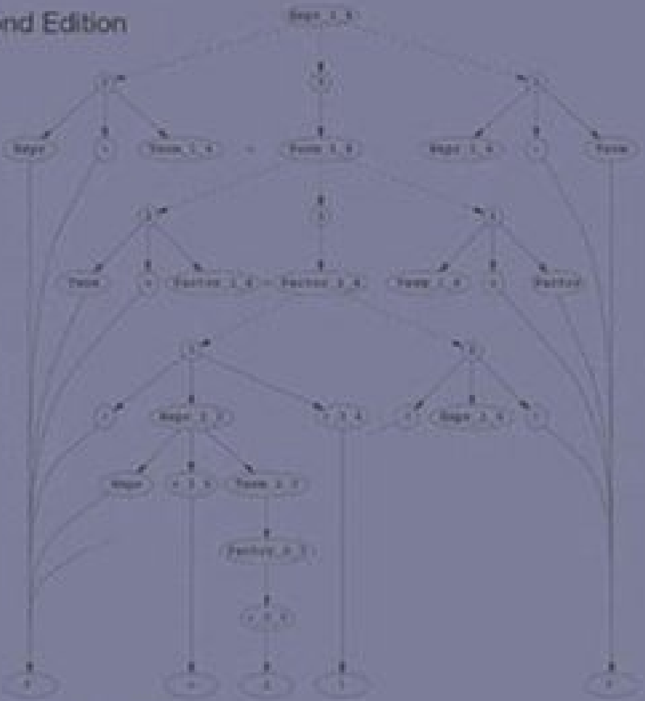
MONOGRAPHS IN COMPUTER SCIENCE

PARSING TECHNIQUES

A Practical Guide

Dick Grune
Ceriel J.H. Jacobs

Second Edition



Springer

Monographs in Computer Science

Editors

David Gries
Fred B. Schneider

Monographs in Computer Science

Abadi and Cardelli, **A Theory of Objects**

Benosman and Kang [editors], **Panoramic Vision: Sensors, Theory, and Applications**

Bhanu, Lin, Krawiec, **Evolutionary Synthesis of Pattern Recognition Systems**

Broy and Stølen, **Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces, and Refinement**

Brzozowski and Seger, **Asynchronous Circuits**

Burgin, **Super-Recursive Algorithms**

Cantone, Omodeo, and Policriti, **Set Theory for Computing: From Decision Procedures to Declarative Programming with Sets**

Castillo, Gutiérrez, and Hadi, **Expert Systems and Probabilistic Network Models**

Downey and Fellows, **Parameterized Complexity**

Feijen and van Gasteren, **On a Method of Multiprogramming**

Grune and Jacobs, **Parsing Techniques: A Practical Guide, Second Edition**

Herbert and Spärck Jones [editors], **Computer Systems: Theory, Technology, and Applications**

Leiss, **Language Equations**

Levin, Heydon, Mann, and Yu, **Software Configuration Management Using VESTA**

McIver and Morgan [editors], **Programming Methodology**

McIver and Morgan [editors], **Abstraction, Refinement and Proof for Probabilistic Systems**

Misra, **A Discipline of Multiprogramming: Programming Theory for Distributed Applications**

Nielson [editor], **ML with Concurrency**

Paton [editor], **Active Rules in Database Systems**

Poernomo, Crossley, and Wirsing, **Adapting Proof-as-Programs: The Curry-Howard Protocol**

Selig, **Geometrical Methods in Robotics**

Selig, **Geometric Fundamentals of Robotics, Second Edition**

Shasha and Zhu, **High Performance Discovery in Time Series: Techniques and Case Studies**

Dick Grune
Ceriel J.H. Jacobs

Parsing Techniques

A Practical Guide

Second Edition

Springer

Dick Grune and Ceriel J.H. Jacobs
Faculteit Exacte Wetenschappen
Vrije Universiteit
De Boelelaan 1081
1081 HV Amsterdam
The Netherlands

Series Editors
David Gries
Department of Computer Science
Cornell University
4130 Upson Hall
Ithaca, NY 14853-7501
USA

Fred P. Schneider
Department of Computer Science
Cornell University
4130 Upson Hall
Ithaca, NY 14853-7501
USA

ISBN-13: 978-0-387-20248-8 e-ISBN-13: 978-0-387-68954-8

Library of Congress Control Number: 2007936901

©2008 Springer Science+Business Media, LLC

©1990 Ellis Horwood Ltd.

All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media LLC, 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use in this publication of trade names, trademarks, service marks and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed on acid-free paper. (SB)

9 8 7 6 5 4 3 2 1

Preface to the Second Edition

As is fit, this second edition arose out of our readers' demands to read about new developments and our desire to write about them. Although parsing techniques is not a fast moving field, it does move. When the first edition went to press in 1990, there was only one tentative and fairly restrictive algorithm for linear-time substring parsing. Now there are several powerful ones, covering all deterministic languages; we describe them in Chapter 12. In 1990 Theorem 8.1 from a 1961 paper by Bar-Hillel, Perles, and Shamir lay gathering dust; in the last decade it has been used to create new algorithms, and to obtain insight into existing ones. We report on this in Chapter 13.

More and more non-Chomsky systems are used, especially in linguistics. None except two-level grammars had any prominence 20 years ago; we now describe six of them in Chapter 15. Non-canonical parsers were considered oddities for a very long time; now they are among the most powerful linear-time parsers we have; see Chapter 10.

Although still not very practical, marvelous algorithms for parallel parsing have been designed that shed new light on the principles; see Chapter 14. In 1990 a generalized LL parser was deemed impossible; now we describe two in Chapter 11.

Traditionally, and unsurprisingly, parsers have been used for parsing; more recently they are also being used for code generation, data compression and logic language implementation, as shown in Section 17.5. Enough. The reader can find more developments in many places in the book and in the Annotated Bibliography in Chapter 18.

Kees van Reeuwijk has — only half in jest — called our book "a reservation for endangered parsers". We agree — partly; it is more than that — and we make no apologies. Several algorithms in this book have very limited or just no practical value. We have included them because we feel they embody interesting ideas and offer food for thought; they might also grow and acquire practical value. But we also include many algorithms that do have practical value but are sorely underused; describing them here might raise their status in the world.

Exercises and Problems

This book is not a textbook in the school sense of the word. Few universities have a course in Parsing Techniques, and, as stated in the Preface to the First Edition, readers will have very different motivations to use this book. We have therefore included hardly any questions or tasks that exercise the material contained within this book; readers can no doubt make up such tasks for themselves. The questions posed in the problem sections at the end of each chapter usually require the reader to step outside the bounds of the covered material. The problems have been divided into three not too well-defined classes:

- not marked — probably doable in a few minutes to a couple of hours.
- marked *Project* — probably a lot of work, but almost certainly doable.
- marked *Research Project* — almost certainly a lot of work, but hopefully doable.

We make no claims as to the relevance of any of these problems; we hope that some readers will find some of them enlightening, interesting, or perhaps even useful. Ideas, hints, and partial or complete solutions to a number of the problems can be found in Chapter A.

There are also a few questions on formal language that were not answered easily in the existing literature but have some importance to parsing. These have been marked accordingly in the problem sections.

Annotated Bibliography

For the first edition, we, the authors, read and summarized all papers on parsing that we could lay our hands on. Seventeen years later, with the increase in publications and easier access thanks to the Internet, that is no longer possible, much to our chagrin. In the first edition we included all relevant summaries. Again that is not possible now, since doing so would have greatly exceeded the number of pages allotted to this book. The printed version of this second edition includes only those references to the literature and their summaries that are actually referred to in this book. The complete bibliography with summaries as far as available can be found on the web site of this book; it includes its own authors index and subject index. This setup also allows us to list without hesitation technical reports and other material of possibly low accessibility. Often references to sections from Chapter 18 refer to the Web version of those sections; attention is drawn to this by calling them "(Web)Sections".

We do not supply URLs in this book, for two reasons: they are ephemeral and may be incorrect next year, tomorrow, or even before the book is printed; and, especially for software, better URLs may be available by the time you read this book. The best URL is a few well-chosen search terms submitted to a good Web search engine.

Even in the last ten years we have seen a number of Ph.D theses written in languages other than English, specifically German, French, Spanish and Estonian. This choice of language has the regrettable but predictable consequence that their contents have been left out of the main stream of science. This is a loss, both to the authors and to the scientific community. Whether we like it or not, English is the de facto standard language of present-day science. The time that a scientifically interested gentleman of leisure could be expected to read French, German, English, Greek, Latin and a tad of Sanskrit is 150 years in the past; today, students and scientists need the room in their heads and the time in their schedules for the vastly increased amount of knowledge. Although we, the authors, can still read most (but not all) of the above languages and have done our best to represent the contents of the non-English theses adequately, this will not suffice to give them the international attention they deserve.

The Future of Parsing, aka The Crystal Ball

If there will ever be a third edition of this book, we expect it to be substantially thinner (except for the bibliography section!). The reason is that the more parsing algorithms one studies the more they seem similar, and there seems to be great opportunity for unification. Basically almost all parsing is done by top-down search with left-recursion protection; this is true even for traditional bottom-up techniques like LR(1), where the top-down search is built into the LR(1) parse tables. In this respect it is significant that Earley's method is classified as top-down by some and as bottom-up by others. The general memoizing mechanism of tabular parsing takes the exponential sting out of the search. And it seems likely that transforming the usual depth-first search into breadth-first search will yield many of the generalized deterministic algorithms; in this respect we point to Sikkel's Ph.D thesis [158]. Together this seems to cover almost all algorithms in this book, including parsing by intersection. Pure bottom-up parsers without a top-down component are rare and not very powerful.

So in the theoretical future of parsing we see considerable simplification through unification of algorithms; the role that parsing by intersection can play in this is not clear. The simplification does not seem to extend to formal languages: it is still as difficult to prove the intuitively obvious fact that all LL(1) grammars are LR(1) as it was 35 years ago.

The practical future of parsing may lie in advanced pattern recognition, in addition to its traditional tasks; the practical contributions of parsing by intersection are again not clear.

Amsterdam, Amstelveen
June 2007

Dick Grune
Ceriel J.H. Jacobs

Acknowledgments

We thank Manuel E. Bermudez, Stuart Broad, Peter Bumbulis, Salvador Cavadini, Carl Cerecke, Julia Dain, Akim Demaille, Matthew Estes, Wan Fokkink, Brian Ford, Richard Frost, Clemens Grabmayer, Robert Grimm, Karin Harbusch, Stephen Horne, Jaco Imthorn, Quinn Tyler Jackson, Adrian Johnstone, Michiel Koens, Jaroslav Král, Olivier Lecarme, Lillian Lee, Olivier Lefevre, Joop Leo, JianHua Li, Neil Mitchell, Peter Pepper, Wim Pijls, José F. Quesada, Kees van Reeuwijk, Walter L. Ruzzo, Lothar Schmitz, Sylvain Schmitz, Thomas Schoebel-Theuer, Klaas Sikkel, Michael Sperberg-McQueen, Michal Žemlička, Hans Åberg, and many others, for helpful correspondence, comments on and errata to the First Edition, and support for the Second Edition. In particular we want to thank Kees van Reeuwijk and Sylvain Schmitz for their extensive "beta reading", which greatly helped the book — and us.

We thank the Faculteit Exacte Wetenschappen of the Vrije Universiteit for the use of their equipment.

In a wider sense, we extend our thanks to the close to 1500 authors listed in the (Web)Authors Index, who have been so kind as to invent scores of clever and elegant algorithms and techniques for us to exhibit. Every page of this book leans on them.

Preface to the First Edition

Parsing (syntactic analysis) is one of the best understood branches of computer science. Parsers are already being used extensively in a number of disciplines: in computer science (for compiler construction, database interfaces, self-describing databases, artificial intelligence), in linguistics (for text analysis, corpora analysis, machine translation, textual analysis of biblical texts), in document preparation and conversion, in typesetting chemical formulae and in chromosome recognition, to name a few; they can be used (and perhaps are) in a far larger number of disciplines. It is therefore surprising that there is no book which collects the knowledge about parsing and explains it to the non-specialist. Part of the reason may be that parsing has a name for being "difficult". In discussing the Amsterdam Compiler Kit and in teaching compiler construction, it has, however, been our experience that seemingly difficult parsing techniques can be explained in simple terms, given the right approach. The present book is the result of these considerations.

This book does not address a strictly uniform audience. On the contrary, while writing this book, we have consistently tried to imagine giving a course on the subject to a diffuse mixture of students and faculty members of assorted faculties, sophisticated laymen, the avid readers of the science supplement of the large newspapers, etc. Such a course was never given; a diverse audience like that would be too uncoordinated to convene at regular intervals, which is why we wrote this book, to be read, studied, perused or consulted wherever or whenever desired.

Addressing such a varied audience has its own difficulties (and rewards). Although no explicit math was used, it could not be avoided that an amount of mathematical thinking should pervade this book. Technical terms pertaining to parsing have of course been explained in the book, but sometimes a term on the fringe of the subject has been used without definition. Any reader who has ever attended a lecture on a non-familiar subject knows the phenomenon. He skips the term, assumes it refers to something reasonable and hopes it will not recur too often. And then there will be passages where the reader will think we are elaborating the obvious (this paragraph may be one such place). The reader may find solace in the fact that he does not have to doodle his time away or stare out of the window until the lecturer progresses.

