attribute, *see* Left attribute
- T**
- Target
 - grammar, 299
- language, 81, 296
- Terminal
 - alphabet, 31
 - shift in Earley parser, 250
 - symbol, 7
- Thompson, 119
- Token, 352
- TOL*, 349
- Tomita, 248
- Top-down
 - deterministic translation, 311
 - parsing, 211
 - syntax analysis, 163
- Topological sort, 346
- TOR*, 349
- TOS*, 349
- Transition networks, 169
- Translation
 - closure properties, 332
 - function, 296, 297, 330
 - sequential, 330
 - grammar, 298, 307, 327
 - EBNF*, 313
 - postfix normal form, 315
 - rule normalization, 307
 - to pushdown transducer, 307
 - rational, *see* Regular translation
 - regular, *see* Regular translation
 - relation, 296
 - scheme, 299
 - single-valued, 305
 - syntactic, 298
- Translator
 - bottom-up, 313
 - comparison, 320
 - top-down, 311
 - two-way, 331
 - with recursive procedures, 311
- Transliteration, 79, 298
 - to words, 80
- Trap state, 98
- Tree
 - construction, 261
 - decorated, 337
 - syntax, 38
- Tree pattern matching, 364
- Turing machine, 86, 95
- Two-input automaton, 323, 324
- Two-pass compiler, 335
- Type 0 grammar, 86
- Type 1 grammar, 86
- Type 2 grammar, 86
- Type 3 grammar, 69, 86
- Type checking, 361

U

- Unary
 - alphabet, 75
 - operator, 301
- Undecidability
 - deterministic context-free language, 245
 - deterministic pushdown automaton, 245
 - $LR(k)$, 245
- Undistinguishability of states, 101
- Unilinear grammar, 69, 110
 - equations, 71
- Uninitialized variable, 386
- Union of context-free languages, 43, 78
- Universal language, 13, 14
- Unreachable code, 373, 384
- Useful state, 99
- Useless
 - assignment, 374
 - state, 99
 - elimination, 99
 - variable definition, 379
- Uzgalis, 89

V

- Van Wijngarten, 89
- Variable
 - availability, 385
 - definition, 371
 - suppressed, 381
 - initialization, 385, 386
 - use, 371
- Variadic operator, 301
- Vector-stack parser, 201
- Vocabulary, 8
- VW grammar, 89

W

- Weak equivalence of grammars, 55
- Well-defined nonterminal, 35
- Word, 8

X

- XML, 42, 162

Y

- Yang, 332