On the positive side, and that is the main purpose of this enterprise, we hope that by means of a book with this approach we can reach those who were dimly aware of the existence and perhaps of the usefulness of parsing but who thought it would forever be hidden behind phrases like:

$$\text{Let } \mathfrak{P} \text{ be a mapping } V_N \xrightarrow{\Phi} 2^{(V_N \cup V_T)^*} \text{ and } \mathfrak{H} \text{ a homomorphism} \ldots$$

No knowledge of any particular programming language is required. The book contains two or three programs in Pascal, which serve as actualizations only and play a minor role in the explanation. What is required, though, is an understanding of algorithmic thinking, especially of recursion. Books like *Learning to program* by Howard Johnston (Prentice-Hall, 1985) or *Programming from first principles* by Richard Bornat (Prentice-Hall 1987) provide an adequate background (but supply more detail than required). Pascal was chosen because it is about the only programming language more or less widely available outside computer science environments.

The book features an extensive annotated bibliography. The user of the bibliography is expected to be more than casually interested in parsing and to possess already a reasonable knowledge of it, either through this book or otherwise. The bibliography as a list serves to open up the more accessible part of the literature on the subject to the reader; the annotations are in terse technical prose and we hope they will be useful as stepping stones to reading the actual articles.

On the subject of applications of parsers, this book is vague. Although we suggest a number of applications in Chapter 1, we lack the expertise to supply details. It is obvious that musical compositions possess a structure which can largely be described by a grammar and thus is amenable to parsing, but we shall have to leave it to the musicologists to implement the idea. It was less obvious to us that behaviour at corporate meetings proceeds according to a grammar, but we are told that this is so and that it is a subject of socio-psychological research.

Acknowledgements

We thank the people who helped us in writing this book. Marion de Krieger has retrieved innumerable books and copies of journal articles for us and without her effort the annotated bibliography would be much further from completeness. Ed Keizer has patiently restored peace between us and the pic|tbl|eqn|psfig|troff pipeline, on the many occasions when we abused, overloaded or just plainly misunderstood the latter. Leo van Moergestel has made the hardware do things for us that it would not do for the uninitiated. We also thank Erik Baalbergen, Frans Kaashoek, Erik Groeneveld, Gerco Ballintijn, Jaco Imthorn, and Egon Amada for their critical remarks and contributions. The rose at the end of Chapter 2 is by Arwen Grune. Ilana and Lily Grune typed parts of the text on various occasions.

We thank the Faculteit Wiskunde en Informatica of the Vrije Universiteit for the use of the equipment.

In a wider sense, we extend our thanks to the hundreds of authors who have been so kind as to invent scores of clever and elegant algorithms and techniques for us to exhibit. We hope we have named them all in our bibliography.

Amsterdam, Amstelveen	*Dick Grune*
July 1990	*Ceriel J.H. Jacobs*

Contents

Preface to the Second Edition v

Preface to the First Edition xi

1 Introduction 1
  1.1 Parsing as a Craft 2
  1.2 The Approach Used 2
  1.3 Outline of the Contents 3
  1.4 The Annotated Bibliography 4

2 Grammars as a Generating Device 5
  2.1 Languages as Infinite Sets 5
    2.1.1 Language 5
    2.1.2 Grammars 7
    2.1.3 Problems with Infinite Sets 8
    2.1.4 Describing a Language through a Finite Recipe 12
  2.2 Formal Grammars 14
    2.2.1 The Formalism of Formal Grammars 14
    2.2.2 Generating Sentences from a Formal Grammar 15
    2.2.3 The Expressive Power of Formal Grammars 17
  2.3 The Chomsky Hierarchy of Grammars and Languages 19
    2.3.1 Type 1 Grammars 19
    2.3.2 Type 2 Grammars 23
    2.3.3 Type 3 Grammars 30
    2.3.4 Type 4 Grammars 33
    2.3.5 Conclusion 34
  2.4 Actually Generating Sentences from a Grammar 34
    2.4.1 The Phrase-Structure Case 34
    2.4.2 The CS Case 36
    2.4.3 The CF Case 36
  2.5 To Shrink or Not To Shrink 38

2.6 Grammars that Produce the Empty Language 41
2.7 The Limitations of CF and FS Grammars 42
  2.7.1 The *uvwxy* Theorem 42
  2.7.2 The *uvw* Theorem 45
2.8 CF and FS Grammars as Transition Graphs 45
2.9 Hygiene in Context-Free Grammars 47
  2.9.1 Undefined Non-Terminals 48
  2.9.2 Unreachable Non-Terminals 48
  2.9.3 Non-Productive Rules and Non-Terminals 48
  2.9.4 Loops 48
  2.9.5 Cleaning up a Context-Free Grammar 49
2.10 Set Properties of Context-Free and Regular Languages 52
2.11 The Semantic Connection 54
  2.11.1 Attribute Grammars 54
  2.11.2 Transduction Grammars 55
  2.11.3 Augmented Transition Networks 56
2.12 A Metaphorical Comparison of Grammar Types 56
2.13 Conclusion 59

3 Introduction to Parsing 61
3.1 The Parse Tree 61
  3.1.1 The Size of a Parse Tree 62
  3.1.2 Various Kinds of Ambiguity 63
  3.1.3 Linearization of the Parse Tree 65
3.2 Two Ways to Parse a Sentence 65
  3.2.1 Top-Down Parsing 66
  3.2.2 Bottom-Up Parsing 67
  3.2.3 Applicability 68
3.3 Non-Deterministic Automata 69
  3.3.1 Constructing the NDA 70
  3.3.2 Constructing the Control Mechanism 70
3.4 Recognition and Parsing for Type 0 to Type 4 Grammars 71
  3.4.1 Time Requirements 71
  3.4.2 Type 0 and Type 1 Grammars 72
  3.4.3 Type 2 Grammars 73
  3.4.4 Type 3 Grammars 75
  3.4.5 Type 4 Grammars 75
3.5 An Overview of Context-Free Parsing Methods 76
  3.5.1 Directionality 76
  3.5.2 Search Techniques 77
  3.5.3 General Directional Methods 78
  3.5.4 Linear Methods 80
  3.5.5 Deterministic Top-Down and Bottom-Up Methods 82
  3.5.6 Non-Canonical Methods 83
  3.5.7 Generalized Linear Methods 84

3.5.8 Conclusion . . . 84
3.6 The "Strength" of a Parsing Technique . . . 84
3.7 Representations of Parse Trees . . . 85
3.7.1 Parse Trees in the Producer-Consumer Model . . . 86
3.7.2 Parse Trees in the Data Structure Model . . . 87
3.7.3 Parse Forests . . . 87
3.7.4 Parse-Forest Grammars . . . 91
3.8 When are we done Parsing? . . . 93
3.9 Transitive Closure . . . 95
3.10 The Relation between Parsing and Boolean Matrix Multiplication . . 97
3.11 Conclusion . . . 100

4 General Non-Directional Parsing . . . 103
4.1 Unger's Parsing Method . . . 104
4.1.1 Unger's Method without ε-Rules or Loops . . . 104
4.1.2 Unger's Method with ε-Rules . . . 107
4.1.3 Getting Parse-Forest Grammars from Unger Parsing . . . 110
4.2 The CYK Parsing Method . . . 112
4.2.1 CYK Recognition with General CF Grammars . . . 112
4.2.2 CYK Recognition with a Grammar in Chomsky Normal Form 116
4.2.3 Transforming a CF Grammar into Chomsky Normal Form . . 119
4.2.4 The Example Revisited . . . 122
4.2.5 CYK Parsing with Chomsky Normal Form . . . 124
4.2.6 Undoing the Effect of the CNF Transformation . . . 125
4.2.7 A Short Retrospective of CYK . . . 128
4.2.8 Getting Parse-Forest Grammars from CYK Parsing . . . 129
4.3 Tabular Parsing . . . 129
4.3.1 Top-Down Tabular Parsing . . . 131
4.3.2 Bottom-Up Tabular Parsing . . . 133
4.4 Conclusion . . . 134

5 Regular Grammars and Finite-State Automata . . . 137
5.1 Applications of Regular Grammars . . . 137
5.1.1 Regular Languages in CF Parsing . . . 137
5.1.2 Systems with Finite Memory . . . 139
5.1.3 Pattern Searching . . . 141
5.1.4 SGML and XML Validation . . . 141
5.2 Producing from a Regular Grammar . . . 141
5.3 Parsing with a Regular Grammar . . . 143
5.3.1 Replacing Sets by States . . . 144
5.3.2 ε-Transitions and Non-Standard Notation . . . 147
5.4 Manipulating Regular Grammars and Regular Expressions . . . 148
5.4.1 Regular Grammars from Regular Expressions . . . 149
5.4.2 Regular Expressions from Regular Grammars . . . 151
5.5 Manipulating Regular Languages . . . 152

5.6 Left-Regular Grammars 154
5.7 Minimizing Finite-State Automata 156
5.8 Top-Down Regular Expression Recognition 158
5.8.1 The Recognizer 158
5.8.2 Evaluation 159
5.9 Semantics in FS Systems 160
5.10 Fast Text Search Using Finite-State Automata 161
5.11 Conclusion 162

6 General Directional Top-Down Parsing 165
6.1 Imitating Leftmost Derivations 165
6.2 The Pushdown Automaton 167
6.3 Breadth-First Top-Down Parsing 171
6.3.1 An Example 173
6.3.2 A Counterexample: Left Recursion 173
6.4 Eliminating Left Recursion 175
6.5 Depth-First (Backtracking) Parsers 176
6.6 Recursive Descent 177
6.6.1 A Naive Approach 179
6.6.2 Exhaustive Backtracking Recursive Descent 183
6.6.3 Breadth-First Recursive Descent 185
6.7 Definite Clause Grammars 188
6.7.1 Prolog 188
6.7.2 The DCG Format 189
6.7.3 Getting Parse Tree Information 190
6.7.4 Running Definite Clause Grammar Programs 190
6.8 Cancellation Parsing 192
6.8.1 Cancellation Sets 192
6.8.2 The Transformation Scheme 193
6.8.3 Cancellation Parsing with ε-Rules 196
6.9 Conclusion 197

7 General Directional Bottom-Up Parsing 199
7.1 Parsing by Searching 201
7.1.1 Depth-First (Backtracking) Parsing 201
7.1.2 Breadth-First (On-Line) Parsing 202
7.1.3 A Combined Representation 203
7.1.4 A Slightly More Realistic Example 204
7.2 The Earley Parser 206
7.2.1 The Basic Earley Parser 206
7.2.2 The Relation between the Earley and CYK Algorithms 212
7.2.3 Handling ε-Rules 214
7.2.4 Exploiting Look-Ahead 219
7.2.5 Left and Right Recursion 224
7.3 Chart Parsing 226

- 7.3.1 Inference Rules . . . 227
- 7.3.2 A Transitive Closure Algorithm . . . 227
- 7.3.3 Completion . . . 229
- 7.3.4 Bottom-Up (Actually Left-Corner) . . . 229
- 7.3.5 The Agenda . . . 229
- 7.3.6 Top-Down . . . 231
- 7.3.7 Conclusion . . . 232
- 7.4 Conclusion . . . 233

8 Deterministic Top-Down Parsing . . . 235
- 8.1 Replacing Search by Table Look-Up . . . 236
- 8.2 LL(1) Parsing . . . 239
  - 8.2.1 LL(1) Parsing without ε-Rules . . . 239
  - 8.2.2 LL(1) Parsing with ε-Rules . . . 242
  - 8.2.3 LL(1) versus Strong-LL(1) . . . 247
  - 8.2.4 Full LL(1) Parsing . . . 248
  - 8.2.5 Solving LL(1) Conflicts . . . 251
  - 8.2.6 LL(1) and Recursive Descent . . . 253
- 8.3 Increasing the Power of Deterministic LL Parsing . . . 254
  - 8.3.1 LL(k) Grammars . . . 254
  - 8.3.2 Linear-Approximate LL(k) . . . 256
  - 8.3.3 LL-Regular . . . 257
- 8.4 Getting a Parse Tree Grammar from LL(1) Parsing . . . 258
- 8.5 Extended LL(1) Grammars . . . 259
- 8.6 Conclusion . . . 260

9 Deterministic Bottom-Up Parsing . . . 263
- 9.1 Simple Handle-Finding Techniques . . . 265
- 9.2 Precedence Parsing . . . 266
  - 9.2.1 Parenthesis Generators . . . 267
  - 9.2.2 Constructing the Operator-Precedence Table . . . 269
  - 9.2.3 Precedence Functions . . . 271
  - 9.2.4 Further Precedence Methods . . . 272
- 9.3 Bounded-Right-Context Parsing . . . 275
  - 9.3.1 Bounded-Context Techniques . . . 276
  - 9.3.2 Floyd Productions . . . 277
- 9.4 LR Methods . . . 278
- 9.5 LR(0) . . . 280
  - 9.5.1 The LR(0) Automaton . . . 280
  - 9.5.2 Using the LR(0) Automaton . . . 283
  - 9.5.3 LR(0) Conflicts . . . 286
  - 9.5.4 ε-LR(0) Parsing . . . 287
  - 9.5.5 Practical LR Parse Table Construction . . . 289
- 9.6 LR(1) . . . 290
  - 9.6.1 LR(1) with ε-Rules . . . 295

9.6.2 LR($k > 1$) Parsing . . . 297
9.6.3 Some Properties of LR(k) Parsing . . . 299
9.7 LALR(1) . . . 300
9.7.1 Constructing the LALR(1) Parsing Tables . . . 302
9.7.2 Identifying LALR(1) Conflicts . . . 314
9.8 SLR(1) . . . 314
9.9 Conflict Resolvers . . . 315
9.10 Further Developments of LR Methods . . . 316
9.10.1 Elimination of Unit Rules . . . 316
9.10.2 Reducing the Stack Activity . . . 317
9.10.3 Regular Right Part Grammars . . . 318
9.10.4 Incremental Parsing . . . 318
9.10.5 Incremental Parser Generation . . . 318
9.10.6 Recursive Ascent . . . 319
9.10.7 Regular Expressions of LR Languages . . . 319
9.11 Getting a Parse Tree Grammar from LR Parsing . . . 319
9.12 Left and Right Contexts of Parsing Decisions . . . 320
9.12.1 The Left Context of a State . . . 321
9.12.2 The Right Context of an Item . . . 322
9.13 Exploiting the Left and Right Contexts . . . 323
9.13.1 Discriminating-Reverse (DR) Parsing . . . 324
9.13.2 LR-Regular . . . 327
9.13.3 LAR(m) Parsing . . . 333
9.14 LR(k) as an Ambiguity Test . . . 338
9.15 Conclusion . . . 338

10 Non-Canonical Parsers . . . 343
10.1 Top-Down Non-Canonical Parsing . . . 344
10.1.1 Left-Corner Parsing . . . 344
10.1.2 Deterministic Cancellation Parsing . . . 353
10.1.3 Partitioned LL . . . 354
10.1.4 Discussion . . . 357
10.2 Bottom-Up Non-Canonical Parsing . . . 357
10.2.1 Total Precedence . . . 358
10.2.2 NSLR(1) . . . 359
10.2.3 LR(k,∞) . . . 364
10.2.4 Partitioned LR . . . 372
10.3 General Non-Canonical Parsing . . . 377
10.4 Conclusion . . . 379

11 Generalized Deterministic Parsers . . . 381
11.1 Generalized LR Parsing . . . 382
11.1.1 The Basic GLR Parsing Algorithm . . . 382
11.1.2 Necessary Optimizations . . . 383
11.1.3 Hidden Left Recursion and Loops . . . 387

11.1.4 Extensions and Improvements 390
11.2 Generalized LL Parsing 391
11.2.1 Simple Generalized LL Parsing 391
11.2.2 Generalized LL Parsing with Left-Recursion 393
11.2.3 Generalized LL Parsing with ε-Rules 395
11.2.4 Generalized Cancellation and LC Parsing 397
11.3 Conclusion 398

12 Substring Parsing 399
12.1 The Suffix Grammar 401
12.2 General (Non-Linear) Methods 402
12.2.1 A Non-Directional Method 403
12.2.2 A Directional Method 407
12.3 Linear-Time Methods for LL and LR Grammars 408
12.3.1 Linear-Time Suffix Parsing for LL(1) Grammars 409
12.3.2 Linear-Time Suffix Parsing for LR(1) Grammars 414
12.3.3 Tabular Methods 418
12.3.4 Discussion 421
12.4 Conclusion 421

13 Parsing as Intersection 425
13.1 The Intersection Algorithm 426
13.1.1 The Rule Sets I_{rules}, I_{rough}, and I 427
13.1.2 The Languages of I_{rules}, I_{rough}, and I 429
13.1.3 An Example: Parsing Arithmetic Expressions 430
13.2 The Parsing of FSAs 431
13.2.1 Unknown Tokens 431
13.2.2 Substring Parsing by Intersection 431
13.2.3 Filtering 435
13.3 Time and Space Requirements 436
13.4 Reducing the Intermediate Size: Earley's Algorithm on FSAs 437
13.5 Error Handling Using Intersection Parsing 439
13.6 Conclusion 441

14 Parallel Parsing 443
14.1 The Reasons for Parallel Parsing 443
14.2 Multiple Serial Parsers 444
14.3 Process-Configuration Parsers 447
14.3.1 A Parallel Bottom-up GLR Parser 448
14.3.2 Some Other Process-Configuration Parsers 452
14.4 Connectionist Parsers 453
14.4.1 Boolean Circuits 453
14.4.2 A CYK Recognizer on a Boolean Circuit 454
14.4.3 Rytter's Algorithm 460
14.5 Conclusion 470

15 Non-Chomsky Grammars and Their Parsers 473
15.1 The Unsuitability of Context-Sensitive Grammars 473
15.1.1 Understanding Context-Sensitive Grammars 474
15.1.2 Parsing with Context-Sensitive Grammars 475
15.1.3 Expressing Semantics in Context-Sensitive Grammars 475
15.1.4 Error Handling in Context-Sensitive Grammars 475
15.1.5 Alternatives 476
15.2 Two-Level Grammars 476
15.2.1 VW Grammars 477
15.2.2 Expressing Semantics in a VW Grammar 480
15.2.3 Parsing with VW Grammars 482
15.2.4 Error Handling in VW Grammars 484
15.2.5 Infinite Symbol Sets 484
15.3 Attribute and Affix Grammars 485
15.3.1 Attribute Grammars 485
15.3.2 Affix Grammars 488
15.4 Tree-Adjoining Grammars 492
15.4.1 Cross-Dependencies 492
15.4.2 Parsing with TAGs 497
15.5 Coupled Grammars 500
15.5.1 Parsing with Coupled Grammars 501
15.6 Ordered Grammars 502
15.6.1 Rule Ordering by Control Grammar 502
15.6.2 Parsing with Rule-Ordered Grammars 503
15.6.3 Marked Ordered Grammars 504
15.6.4 Parsing with Marked Ordered Grammars 505
15.7 Recognition Systems 506
15.7.1 Properties of a Recognition System 507
15.7.2 Implementing a Recognition System 509
15.7.3 Parsing with Recognition Systems 512
15.7.4 Expressing Semantics in Recognition Systems 512
15.7.5 Error Handling in Recognition Systems 513
15.8 Boolean Grammars 514
15.8.1 Expressing Context Checks in Boolean Grammars 514
15.8.2 Parsing with Boolean Grammars 516
15.8.3 §-Calculus 516
15.9 Conclusion 517

16 Error Handling 521
16.1 Detection versus Recovery versus Correction 521
16.2 Parsing Techniques and Error Detection 523
16.2.1 Error Detection in Non-Directional Parsing Methods 523
16.2.2 Error Detection in Finite-State Automata 524
16.2.3 Error Detection in General Directional Top-Down Parsers . . 524
16.2.4 Error Detection in General Directional Bottom-Up Parsers . . 524

16.2.5 Error Detection in Deterministic Top-Down Parsers 525
16.2.6 Error Detection in Deterministic Bottom-Up Parsers 525
16.3 Recovering from Errors 526
16.4 Global Error Handling 526
16.5 Regional Error Handling 530
16.5.1 Backward/Forward Move Error Recovery 530
16.5.2 Error Recovery with Bounded-Context Grammars 532
16.6 Local Error Handling 533
16.6.1 Panic Mode 534
16.6.2 FOLLOW-Set Error Recovery 534
16.6.3 Acceptable-Sets Derived from Continuations 535
16.6.4 Insertion-Only Error Correction 537
16.6.5 Locally Least-Cost Error Recovery 539
16.7 Non-Correcting Error Recovery 540
16.7.1 Detection and Recovery 540
16.7.2 Locating the Error 541
16.8 Ad Hoc Methods 542
16.8.1 Error Productions 542
16.8.2 Empty Table Slots 543
16.8.3 Error Tokens 543
16.9 Conclusion 543

17 Practical Parser Writing and Usage 545
17.1 A Comparative Survey 545
17.1.1 Considerations 545
17.1.2 General Parsers 546
17.1.3 General Substring Parsers 547
17.1.4 Linear-Time Parsers 548
17.1.5 Linear-Time Substring Parsers 549
17.1.6 Obtaining and Using a Parser Generator 549
17.2 Parser Construction 550
17.2.1 Interpretive, Table-Based, and Compiled Parsers 550
17.2.2 Parsing Methods and Implementations 551
17.3 A Simple General Context-Free Parser 553
17.3.1 Principles of the Parser 553
17.3.2 The Program 554
17.3.3 Handling Left Recursion 559
17.3.4 Parsing in Polynomial Time 560
17.4 Programming Language Paradigms 563
17.4.1 Imperative and Object-Oriented Programming 563
17.4.2 Functional Programming 564
17.4.3 Logic Programming 567
17.5 Alternative Uses of Parsing 567
17.5.1 Data Compression 567
17.5.2 Machine Code Generation 570

17.5.3 Support of Logic Languages 573
17.6 Conclusion 573

18 Annotated Bibliography 575
18.1 Major Parsing Subjects 576
18.1.1 Unrestricted PS and CS Grammars 576
18.1.2 General Context-Free Parsing 576
18.1.3 LL Parsing 584
18.1.4 LR Parsing 585
18.1.5 Left-Corner Parsing 592
18.1.6 Precedence and Bounded-Right-Context Parsing 593
18.1.7 Finite-State Automata 596
18.1.8 General Books and Papers on Parsing 599
18.2 Advanced Parsing Subjects 601
18.2.1 Generalized Deterministic Parsing 601
18.2.2 Non-Canonical Parsing 605
18.2.3 Substring Parsing 609
18.2.4 Parsing as Intersection 611
18.2.5 Parallel Parsing Techniques 612
18.2.6 Non-Chomsky Systems 614
18.2.7 Error Handling 623
18.2.8 Incremental Parsing 629
18.3 Parsers and Applications 630
18.3.1 Parser Writing 630
18.3.2 Parser-Generating Systems 634
18.3.3 Applications 634
18.3.4 Parsing and Deduction 635
18.3.5 Parsing Issues in Natural Language Handling 636
18.4 Support Material 638
18.4.1 Formal Languages 638
18.4.2 Approximation Techniques 641
18.4.3 Transformations on Grammars 641
18.4.4 Miscellaneous Literature 642

A Hints and Solutions to Selected Problems 645

Author Index 651

Subject Index 655

1

Introduction

Parsing is the process of structuring a linear representation in accordance with a given grammar. This definition has been kept abstract on purpose to allow as wide an interpretation as possible. The "linear representation" may be a sentence, a computer program, a knitting pattern, a sequence of geological strata, a piece of music, actions in ritual behavior, in short any linear sequence in which the preceding elements in some way restrict[1] the next element. For some of the examples the grammar is well known, for some it is an object of research, and for some our notion of a grammar is only just beginning to take shape.

For each grammar, there are generally an infinite number of linear representations ("sentences") that can be structured with it. That is, a finite-size grammar can supply structure to an infinite number of sentences. This is the main strength of the grammar paradigm and indeed the main source of the importance of grammars: they summarize succinctly the structure of an infinite number of objects of a certain class.

There are several reasons to perform this structuring process called parsing. One reason derives from the fact that the obtained structure helps us to process the object further. When we know that a certain segment of a sentence is the subject, that information helps in understanding or translating the sentence. Once the structure of a document has been brought to the surface, it can be converted more easily.

A second reason is related to the fact that the grammar in a sense represents our understanding of the observed sentences: the better a grammar we can give for the movements of bees, the deeper our understanding is of them.

A third lies in the completion of missing information that parsers, and especially error-repairing parsers, can provide. Given a reasonable grammar of the language, an error-repairing parser can suggest possible word classes for missing or unknown words on clay tablets.

The reverse problem — given a (large) set of sentences, find the/a grammar which produces them — is called *grammatical inference*. Much less is known about it than about parsing, but progress is being made. The subject would require a complete book. Proceedings of the International Colloquiums on Grammatical Inference are published as Lecture Notes in Artificial Intelligence by Springer.

---

[1] If there is no restriction, the sequence still has a grammar, but this grammar is trivial and uninformative.

1.1 Parsing as a Craft

Parsing is no longer an arcane art; it has not been so since the early 1970s when Aho, Ullman, Knuth and many others put various parsing techniques solidly on their theoretical feet. It need not be a mathematical discipline either; the inner workings of a parser can be visualized, understood and modified to fit the application, with little more than cutting and pasting strings.

There is a considerable difference between a mathematician's view of the world and a computer scientist's. To a mathematician all structures are static: they have always been and will always be; the only time dependence is that we just have not discovered them all yet. The computer scientist is concerned with (and fascinated by) the continuous creation, combination, separation and destruction of structures: time is of the essence. In the hands of a mathematician, the Peano axioms create the integers without reference to time, but if a computer scientist uses them to implement integer addition, he finds they describe a very slow process, which is why he will be looking for a more efficient approach. In this respect the computer scientist has more in common with the physicist and the chemist; like them, he cannot do without a solid basis in several branches of applied mathematics, but, like them, he is willing (and often virtually obliged) to take on faith certain theorems handed to him by the mathematician. Without the rigor of mathematics all science would collapse, but not all inhabitants of a building need to know all the spars and girders that keep it upright. Factoring out certain detailed knowledge to specialists reduces the intellectual complexity of a task, which is one of the things computer science is about.

This is the vein in which this book is written: parsing for anybody who has parsing to do: the compiler writer, the linguist, the database interface writer, the geologist or musicologist who wants to test grammatical descriptions of their respective objects of interest, and so on. We require a good ability to visualize, some programming experience and the willingness and patience to follow non-trivial examples; there is nothing better for understanding a kangaroo than seeing it jump. We treat, of course, the popular parsing techniques, but we will not shun some weird techniques that look as if they are of theoretical interest only: they often offer new insights and a reader might find an application for them.

1.2 The Approach Used

This book addresses the reader at least three different levels. The interested non-computer scientist can read the book as "the story of grammars and parsing"; he or she can skip the detailed explanations of the algorithms: each algorithm is first explained in general terms. The computer scientist will find much technical detail on a wide array of algorithms. To the expert we offer a systematic bibliography of over

1700 entries. The printed book holds only those entries referenced in the book itself; the full list is available on the web site of this book. All entries in the printed book and about two-thirds of the entries in the web site list come with an annotation; this annotation, or summary, is unrelated to the abstract in the referred article, but rather provides a short explanation of the contents and enough material for the reader to decide if the referred article is worth reading.

No ready-to-run algorithms are given, except for the general context-free parser of Section 17.3. The formulation of a parsing algorithm with sufficient precision to enable a programmer to implement and run it without problems requires a considerable support mechanism that would be out of place in this book and in our experience does little to increase one's understanding of the process involved. The popular methods are given in algorithmic form in most books on compiler construction. The less widely used methods are almost always described in detail in the original publication, for which see Chapter 18.

1.3 Outline of the Contents

Since parsing is concerned with sentences and grammars and since grammars are themselves fairly complicated objects, ample attention is paid to them in Chapter 2. Chapter 3 discusses the principles behind parsing and gives a classification of parsing methods. In summary, parsing methods can be classified as top-down or bottom-up and as directional or non-directional; the directional methods can be further distinguished into deterministic and non-deterministic ones. This situation dictates the contents of the next few chapters.

In Chapter 4 we treat non-directional methods, including Unger and CYK. Chapter 5 forms an intermezzo with the treatment of finite-state automata, which are needed in the subsequent chapters. Chapters 6 through 10 are concerned with directional methods, as follows. Chapter 6 covers non-deterministic directional top-down parsers (recursive descent, Definite Clause Grammars), Chapter 7 non-deterministic directional bottom-up parsers (Earley). Deterministic methods are treated in Chapters 8 (top-down: LL in various forms) and 9 (bottom-up: LR methods). Chapter 10 covers non-canonical parsers, parsers that determine the nodes of a parse tree in a not strictly top-down or bottom-up order (for example left-corner). Non-deterministic versions of the above deterministic methods (for example the GLR parser) are described in Chapter 11.

The next four chapters are concerned with material that does not fit the above framework. Chapter 12 shows a number of recent techniques, both deterministic and non-deterministic, for parsing substrings of complete sentences in a language. Another recent development, in which parsing is viewed as intersecting a context-free grammar with a finite-state automaton is covered in Chapter 13. A few of the numerous parallel parsing algorithms are explained in Chapter 14, and a few of the numerous proposals for non-Chomsky language formalisms are explained in Chapter 15, with their parsers. That completes the parsing methods per se.

Error handling for a selected number of methods is treated in Chapter 16, and Chapter 17 discusses practical parser writing and use.

1.4 The Annotated Bibliography

The annotated bibliography is presented in Chapter 18 both in the printed book and, in a much larger version, on the web site of this book. It is an easily accessible and essential supplement of the main body of the book. Rather than listing all publications in author-alphabetic order, the bibliography is divided into a number of named sections, each concerned with a particular aspect of parsing; there are 25 of them in the printed book and 30 in the web bibliography. Within the sections, the publications are listed chronologically. An author index at the end of the book replaces the usual alphabetic list of publications. A numerical reference placed in brackets is used in the text to refer to a publication. For example, the annotated reference to Earley's publication of the Earley parser is indicated in the text by [14] and can be found on page 578, in the entry marked 14.

2

Grammars as a Generating Device

2.1 Languages as Infinite Sets

In computer science as in everyday parlance, a "grammar" serves to "describe" a "language". If taken at face value, this correspondence, however, is misleading, since the computer scientist and the naive speaker mean slightly different things by the three terms. To establish our terminology and to demarcate the universe of discourse, we shall examine the above terms, starting with the last one.

2.1.1 Language

To the larger part of mankind, language is first and foremost a means of communication, to be used almost unconsciously, certainly so in the heat of a debate. Communication is brought about by sending messages, through air vibrations or through written symbols. Upon a closer look the language messages ("utterances") fall apart into sentences, which are composed of words, which in turn consist of symbol sequences when written. Languages can differ on all three levels of composition. The script can be slightly different, as between English and Irish, or very different, as between English and Chinese. Words tend to differ greatly, and even in closely related languages people call *un cheval* or *ein Pferd*, that which is known to others as *a horse*. Differences in sentence structure are often underestimated; even the closely related Dutch often has an almost Shakespearean word order: "*Ik geloof je niet*", "*I believe you not*", and more distantly related languages readily come up with constructions like the Hungarian "*Pénzem van*", "*Money-my is*", where the English say "*I have money*".

The computer scientist takes a very abstracted view of all this. Yes, a language has sentences, and these sentences possess structure; whether they communicate something or not is not his concern, but information may possibly be derived from their structure and then it is quite all right to call that information the "meaning" of the sentence. And yes, sentences consist of words, which he calls "tokens", each possibly carrying a piece of information, which is its contribution to the meaning of the whole sentence. But no, words cannot be broken down any further. This does not worry the computer scientist. With his love of telescoping solutions and multi-level techniques, he blithely claims that if words turn out to have structure after all, they are sentences in a different language, of which the letters are the tokens.

The practitioner of formal linguistics, henceforth called the formal-linguist (to distinguish him from the "formal linguist", the specification of whom is left to the imagination of the reader) again takes an abstracted view of this. A language is a "set" of sentences, and each sentence is a "sequence" of "symbols"; that is all there is: no meaning, no structure, either a sentence belongs to the language or it does not. The only property of a symbol is that it has an identity; in any language there are a certain number of different symbols, the *alphabet*, and that number must be finite. Just for convenience we write these symbols as *a*, *b*, *c*, . . . , but ✆, ✈, ❐, . . . would do equally well, as long as there are enough symbols. The word *sequence* means that the symbols in each sentence are in a fixed order and we should not shuffle them. The word *set* means an unordered collection with all the duplicates removed. A set can be written down by writing the objects in it, surrounded by curly brackets. All this means that to the formal-linguist the following is a language: *a, b, ab, ba*, and so is {*a, aa, aaa, aaaa,* . . . } although the latter has notational problems that will be solved later. In accordance with the correspondence that the computer scientist sees between sentence/word and word/letter, the formal-linguist also calls a sentence a *word* and he says that "the word *ab* is in the language {*a, b, ab, ba*}".

Now let us consider the implications of these compact but powerful ideas.

To the computer scientist, a language is a probably infinitely large set of sentences, each composed of tokens in such a way that it has structure; the tokens and the structure cooperate to describe the semantics of the sentence, its "meaning" if you will. Both the structure and the semantics are new, that is, were not present in the formal model, and it is his responsibility to provide and manipulate them both. To a computer scientist $3+4\times5$ is a sentence in the language of "arithmetics on single digits" ("single digits" to avoid having an infinite number of symbols); its structure can be shown by inserting parentheses: $(3+(4\times5))$; and its semantics is probably 23.

To the linguist, whose view of languages, it has to be conceded, is much more normal than that of either of the above, a language is an infinite set of possibly interrelated sentences. Each sentence consists, in a structured fashion, of words which have a meaning in the real world. Structure and words together give the sentence a meaning, which it communicates. Words, again, possess structure and are composed of letters; the letters cooperate with some of the structure to give a meaning to the word. The heavy emphasis on semantics, the relation with the real world and the integration of the two levels sentence/word and word/letters are the domain of the linguist. "*The circle spins furiously*" is a sentence, "*The circle sleeps red*" is nonsense.

The formal-linguist holds his views of language because he wants to study the fundamental properties of languages in their naked beauty; the computer scientist holds his because he wants a clear, well-understood and unambiguous means of describing objects in the computer and of communication with the computer, a most exacting communication partner, quite unlike a human; and the linguist holds his view of language because it gives him a formal tight grip on a seemingly chaotic and perhaps infinitely complex object: natural language.

2.1.2 Grammars

Everyone who has studied a foreign language knows that a grammar is a book of rules and examples which describes and teaches the language. Good grammars make a careful distinction between the sentence/word level, which they often call *syntax* or *syntaxis* and the word/letter level, which they call *morphology*. Syntax contains rules like "*pour que* is followed by the subjunctive, but *parce que* is not". Morphology contains rules like "the plural of an English noun is formed by appending an *-s*, except when the word ends in *-s*, *-sh*, *-o*, *-ch* or *-x*, in which case *-es* is appended, or when the word has an irregular plural."

We skip the computer scientist's view of a grammar for the moment and proceed immediately to that of the formal-linguist. His view is at the same time very abstract and quite similar to the layman's: a grammar is any exact, finite-size, complete description of the language, i.e., of the set of sentences. This is in fact the school grammar, with the fuzziness removed. Although it will be clear that this definition has full generality, it turns out that it is too general, and therefore relatively powerless. It includes descriptions like "the set of sentences that could have been written by Chaucer"; platonically speaking this defines a set, but we have no way of creating this set or testing whether a given sentence belongs to this language. This particular example, with its "could have been" does not worry the formal-linguist, but there are examples closer to his home that do. "The longest block of consecutive sevens in the decimal expansion of π" describes a language that has at most one word in it (and then that word will consist of sevens only), and as a definition it is exact, of finite-size and complete. One bad thing with it, however, is that one cannot find this word: suppose one finds a block of one hundred sevens after billions and billions of digits, there is always a chance that further on there is an even longer block. And another bad thing is that one cannot even know if this longest block exists at all. It is quite possible that, as one proceeds further and further up the decimal expansion of π, one would find longer and longer stretches of sevens, probably separated by ever-increasing gaps. A comprehensive theory of the decimal expansion of π might answer these questions, but no such theory exists.

For these and other reasons, the formal-linguists have abandoned their static, platonic view of a grammar for a more constructive one, that of the generative grammar: a *generative grammar* is an exact, fixed-size recipe for constructing the sentences in the language. This means that, following the recipe, it must be possible to construct each sentence of the language (in a finite number of actions) and no others. This does not mean that, given a sentence, the recipe tells us *how* to construct that particular sentence, only that it is possible to do so. Such recipes can have several forms, of which some are more convenient than others.

The computer scientist essentially subscribes to the same view, often with the additional requirement that the recipe should imply how a sentence can be constructed.

2.1.3 Problems with Infinite Sets

The above definition of a language as a possibly infinite set of sequences of symbols and of a grammar as a finite recipe to generate these sentences immediately gives rise to two embarrassing questions:

1. How can finite recipes generate enough infinite sets of sentences?
2. If a sentence is just a sequence and has no structure and if the meaning of a sentence derives, among other things, from its structure, how can we assess the meaning of a sentence?

These questions have long and complicated answers, but they do have answers. We shall first pay some attention to the first question and then devote the main body of this book to the second.

2.1.3.1 Infinite Sets from Finite Descriptions

In fact there is nothing wrong with getting a single infinite set from a single finite description: "the set of all positive integers" is a very finite-size description of a definitely infinite-size set. Still, there is something disquieting about the idea, so we shall rephrase our question: "Can all languages be described by finite descriptions?" As the lead-up already suggests, the answer is "No", but the proof is far from trivial. It is, however, very interesting and famous, and it would be a shame not to present at least an outline of it here.

2.1.3.2 Descriptions can be Enumerated

The proof is based on two observations and a trick. The first observation is that descriptions can be listed and given a number. This is done as follows. First, take all descriptions of size one, that is, those of only one letter long, and sort them alphabetically. This is the beginning of our list. Depending on what, exactly, we accept as a description, there may be zero descriptions of size one, or 27 (all letters + space), or 95 (all printable ASCII characters) or something similar; this is immaterial to the discussion which follows.

Second, we take all descriptions of size two, sort them alphabetically to give the second chunk on the list, and so on for lengths 3, 4 and further. This assigns a position on the list to each and every description. Our description "the set of all positive integers", for example, is of size 32, not counting the quotation marks. To find its position on the list, we have to calculate how many descriptions there are with less than 32 characters, say L. We then have to generate all descriptions of size 32, sort them and determine the position of our description in it, say P, and add the two numbers L and P. This will, of course, give a huge number[1] but it does ensure that the description is on the list, in a well-defined position; see Figure 2.1.

---

[1] Some calculations tell us that, under the ASCII-128 assumption, the number is 248 17168 89636 37891 49073 14874 06454 89259 38844 52556 26245 57755 89193 30291, or roughly 2.5×10^{67}.

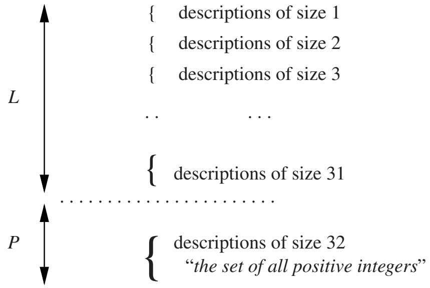

{ descriptions of size 1
{ descriptions of size 2
{ descriptions of size 3
L
. . . . .
{ descriptions of size 31
P
{ descriptions of size 32
"the set of all positive integers"

Fig. 2.1. List of all descriptions of length 32 or less

Two things should be pointed out here. The first is that just listing all descriptions alphabetically, without reference to their lengths, would not do: there are already infinitely many descriptions starting with an "a" and no description starting with a higher letter could get a number on the list. The second is that there is no need to actually do all this. It is just a thought experiment that allows us to examine and draw conclusions about the behavior of a system in a situation which we cannot possibly examine physically.

Also, there will be many nonsensical descriptions on the list; it will turn out that this is immaterial to the argument. The important thing is that all meaningful descriptions are on the list, and the above argument ensures that.

2.1.3.3 Languages are Infinite Bit-Strings

We know that words (sentences) in a language are composed of a finite set of symbols; this set is called quite reasonably the "alphabet". We will assume that the symbols in the alphabet are ordered. Then the words in the language can be ordered too. We shall indicate the alphabet by Σ.

Now the simplest language that uses alphabet Σ is that which consists of all words that can be made by combining letters from the alphabet. For the alphabet $\Sigma = \{a, b\}$ we get the language $\{\ , a, b, aa, ab, ba, bb, aaa, \ldots\}$. We shall call this language Σ^*, for reasons to be explained later; for the moment it is just a name.

The set notation Σ^* above started with "$\{\ , a,$", a remarkable construction; the first word in the language is the *empty word*, the word consisting of zero *a*s and zero *b*s. There is no reason to exclude it, but, if written down, it may easily be overlooked, so we shall write it as ε (epsilon), regardless of the alphabet. So, $\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$. In some natural languages, forms of the present tense of the verb "to be" are empty words, giving rise to sentences of the form "I student", meaning "I am a student." Russian and Hebrew are examples of this.

Since the symbols in the alphabet Σ are ordered, we can list the words in the language Σ^*, using the same technique as in the previous section: First, all words of size zero, sorted; then all words of size one, sorted; and so on. This is actually the order already used in our set notation for Σ^*.

The language Σ^* has the interesting property that all languages using alphabet Σ are subsets of it. That means that, given another possibly less trivial language over Σ, called L, we can go through the list of words in Σ^* and put ticks on all words that are in L. This will cover all words in L, since Σ^* contains any possible word over Σ.

Suppose our language L is “the set of all words that contain more *a*s than *b*s”. L is the set $\{a, aa, aab, aba, baa, \dots\}$. The beginning of our list, with ticks, will look as follows:

	ε
✔	*a*
	b
✔	*aa*
	ab
	ba
	bb
✔	*aaa*
✔	*aab*
✔	*aba*
	abb
✔	*baa*
	bab
	bba
	bbb
✔	*aaaa*
...	...

Given the alphabet with its ordering, the list of blanks and ticks alone is entirely sufficient to identify and describe the language. For convenience we write the blank as a 0 and the tick as a 1 as if they were bits in a computer, and we can now write $L = 0101000111010001\cdots$ (and $\Sigma^* = 1111111111111111\cdots$). It should be noted that this is true for *any* language, be it a formal language like L, a programming language like Java or a natural language like English. In English, the 1s in the bit-string will be very scarce, since hardly any arbitrary sequence of words is a good English sentence (and hardly any arbitrary sequence of letters is a good English word, depending on whether we address the sentence/word level or the word/letter level).

2.1.3.4 Diagonalization

The previous section attaches the infinite bit-string $0101000111010001\cdots$ to the description “the set of all the words that contain more *a*s than *b*s”. In the same vein we can attach such bit-strings to all descriptions. Some descriptions may not yield a language, in which case we can attach an arbitrary infinite bit-string to it. Since all descriptions can be put on a single numbered list, we get, for example, the following picture:

Description	Language
Description #1	$000000100\cdots$
Description #2	$110010001\cdots$
Description #3	$011011010\cdots$
Description #4	$110011010\cdots$
Description #5	$100000011\cdots$
Description #6	$111011011\cdots$
...	...

At the left we have all descriptions, at the right all languages they describe. We now claim that many languages exist that are not on the list of languages above: the above list is far from complete, although the list of descriptions is complete. We shall prove this by using the diagonalization process ("Diagonalverfahren") of Cantor.

Consider the language $C = 100110\cdots$, which has the property that its n-th bit is unequal to the n-th bit of the language described by Description #n. The first bit of C is a 1, because the first bit for Description #1 is a 0; the second bit of C is a 0, because the second bit for Description #2 is a 1, and so on. C is made by walking the NW to SE diagonal of the language field and copying the opposites of the bits we meet. This is the diagonal in Figure 2.2(*a*). The language C cannot be on the list! It cannot be on line 1, since its first bit differs (is made to differ, one should say) from that on line 1, and in general it cannot be on line n, since its n-th bit will differ from that on line n, by definition.

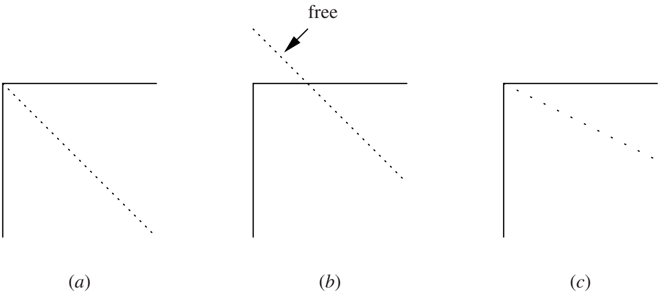


Fig. 2.2. "Diagonal" languages along n (*a*), $n+10$ (*b*), and $2n$ (*c*)

So, in spite of the fact that we have exhaustively listed all possible finite descriptions, we have at least one language that has no description on the list. But there exist more languages that are not on the list. Construct, for example, the language whose $n+10$-th bit differs from the $n+10$-th bit in Description #n. Again it cannot be on the list since for every $n > 0$ it differs from line n in the $n+10$-th bit. But that means that bits $1\ldots9$ play no role, and can be chosen arbitrarily, as shown in Figure 2.2(*b*); this yields another $2^9 = 512$ languages that are not on the list. And we can do even much better than that! Suppose we construct a language whose $2n$-th bit differs from the $2n$-th bit in Description #n (*c*). Again it is clear that it cannot be on the list, but now every odd bit is left unspecified and can be chosen freely! This allows us to create freely an infinite number of languages none of which allows a finite description; see the slanting diagonal in Figure 2.2. In short, for every language that can be described there are infinitely many that cannot.

The diagonalization technique is described more formally in most books on theoretical computer science; see e.g., Rayward-Smith [393, pp. 5-6], or Sudkamp [397, Section 1.4].

2.1.3.5 Discussion

The above demonstration shows us several things. First, it shows the power of treating languages as formal objects. Although the above outline clearly needs considerable amplification and substantiation to qualify as a proof (for one thing it still has to be clarified why the above explanation, which defines the language C, is not itself on the list of descriptions; see Problem 2.1, it allows us to obtain insight into properties not otherwise assessable.

Secondly, it shows that we can only describe a tiny subset (not even a fraction) of all possible languages: there is an infinity of languages out there, forever beyond our reach.

Thirdly, we have proved that, although there are infinitely many descriptions and infinitely many languages, these infinities are not equal to each other, and the latter is larger than the former. These infinities are called $\aleph_0$ and $\aleph_1$ by Cantor, and the above is just a special case of his proof that $\aleph_0 < \aleph_1$.

2.1.4 Describing a Language through a Finite Recipe

A good way to build a set of objects is to start with a small object and to give rules for how to add to it and construct new objects from it. “Two is an even number and the sum of two even numbers is again an even number” effectively generates the set of all even numbers. Formalists will add “and no other numbers are even”, but we will take that as understood.

Suppose we want to generate the set of all enumerations of names, of the type “Tom, Dick and Harry”, in which all names but the last two are separated by commas. We will not accept “Tom, Dick, Harry” nor “Tom and Dick and Harry”, but we shall not object to duplicates: “Grubb, Grubb and Burrowes”[2] is all right. Although these are not complete sentences in normal English, we shall still call them “sentences” since that is what they are in our midget language of name enumerations. A simple-minded recipe would be:

0. Tom is a name, Dick is a name, Harry is a name;
1. a name is a sentence;
2. a sentence followed by a comma and a name is again a sentence;
3. before finishing, if the sentence ends in “, name”, replace it by “and name”.

---

[2] *The Hobbit*, by J.R.R. Tolkien, Allen and Unwin, 1961, p. 311.

Although this will work for a cooperative reader, there are several things wrong with it. Clause 3 is especially wrought with trouble. For example, the sentence does not really end in “, name”, it ends in “, Dick” or such, and “name” is just a symbol that stands for a real name; such symbols cannot occur in a real sentence and must in the end be replaced by a real name as given in clause 0. Likewise, the word “sentence” in the recipe is a symbol that stands for all the actual sentences. So there are two kinds of symbols involved here: real symbols, which occur in finished sentences, like “Tom”, “Dick”, a comma and the word “and”; and there are intermediate symbols, like “sentence” and “name” that cannot occur in a finished sentence. The first kind corresponds to the words or *tokens* explained above; the technical term for them is *terminal symbols* (or *terminals* for short). The intermediate symbols are called *non-terminals*, a singularly uninspired term. To distinguish them, we write terminals in lower case letters and start non-terminals with an upper case letter. Non-terminals are called *(grammar) variables* or *syntactic categories* in linguistic contexts.

To stress the generative character of the recipe, we shall replace “X is a Y” by “Y may be replaced by X”: if “tom” is an instance of a Name, then everywhere we have a Name we may narrow it down to “tom”. This gives us:

0. Name may be replaced by “tom”
   Name may be replaced by “dick”
   Name may be replaced by “harry”
1. Sentence may be replaced by Name
2. Sentence may be replaced by Sentence, Name
3. “, Name” at the end of a Sentence must be replaced by “and Name” before Name is replaced by any of its replacements
4. a sentence is finished only when it no longer contains non-terminals
5. we start our replacement procedure with Sentence

Clause 0 through 3 describe replacements, but 4 and 5 are different. Clause 4 is not specific to this grammar. It is valid generally and is one of the rules of the game. Clause 5 tells us where to start generating. This name is quite naturally called the *start symbol*, and it is required for every grammar.

Clause 3 still looks worrisome; most rules have “may be replaced”, but this one has “must be replaced”, and it refers to the “end of a Sentence”. The rest of the rules work through replacement, but the problem remains how we can use replacement to test for the end of a Sentence. This can be solved by adding an end marker after it. And if we make the end marker a non-terminal which cannot be used anywhere except in the required replacement from “, Name” to “and Name”, we automatically enforce the restriction that no sentence is finished unless the replacement test has taken place. For brevity we write → instead of “may be replaced by”; since terminal and non-terminal symbols are now identified as technical objects we shall write them in a typewriter-like typeface. The part before the → is called the *left-hand side*, the part after it the *right-hand side*. This results in the recipe in Figure 2.3.

This is a simple and relatively precise form for a recipe, and the rules are equally straightforward: start with the start symbol, and keep replacing until there are no non-terminals left.

```
0.          Name  ->  tom
            Name  ->  dick
            Name  ->  harry
1.      Sentence  ->  Name
        Sentence  ->  List End
2.          List  ->  Name
            List  ->  List , Name
3.    , Name End  ->  and Name
4.  the start symbol is  Sentence
```

Fig. 2.3. A finite recipe for generating strings in the t, d & h language

2.2 Formal Grammars

The above recipe form, based on replacement according to rules, is strong enough to serve as a basis for formal grammars. Similar forms, often called "rewriting systems", have a long history among mathematicians, and were already in use several centuries B.C. in India (see, for example, Bhate and Kak [411]). The specific form of Figure 2.3 was first studied extensively by Chomsky [385]. His analysis has been the foundation for almost all research and progress in formal languages, parsers and a considerable part of compiler construction and linguistics.

2.2.1 The Formalism of Formal Grammars

Since formal languages are a branch of mathematics, work in this field is done in a special notation. To show some of its flavor, we shall give the formal definition of a grammar and then explain why it describes a grammar like the one in Figure 2.3. The formalism used is indispensable for correctness proofs, etc., but not for understanding the principles; it is shown here only to give an impression and, perhaps, to bridge a gap.

Definition 2.1: A *generative grammar* is a 4-tuple (V_N, V_T, R, S) such that

(1) V_N and V_T are finite sets of symbols,
(2) $V_N \cap V_T = \emptyset$,
(3) R is a set of pairs (P, Q) such that
    (3a) $P \in (V_N \cup V_T)^+$ and
    (3b) $Q \in (V_N \cup V_T)^*$,
(4) $S \in V_N$

A *4-tuple* is just an object consisting of 4 identifiable parts; they are the nonterminals, the terminals, the rules and the start symbol, in that order. The above definition does not tell this, so this is for the teacher to explain. The set of nonterminals is named V_N and the set of terminals V_T. For our grammar we have:

V_N = {`Name`, `Sentence`, `List`, `End`}
V_T = {`tom`, `dick`, `harry`, `,`, `and`}

(note the `,` in the set of terminal symbols).

The intersection of V_N and V_T (2) must be empty, indicated by the symbol for the empty set, $\emptyset$. So the non-terminals and the terminals may not have a symbol in common, which is understandable.

R is the set of all rules (3), and P and Q are the left-hand sides and right-hand sides, respectively. Each P must consist of sequences of one or more non-terminals and terminals and each Q must consist of sequences of zero or more non-terminals and terminals. For our grammar we have:

$$\begin{aligned}
&R = \{(\texttt{Name}, \texttt{tom}), (\texttt{Name}, \texttt{dick}), (\texttt{Name}, \texttt{harry}),\\
&(\texttt{Sentence}, \texttt{Name}), (\texttt{Sentence}, \texttt{List End}), (\texttt{List}, \texttt{Name}),\\
&(\texttt{List}, \texttt{List , Name}), (\texttt{, Name End}, \texttt{and Name})\}
\end{aligned}$$

Note again the two different commas.

The start symbol S must be an element of V_N, that is, it must be a non-terminal:

$$S = \texttt{Sentence}$$

This concludes our field trip into formal linguistics. In short, the mathematics of formal languages is a language, a language that has to be learned; it allows very concise expression of *what* and *how* but gives very little information on *why*. Consider this book a translation and an exegesis.

2.2.2 Generating Sentences from a Formal Grammar

The grammar in Figure 2.3 is what is known as a *phrase structure grammar* for our `t,d&h` language (often abbreviated to *PS grammar*). There is a more compact notation, in which several right-hand sides for one and the same left-hand side are grouped together and then separated by vertical bars, |. This bar belongs to the formalism, just as the arrow →, and can be read “or else”. The right-hand sides separated by vertical bars are also called *alternatives*. In this more concise form our grammar becomes

$$\begin{array}{lrcl}
0. & \texttt{Name} & \rightarrow & \texttt{tom} \mid \texttt{dick} \mid \texttt{harry}\\
1. & \texttt{Sentence}_{\mathrm{S}} & \rightarrow & \texttt{Name} \mid \texttt{List End}\\
2. & \texttt{List} & \rightarrow & \texttt{Name} \mid \texttt{Name , List}\\
3. & \texttt{, Name End} & \rightarrow & \texttt{and Name}
\end{array}$$

where the non-terminal with the subscript $_{\mathrm{S}}$ is the start symbol. (The subscript identifies the symbol, not the rule.)

Now let us generate our initial example from this grammar, using replacement according to the above rules only. We obtain the following successive forms for `Sentence`:

Intermediate form	Rule used	Explanation
`Sentence`		the start symbol
`List End`	`Sentence → List End`	rule 1
`Name , List End`	`List → Name , List`	rule 2
`Name , Name , List End`	`List → Name , List`	rule 2
`Name , Name , Name End`	`List → Name`	rule 2
`Name , Name and Name`	`, Name End → and Name`	rule 3
`tom , dick and harry`		rule 0, three times

The intermediate forms are called *sentential forms*. If a sentential form contains no non-terminals it is called a *sentence* and belongs to the generated language. The transitions from one line to the next are called *production steps* and the rules are called *production rules*, for obvious reasons.

The production process can be made more visual by drawing connective lines between corresponding symbols, using a “graph”. A *graph* is a set of *nodes* connected by a set of *edges*. A node can be thought of as a point on paper, and an edge as a line, where each line connects two points; one point may be the end point of more than one line. The nodes in a graph are usually “labeled”, which means that they have been given names, and it is convenient to draw the nodes on paper as bubbles with their names in them, rather than as points. If the edges are arrows, the graph is a *directed graph*; if they are lines, the graph is *undirected*. Almost all graphs used in parsing techniques are directed.

The graph corresponding to the above production process is shown in Figure 2.4. Such a picture is called a *production graph* or *syntactic graph* and depicts the syntactic structure (with regard to the given grammar) of the final sentence. We see that the production graph normally fans out downwards, but occasionally we may see starlike constructions, which result from rewriting a group of symbols.

A *cycle* in a graph is a path from a node *N* following the arrows, leading back to *N*. A production graph cannot contain cycles; we can see that as follows. To get a cycle we would need a non-terminal node *N* in the production graph that has produced children that are directly or indirectly *N* again. But since the production process always makes new copies for the nodes it produces, it cannot produce an already existing node. So a production graph is always “acyclic”; *d*irected *a*cyclic *g*raphs are called *dags*. for d(irected)a(cyclic)g(raph)s

It is patently impossible to have the grammar generate **`tom, dick, harry`**, since any attempt to produce more than one name will drag in an **`End`** and the only way to get rid of it again (and get rid of it we must, since it is a non-terminal) is to have it absorbed by rule 3, which will produce the **`and`**. Amazingly, we have succeeded in implementing the notion “must replace” in a system that only uses “may replace”; looking more closely, we see that we have split “must replace” into “may replace” and “must not be a non-terminal”.

Apart from our standard example, the grammar will of course also produce many other sentences; examples are

```
harry and tom
harry
tom, tom, tom and tom
```

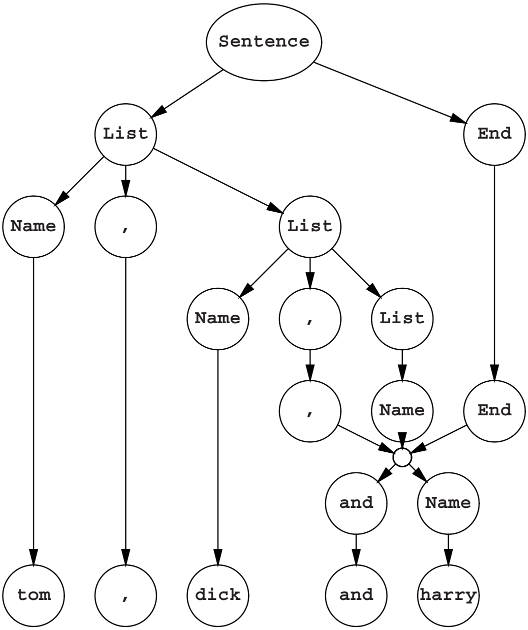


Fig. 2.4. Production graph for a sentence

and an infinity of others. A determined and foolhardy attempt to generate the incorrect form without the **`and`** will lead us to sentential forms like

`tom, dick, harry End`

which are not sentences and to which no production rule applies. Such forms are called *blind alleys*. As the right arrow in a production rule already suggests, the rule may not be applied in the reverse direction.

2.2.3 The Expressive Power of Formal Grammars

The main property of a formal grammar is that it has production rules, which may be used for rewriting part of the sentential form (= sentence under construction) and a starting symbol which is the mother of all sentential forms. In the production rules we find non-terminals and terminals; finished sentences contain terminals only. That is about it: the rest is up to the creativity of the grammar writer and the sentence producer.

This is a framework of impressive frugality and the question immediately rises: Is it sufficient? That is hard to say, but if it is not, we do not have anything more expressive. Strange as it may sound, all other methods known to mankind for generating sets have been proved to be equivalent to or less powerful than a phrase structure grammar. One obvious method for generating a set is, of course, to write a program generating it, but it has been proved that any set that can be generated by a program can be generated by a phrase structure grammar. There are even more arcane methods, but all of them have been proved not to be more expressive. On the other hand there is no proof that no such stronger method can exist. But in view of the fact that many quite different methods all turn out to halt at the same barrier, it is highly unlikely[3] that a stronger method will ever be found. See, e.g. Révész [394, pp 100-102].

As a further example of the expressive power we shall give a grammar for the movements of a Manhattan turtle. A *Manhattan turtle* moves in a plane and can only move north, east, south or west in distances of one block. The grammar of Figure 2.5 produces all paths that return to their own starting point. As to rule 2, it should be noted that many authors require at least one of the symbols in the left-hand side to be a non-terminal. This restriction can always be enforced by adding new non-terminals.

```
1.         Move_s  -->  north Move south | east Move west | ε
2.     north east  -->  east north
      north south  -->  south north
       north west  -->  west north
       east north  -->  north east
       east south  -->  south east
        east west  -->  west east
      south north  -->  north south
       south east  -->  east south
       south west  -->  west south
       west north  -->  north west
        west east  -->  east west
       west south  -->  south west
```

Fig. 2.5. Grammar for the movements of a Manhattan turtle

The simple round trip `north east south west` is produced as shown in Figure 2.6 (names abbreviated to their first letter). Note the empty alternative in rule 1 (the ε), which results in the dying out of the third **M** in the above production graph.

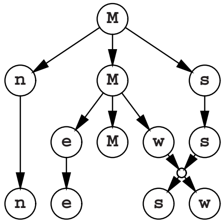


Fig. 2.6. How the grammar of Figure 2.5 produces a round trip

---

[3] Paul Vitány has pointed out that if scientists call something "highly unlikely" they are still generally not willing to bet a year's salary on it, double or quit.

2.3 The Chomsky Hierarchy of Grammars and Languages

The grammars from Figures 2.3 and 2.5 are easy to understand and indeed some simple phrase structure grammars generate very complicated sets. The grammar for any given set is, however, usually far from simple. (We say "*The* grammar for a given set" although there can be, of course, infinitely many grammars for a set. By *the* grammar for a set, we mean any grammar that does the job and is not obviously overly complicated.) Theory says that if a set can be generated at all (for example, by a program) it can be generated by a phrase structure grammar, but theory does not say that it will be easy to do so, or that the grammar will be understandable. In this context it is illustrative to try to write a grammar for those Manhattan turtle paths in which the turtle is never allowed to the west of its starting point (Problem 2.3).

Apart from the intellectual problems phrase structure grammars pose, they also exhibit fundamental and practical problems. We shall see that no general parsing algorithm for them can exist, and all known special parsing algorithms are either very inefficient or very complex; see Section 3.4.2.

The desire to restrict the unmanageability of phrase structure grammars, while keeping as much of their generative powers as possible, has led to the *Chomsky hierarchy* of grammars. This hierarchy distinguishes four types of grammars, numbered from 0 to 3; it is useful to include a fifth type, called Type 4 here. Type 0 grammars are the (unrestricted) phrase structure grammars of which we have already seen examples. The other types originate from applying more and more restrictions to the allowed form of the rules in the grammar. Each of these restrictions has far-reaching consequences; the resulting grammars are gradually easier to understand and to manipulate, but are also gradually less powerful. Fortunately, these less powerful types are still very useful, actually more useful even than Type 0.

We shall now consider each of the three remaining types in turn, followed by a trivial but useful fourth type.

For an example of a completely different method of generating Type 0 languages see Geffert [395].

2.3.1 Type 1 Grammars

The characteristic property of a Type 0 grammar is that it may contain rules that transform an arbitrary (non-zero) number of symbols into an arbitrary (possibly zero) number of symbols. Example:

`, N E --> and N`

in which three symbols are replaced by two. By restricting this freedom, we obtain Type 1 grammars. Strangely enough there are two, intuitively completely different definitions of Type 1 grammars, which can be easily proved to be equivalent.

2.3.1.1 Two Types of Type 1 Grammars

A grammar is *Type 1 monotonic* if it contains no rules in which the left-hand side consists of more symbols than the right-hand side. This forbids, for example, the rule `, N E → and N`.

A grammar is *Type 1 context-sensitive* if all of its rules are context-sensitive. A rule is *context-sensitive* if actually only one (non-terminal) symbol in its left-hand side gets replaced by other symbols, while we find the others back, undamaged and in the same order, in the right-hand side. Example:

```
Name Comma Name End → Name and Name End
```

which tells that the rule

```
Comma → and
```

may be applied if the left context is `Name` and the right context is `Name End`. The contexts themselves are not affected. The replacement must be at least one symbol long. This means that context-sensitive grammars are always monotonic; see Section 2.5.

Here is a monotonic grammar for our `t,d&h` example. In writing monotonic grammars one has to be careful never to produce more symbols than will eventually be produced. We avoid the need to delete the end marker by incorporating it into the rightmost name:

```
      Name  →  tom | dick | harry
Sentence_S  →  Name | List
      List  →  EndName | Name , List
 , EndName  →  and Name
```

where `EndName` is a single symbol.

And here is a context-sensitive grammar for it.

```
         Name  →  tom | dick | harry
   Sentence_S  →  Name | List
         List  →  EndName
               |  Name Comma List
Comma EndName  →  and EndName           context is ... EndName
  and EndName  →  and Name              context is and ...
        Comma  →  ,
```

Note that we need an extra non-terminal `Comma` to produce the terminal `and` in the correct context.

Monotonic and context-sensitive grammars are equally powerful: for each language that can be generated by a monotonic grammar a context-sensitive grammar exists that generates the same language, and vice versa. They are less powerful than the Type 0 grammars, that is, there are languages that can be generated by a Type 0 grammar but not by any Type 1. Strangely enough no simple examples of such languages are known. Although the difference between Type 0 and Type 1 is fundamental and is not just a whim of Mr. Chomsky, grammars for which the difference matters are too complicated to write down; only their existence can be proved (see e.g., Hopcroft and Ullman [391, pp. 183-184], or Révész [394, p. 98]).

Of course any Type 1 grammar is also a Type 0 grammar, since the class of Type 1 grammars is obtained from the class of Type 0 grammars by applying restrictions. But it would be confusing to call a Type 1 grammar a Type 0 grammar; it would be like calling a cat a mammal: correct but imprecise. A grammar is named after the smallest class (that is, the highest type number) in which it will still fit.

We saw that our **`t,d&h`** language, which was first generated by a Type 0 grammar, could also be generated by a Type 1 grammar. We shall see that there is also a Type 2 and a Type 3 grammar for it, but no Type 4 grammar. We therefore say that the **`t,d&h`** language is a Type 3 language, after the most restricted (and simple and amenable) grammar for it. Some corollaries of this are: A Type n language can be generated by a Type n grammar or anything stronger, but not by a weaker Type $n+1$ grammar; and: If a language is generated by a Type n grammar, that does not necessarily mean that there is no (weaker) Type $n+1$ grammar for it. The use of a Type 0 grammar for our **`t,d&h`** language was a serious case of overkill, just for demonstration purposes.

2.3.1.2 Constructing a Type 1 Grammar

The standard example of a Type 1 language is the set of words that consist of equal numbers of **`a`**s, **`b`**s and **`c`**s, in that order:

$$\underbrace{a\,a\,....\,a}_{n\text{ of them}} \quad \underbrace{b\,b\,....\,b}_{n\text{ of them}} \quad \underbrace{c\,c\,....\,c}_{n\text{ of them}}$$

For the sake of completeness and to show how one writes a Type 1 grammar if one is clever enough, we shall now derive a grammar for this toy language. Starting with the simplest case, we have the rule

0. **`S → abc`**

Having obtained one instance of **`S`**, we may want to prepend more **`a`**s to the beginning; if we want to remember how many there were, we shall have to append something to the end as well at the same time, and that cannot be a **`b`** or a **`c`**. We shall use a yet unknown symbol **`Q`**. The following rule both prepends and appends:

1. **`S → aSQ`**

If we apply this rule, for example, three times, we get the sentential form

`aaabcQQ`

Now, to get **`aaabbbccc`** from this, each **`Q`** must be worth one **`b`** and one **`c`**, as expected, but we cannot just write

`Q → bc`

because that would allow `b`s after the first `c`. The above rule would, however, be all right if it were allowed to do replacement only between a `b` on the left and a `c` on the right. There the newly inserted `bc` will do no harm:

2. `bQc` → `bbcc`

Still, we cannot apply this rule since normally the `Q`s are to the right of the `c`. This can be remedied by allowing a `Q` to hop left over a `c`:

3. `cQ` → `Qc`

We can now finish our derivation:

`aaabcQQ`	(3 times rule 1)
`aaabQcQ`	(rule 3)
`aaabbccQ`	(rule 2)
`aaabbcQc`	(rule 3)
`aaabbQcc`	(rule 3)
`aaabbbccc`	(rule 2)

It should be noted that the above derivation only shows that the grammar will produce the right strings, and the reader will still have to convince himself that it will not generate other and incorrect strings (Problem 2.4).

The grammar is summarized in Figure 2.7. Since a derivation graph of $\mathtt{a}^3\mathtt{b}^3\mathtt{c}^3$ is already rather unwieldy, a derivation graph for $\mathtt{a}^2\mathtt{b}^2\mathtt{c}^2$ is given in Figure 2.8. The grammar is monotonic and therefore of Type 1. It can be proved that there is no Type 2 grammar for the language; see Section 2.7.1.

1. $\mathtt{S}_{\mathrm{S}}$ → `abc` | `aSQ`
2. `bQc` → `bbcc`
3. `cQ` → `Qc`

Fig. 2.7. Monotonic grammar for $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$

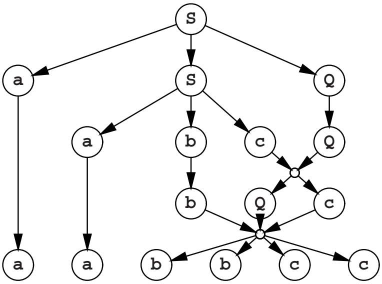


Fig. 2.8. Derivation of `aabbcc`

Although only context-sensitive Type 1 grammars can by rights be called context-sensitive grammars (CS grammars), that name is often used even if the grammar is actually monotonic Type 1. There are no standard initials for monotonic, but MT will do.

2.3.2 Type 2 Grammars

Type 2 grammars are called *context-free grammars* (CF grammars) and their relation to context-sensitive grammars is as direct as the name suggests. A context-free grammar is like a context-sensitive grammar, except that both the left and the right contexts are required to be absent (empty). As a result, the grammar may contain only rules that have a single non-terminal on their left-hand side. Sample grammar:

0. **`Name`** → **`tom`** | **`dick`** | **`harry`**
1. **`Sentence`**$_S$ → **`Name`** | **`List and Name`**
2. **`List`** → **`Name , List`** | **`Name`**

2.3.2.1 Production Independence

Since there is always only one symbol on the left-hand side, each node in a production graph has the property that whatever it produces is independent of what its neighbors produce: the productive life of a non-terminal is independent of its context. Starlike forms as we saw in Figures 2.4, 2.6, and 2.8 cannot occur in a context-free production graph, which consequently has a pure *tree*-form and is called a *production tree*. An example is shown in Figure 2.9.

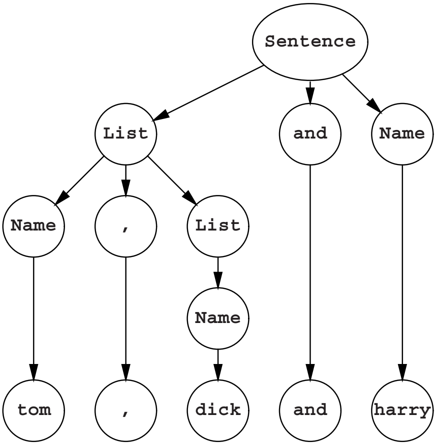


Fig. 2.9. Production tree for a context-free grammar

Since there is only one symbol on the left-hand side, all right-hand sides for a given non-terminal can always be collected in one grammar rule (we have already done that in the above grammar) and then each grammar rule reads like a definition of the left-hand side:

- A **`Sentence`** is either a **`Name`** or a **`List`** followed by **`and`** followed by a **`Name`**.
- A **`List`** is either a **`Name`** followed by a **`,`** followed by a **`List`**, or it is a **`Name`**.

This shows that context-free grammars build the strings they produce by two processes: *concatenation* ("... followed by ...") and *choice* ("either ... or ..."). In addition to these processes there is the *identification mechanism* which links the name of a non-terminal used in a right-hand side to its defining rule ("... is a ...").

At the beginning of this chapter we identified a language as a set of strings, the set of terminal productions of the start symbol. The independent production property allows us to extend this definition to any non-terminal in the grammar: each non-terminal produces a set, a language, independent of the other non-terminals. If we write the set of strings produced by A as $\mathcal{L}(A)$ and A has a production rule with, say, two alternatives, $A \rightarrow \alpha|\beta$, then $\mathcal{L}(A) = \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$, where $\cup$ is the union operator on sets. This corresponds to the choice in the previous paragraph. If α then consists of, say, three members PqR, we have $\mathcal{L}(\alpha) = \mathcal{L}(P) \circ \mathcal{L}(q) \circ \mathcal{L}(R)$, where $\circ$ is the concatenation operator on strings (actually on the strings in the sets). This corresponds to the concatenation above. And $\mathcal{L}(a)$ where a is a terminal is of course the set $\{a\}$. A non-terminal whose language contains ε is called *nullable*. One also says that it "produces empty".

Note that we cannot define a language $\mathcal{L}(Q)$ for the Q in Figure 2.7: Q does not produce anything meaningful by itself. Defining a language for a non-start symbol is possible only for Type 2 grammars and lower, and so is defining a non-start non-terminal as nullable.

Related to the independent production property is the notion of *recursion*. A non-terminal A is recursive if an A in a sentential form can produce something that again contains an A. The production of Figure 2.9 starts with the sentential form **`Sentence`**, which uses rule 1.2 to produce **`List and Name`**. The next step could very well be the replacement of the **`List`** by **`Name,List`**, using rule 2.1. We see that **`List`** produces something that again contains **`List`**:

```
Sentence -> List and Name -> Name , List and Name
```

`List` is recursive, more in particular, it is *directly recursive*. The non-terminal **`A`** in **`A->Bc`**, **`B->dA`** is *indirectly recursive*, but not much significance is to be attached to the difference.

It is more important that **`List`** is right-recursive: a non-terminal A is *right-recursive* if it can produce something that has an A at the right end, as **`List`** can:

```
List -> Name , List
```

Likewise, a non-terminal A is *left-recursive* if it can produce something that has an A at the left end: we could have defined

```
List -> List , Name
```

A non-terminal *A* is *self-embedding* if there is a derivation in which *A* produces *A* with something, say α, before it *and* something, say β, after it. Self-embedding describes nesting: α is the part produced when entering another level of nesting; β is the part produced when leaving that level. The best-known example of nesting is the use of parentheses in arithmetic expressions:

`arith_expression`$_S$ → ··· | `simple_expression`
`simple_expression` → `number` | `'(' arith_expression ')'`

A non-terminal can be left-recursive and right-recursive at the same time; it is then self-embedding. `A→Ab|cA|d` is an example.

If no non-terminal in a grammar is recursive, each production step uses up one non-terminal, since that non-terminal will never occur again in that segment. So the production process cannot continue unlimitedly, and a finite language results. Recursion is essential for life in grammars.

2.3.2.2 Some Examples

In the actual world, many things are defined in terms of other things. Context-free grammars are a very concise way to formulate such interrelationships. An almost trivial example is the composition of a book, as given in Figure 2.10. Of course this is a context-free description of a book, so one can expect it to also generate a lot of good-looking nonsense like

`Book`$_S$ → `Preface ChapterSequence Conclusion`
`Preface` → `"PREFACE" ParagraphSequence`
`ChapterSequence` → `Chapter` | `Chapter ChapterSequence`
`Chapter` → `"CHAPTER" Number ParagraphSequence`
`ParagraphSequence` → `Paragraph` | `Paragraph ParagraphSequence`
`Paragraph` → `SentenceSequence`
`SentenceSequence` → ···
···
`Conclusion` → `"CONCLUSION" ParagraphSequence`

Fig. 2.10. A simple (and incomplete!) grammar of a book

```
PREFACE
qwertyuiop
CHAPTER V
asdfghjkl
zxcvbnm,.
CHAPTER II
qazwsxedcrfvtgb
yhnujmikolp
CONCLUSION
All cats say blert when walking through walls.
```

but at least the result has the right structure. Document preparation and text mark-up systems like SGML, HTML and XML use this approach to express and control the basic structure of documents.

A shorter but less trivial example is the language of all elevator motions that return to the same point (a Manhattan turtle restricted to 5th Avenue would make the same movements)

```
ZeroMotion_s  →  up ZeroMotion down ZeroMotion
              |  down ZeroMotion up ZeroMotion
              |  ε
```

(in which we assume that the elevator shaft is infinitely long; it would be, in Manhattan).

If we ignore enough detail we can also recognize an underlying context-free structure in the sentences of a natural language, for example, English:

```
    Sentence_s  →  Subject Verb Object
       Subject  →  NounPhrase
        Object  →  NounPhrase
    NounPhrase  →  the QualifiedNoun
 QualifiedNoun  →  Noun | Adjective QualifiedNoun
          Noun  →  castle | caterpillar | cats
     Adjective  →  well-read | white | wistful | ···
          Verb  →  admires | bark | criticize | ···
```

which produces sentences like:

`the well-read cats criticize the wistful caterpillar`

Since, however, no context is incorporated, it will equally well produce the incorrect

`the cats admires the white well-read castle`

For keeping context we could use a phrase structure grammar (for a simpler language):

```
    Sentence_s  →  Noun Number Verb
        Number  →  Singular | Plural
 Noun Singular  →  castle Singular | caterpillar Singular | ···
 Singular Verb  →  Singular admires | ···
      Singular  →  ε
   Noun Plural  →  cats Plural | ···
   Plural Verb  →  Plural bark | Plural criticize | ···
        Plural  →  ε
```

where the markers **`Singular`** and **`Plural`** control the production of actual English words. Still, this grammar allows the cats to bark.... For a better way to handle context, see the various sections in Chapter 15, especially Van Wijngaarden grammars (Section 15.2) and attribute and affix grammars (Section 15.3).

The bulk of examples of CF grammars originate from programming languages. Sentences in these languages (that is, programs) have to be processed automatically (that is, by a compiler) and it was soon recognized (around 1958) that this is much easier if the language has a well-defined formal grammar. The syntaxes of all programming languages in use today are defined through formal grammars.

Some authors (for example Chomsky) and some parsing algorithms, require a CF grammar to be monotonic. The only way a CF rule can be non-monotonic is by having an empty right-hand side. Such a rule is called an ε-*rule* and a grammar that contains no such rules is called ε-*free*.

The requirement of being ε-free is not a real restriction, just a nuisance. Almost any CF grammar can be made ε-free by systematic substitution of the ε-rules; the exception is a grammar in which the start symbol already produces ε. The transformation process is explained in detail in Section 4.2.3.1), but it shares with many other grammar transformations the disadvantage that it usually ruins the structure of the grammar. The issue will be discussed further in Section 2.5.

2.3.2.3 Notation Styles

There are several different styles of notation for CF grammars for programming languages, each with endless variants; they are all functionally equivalent. We shall show two main styles here. The first is the *Backus-Naur Form* (*BNF*) which was first used to define ALGOL 60. Here is a sample:

```
<name>::=         tom | dick | harry
<sentence>_S::=   <name> | <list> and <name>
<list>::=         <name>, <list> | <name>
```

This form's main properties are the use of angle brackets to enclose non-terminals and of `::=` for "may produce". In some variants, the rules are terminated by a semicolon.

The second style is that of the CF van Wijngaarden grammars. Again a sample:

```
name:         tom symbol; dick symbol; harry symbol.
sentence_S:   name; list, and symbol, name.
list:         name, comma symbol, list; name.
```

The names of terminal symbols end in `...symbol`; their representations are hardware-dependent and are not defined in the grammar. Rules are properly terminated (with a period). Punctuation is used more or less in the traditional way; for example, the comma binds tighter than the semicolon. The punctuation can be read as follows:

`:`	"is defined as a(n)"
`;`	", or as a(n)"
`,`	"followed by a(n)"
`.`	", and as nothing else."

The second rule in the above grammar would then read as: "a sentence is defined as a name, or as a list followed by an and-symbol followed by a name, and as nothing else." Although this notation achieves its full power only when applied in the two-level Van Wijngaarden grammars, it also has its merits on its own: it is formal and still quite readable.

2.3.2.4 Extended CF Grammars

CF grammars are often made both more compact and more readable by introducing special short-hands for frequently used constructions. If we return to the Book grammar of Figure 2.10, we see that rules like:

$$\texttt{SomethingSequence} \rightarrow \texttt{Something} \mid \texttt{Something SomethingSequence}$$

occur repeatedly. In an *extended context-free grammar* we can write $\texttt{Something}^+$ meaning “one or more `Something`s” and we do not need to give a rule for $\texttt{Something}^+$; the rule

$$\texttt{Something}^+ \rightarrow \texttt{Something} \mid \texttt{Something Something}^+$$

is implicit. Likewise we can use $\texttt{Something}^*$ for “zero or more `Something`s” and $\texttt{Something}^?$ for “zero or one `Something`” (that is, “optionally a `Something`”). In these examples, the operators $^+$, * and $^?$ work on the preceding symbol. Their range can be extended by using parentheses: $\texttt{(Something ;)}^?$ means “optionally a `Something`-followed-by-a-`;`”. These facilities are very useful and allow the Book grammar to be written more efficiently (Figure 2.11). Some styles even allow constructions like $\texttt{Something}^{+4}$, meaning “one or more `Something`s with a maximum of 4”, or $\texttt{Something}^{+}\texttt{,}$ meaning “one or more `Something`s separated by commas”; this seems to be a case of overdoing a good thing. This notation for grammars is called *Extended BNF* (*EBNF*).

$$\begin{array}{rcl}
\texttt{Book}_S & \rightarrow & \texttt{Preface Chapter}^+ \texttt{ Conclusion} \\
\texttt{Preface} & \rightarrow & \texttt{"PREFACE" Paragraph}^+ \\
\texttt{Chapter} & \rightarrow & \texttt{"CHAPTER" Number Paragraph}^+ \\
\texttt{Paragraph} & \rightarrow & \texttt{Sentence}^+ \\
\texttt{Sentence} & \rightarrow & \cdots \\
 & \cdots & \\
\texttt{Conclusion} & \rightarrow & \texttt{"CONCLUSION" Paragraph}^+
\end{array}$$

Fig. 2.11. A grammar of a book in EBFN notation

The extensions of an EBNF grammar do not increase its expressive powers: all implicit rules can be made explicit and then a normal CF grammar in BNF notation results. Their strength lies in their user-friendliness. The star in the notation X^* with the meaning “a sequence of zero or more Xs” is called the *Kleene star*. If X is a set, X^* should be read as “a sequence of zero or more elements of X”; it is the same star that we saw in Σ^* in Section 2.1.3.3. Forms involving the repetition operators *, $^+$ or $^?$ and possibly the separators `(` and `)` are called *regular expressions*. EBNFs, which have regular expressions for their right-hand sides, are for that reason sometimes called *regular right part grammars RRP grammars* which is more descriptive than “extended context free”, but which is perceived to be a tongue twister by some.

There are two different schools of thought about the structural meaning of a regular right-hand side. One school maintains that a rule like:

```
Book  -->  Preface Chapter⁺ Conclusion
```

is an abbreviation of

```
Book  -->  Preface α Conclusion
   α  -->  Chapter | Chapter α
```

as shown above. This is the "(right)recursive" interpretation. It has the advantages that it is easy to explain and that the transformation to "normal" CF is simple. Disadvantages are that the transformation entails anonymous rules (identified by α here) and that the lopsided production tree for, for example, a book of four chapters does not correspond to our idea of the structure of the book; see Figure 2.12.

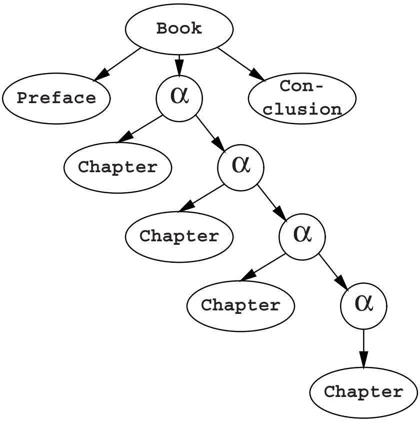


Fig. 2.12. Production tree for the (right)recursive interpretation

The second school claims that

```
Book  -->  Preface Chapter⁺ Conclusion
```

is an abbreviation of

```
Book  -->  Preface Chapter Conclusion
       |   Preface Chapter Chapter Conclusion
       |   Preface Chapter Chapter Chapter Conclusion
       |   ···
      ···
```

This is the "iterative" interpretation. It has the advantage that it yields a beautiful production tree (Figure 2.13), but the disadvantages are that it involves an infinite number of production rules and that the nodes in the production tree have a varying fan-out.

Since the implementation of the iterative interpretation is far from trivial, most practical parser generators use the recursive interpretation in some form or another, whereas most research has been done on the iterative interpretation.

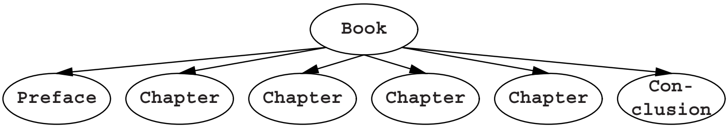


Fig. 2.13. Production tree for the iterative interpretation

2.3.3 Type 3 Grammars

The basic property of CF grammars is that they describe things that nest: an object may contain other objects in various places, which in turn may contain … etc. When during the production process we have finished producing one of the objects, the right-hand side still "remembers" what has to come after it: in the English grammar, after having descended into the depth of the non-terminal `Subject` to produce something like `the wistful cat`, the right-hand side `Subject Verb Object` still remembers that a `Verb` must follow. While we are working on the `Subject`, the `Verb` and `Object` remain queued at the right in the sentential form, for example,

```
the wistful QualifiedNoun Verb Object
```

In the right-hand side

```
up ZeroMotion down ZeroMotion
```

after having performed the `up` and an arbitrarily complicated `ZeroMotion`, the right-hand side still remembers that a `down` must follow.

The restriction to Type 3 disallows this recollection of things that came before: a right-hand side may only contain one non-terminal and it must come at the end. This means that there are only two kinds of rules:[4]

- a non-terminal produces zero or more terminals, and
- a non-terminal produces zero or more terminals followed by one non-terminal.

The original Chomsky definition of Type 3 restricts the kinds of rules to

- a non-terminal produces one terminal.
- A non-terminal produces one terminal followed by one non-terminal.

Our definition is equivalent and more convenient, although the conversion to Chomsky Type 3 is not completely trivial.

Type 3 grammars are also called *regular grammars* (RE grammars) or *finite-state grammars* (FS grammars). More precisely the version defined above is called *right-regular* since the only non-terminal in a rule is found at the right end of the right-hand side. This distinguishes them from the *left-regular grammars*, which are subject to the restrictions

---

[4] There is a natural in-between class, Type 2.5 so to speak, in which only a single non-terminal is allowed in a right-hand side, but where it need not be at the end. This gives us the so-called *linear grammars*.

- a non-terminal produces zero or more terminals
- a non-terminal produces one non-terminal followed by zero or more terminals

where the only non-terminal in a rule is found at the left end of the right-hand side. Left-regular grammars are less intuitive than right-regular ones, occur less frequently, and are more difficult to process, but they do occur occasionally (see for example Section 5.1.1), and need to be considered. They are discussed in Section 5.6.

Given the prevalence of right-regular over left-regular, the term "regular grammar" is usually intended to mean "*right*-regular grammar", and left-regularity is mentioned explicitly. We will follow this convention in this book.

It is interesting to compare the definition of right-regular to that of right-recursive (page 24). A non-terminal *A* is right-*recursive* if it can produce a sentential form that has an *A* at the right end; *A* is right-*regular* if, when it produces a sentential form that contains *A*, the *A* is at the right end.

In analogy to context-free grammars, which are called after what they cannot do, regular grammars could be called "non-nesting grammars".

Since regular grammars are used very often to describe the structure of text on the character level, it is customary for the terminal symbols of a regular grammar to be single characters. We shall therefore write `t` for `Tom`, `d` for `Dick`, `h` for `Harry` and `&` for `and`. Figure 2.14(*a*) shows a right-regular grammar for our `t,d&h` language in this style, 2.14(*b*) a left-regular one.

```
Sentence_S  ->  t | d | h | List
      List  ->  t ListTail | d ListTail | h ListTail
  ListTail  ->  , List | & t | & d | & h
```

(*a*)

```
Sentence_S  ->  t | d | h | List
      List  ->  ListHead & t | ListHead & d | ListHead & h
  ListHead  ->  ListHead , t | ListHead , d | ListHead , h |
                     t | d | h
```

(*b*)

Fig. 2.14. Type 3 grammars for the t, d & h language

The production tree for a sentence from a Type 3 (right-regular) grammar degenerates into a "production chain" of non-terminals that drop a sequence of terminals on their left. Figure 2.15 shows an example. Similar chains are formed by left-regular grammars, with terminals dropping to the left.

The deadly repetition exhibited by the grammar of Figure 2.14 is typical of regular grammars and a number of notational devices have been invented to abate this nuisance. The most common one is the use of square brackets to indicate "one out of a set of characters": `[tdh]` is an abbreviation for `t|d|h`:

```
S_S  ->  [tdh] | L
  L  ->  [tdh] T
  T  ->  , L | & [tdh]
```

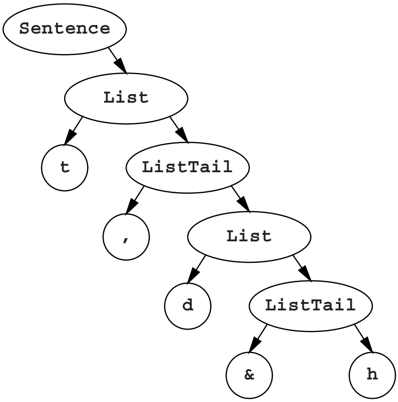


Fig. 2.15. Production chain for a right-regular (Type 3) grammar

which may look more cryptic at first but is actually much more convenient and in fact allows simplification of the grammar to

```
S_s  -->  [tdh] | L
 L   -->  [tdh] , L | [tdh] & [tdh]
```

A second way is to allow macros, names for pieces of the grammar that are substituted properly into the grammar before it is used:

```
Name  -->  t | d | h
  S_s -->  $Name | L
   L  -->  $Name , L | $Name & $Name
```

The popular parser generator for regular grammars *lex* (Lesk and Schmidt [360]) features both facilities.

If we adhere to the Chomsky definition of Type 3, our grammar will not get smaller than:

```
S_s  -->  t | d | h | t M | d M | h M
 M   -->  , N | & P
 N   -->  t M | d M | h M
 P   -->  t | d | h
```

This form is easier to process but less user-friendly than the *lex* version. We observe here that while the formal-linguist is interested in and helped by minimally sufficient means, the computer scientist values a form in which the concepts underlying the grammar (`$Name`, etc.) are easily expressed, at the expense of additional processing.

There are two interesting observations about regular grammars which we want to make here. First, when we use a regular grammar for generating a sentence, the sentential forms will only contain one non-terminal and this will always be at the end; that is where it all happens (using the grammar of Figure 2.14):

$\texttt{Sentence}_S$
$\texttt{List}$
$\texttt{t ListTail}$
$\texttt{t , List}$
$\texttt{t , d ListTail}$
$\texttt{t , d \& h}$

The second observation is that all regular grammars can be reduced considerably in size by using the regular expression operators *, $^+$ and $^?$ introduced in Section 2.3.2 for “zero or more”, “one or more” and “optionally one”, respectively. Using these operators and `(` and `)` for grouping, we can simplify our grammar to:

$$\texttt{S}_S \;\rightarrow\; \texttt{((\ [tdh],\ )}^{*}\ \texttt{[tdh]\&\ )}^{?}\ \texttt{[tdh]}$$

Here the parentheses serve to demarcate the operands of the * and $^?$ operators. Regular expressions exist for all Type 3 grammars. Note that the * and the $^+$ work on what precedes them. To distinguish them from the normal multiplication and addition operators, they are often printed higher than the level text in print, but in computer input they are in line with the rest, and other means must be used to distinguish them.

2.3.4 Type 4 Grammars

The last restriction we shall apply to what is allowed in a production rule is a pretty final one: no non-terminal is allowed in the right-hand side. This removes all the generative power from the mechanism, except for the choosing of alternatives. The start symbol has a (finite) list of alternatives from which we are allowed to choose; this is reflected in the name *finite-choice grammar* (FC grammar).

There is no FC grammar for our **`t,d&h`** language; if, however, we are willing to restrict ourselves to lists of names of a finite length (say, no more than a hundred), then there is one, since one could enumerate all combinations. For the obvious limit of three names, we get:

$$\texttt{S}_S \;\rightarrow\; \texttt{[tdh]} \;|\; \texttt{[tdh]\ \&\ [tdh]} \;|\; \texttt{[tdh]\ ,\ [tdh]\ \&\ [tdh]}$$

for a total of $3+3\times3+3\times3\times3 = 39$ production rules.

FC grammars are not part of the official Chomsky hierarchy in that they are not identified by Chomsky. They are nevertheless very useful and are often required as a tail-piece in some process or reasoning. The set of reserved words (*keywords*) in a programming language can be described by an FC grammar. Although not many grammars are FC in their entirety, some of the rules in many grammars are finite-choice. For example, the first rule of our first grammar (Figure 2.3) was FC. Another example of a FC rule was the macro introduced in Section 2.3.3. We do not need the macro mechanism if we change

> zero or more terminals

in the definition of a regular grammar to

> zero or more terminals or FC non-terminals

In the end, the FC non-terminals will only introduce a finite number of terminals.

2.3.5 Conclusion

The table in Figure 2.16 summarizes the most complicated data structures that can occur in the production of a string, in correlation to the grammar type used. See also Figure 3.15 for the corresponding data types obtained in parsing.

Chomsky type	Grammar type	Most complicated data structure	Example figure
0 / 1	PS / CS	production dag	2.8
2	CF	production tree	2.9
3	FS	production list	2.15
4	FC	production element	—

Fig. 2.16. The most complicated production data structure for the Chomsky grammar types

2.4 Actually Generating Sentences from a Grammar

2.4.1 The Phrase-Structure Case

Until now we have only produced single sentences from our grammars, in an ad hoc fashion, but the purpose of a grammar is to generate all of its sentences. Fortunately there is a systematic way to do so. We shall use the $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ grammar as an example. We start from the start symbol and systematically make all possible substitutions to generate all sentential forms; we just wait and see which ones evolve into sentences and when. Try this by hand for, say, 10 sentential forms. If we are not careful, we are apt to only generate forms like **`aSQ`**, **`aaSQQ`**, **`aaaSQQQ`**, ..., and we will never see a finished sentence. The reason is that we focus too much on a single sentential form: we have to give equal time to all of them. This can be done through the following algorithm, which keeps a queue (that is, a list to which we add at the end and remove from the beginning), of sentential forms.

Start with the start symbol as the only sentential form in the queue. Now continue doing the following:

- Consider the first sentential form in the queue.
- Scan it from left to right, looking for strings of symbols that match the left-hand side of a production rule.
- For each such string found, make enough copies of the sentential form, replace in each one the string that matched a left-hand side of a rule by a different alternative of that rule, and add them all to the end of the queue.
- If the original sentential form does not contain any non-terminals, write it down as a sentence in the language.
- Throw away the original sentential form; it has been fully processed.

If no rule matched, and the sentential form was not a finished sentence, it was a blind alley; they are removed automatically by the above process and leave no trace.

Since the above procedure enumerates all strings in a PS language, PS languages are also called *recursively enumerable* sets, where "recursively" is to be taken to mean "by a possibly recursive algorithm".

The first couple of steps of this process for our $\texttt{a}^n\texttt{b}^n\texttt{c}^n$ grammar from Figure 2.7 are depicted in Figure 2.17. The queue runs to the right, with the first item on the left. We see that we do not get a sentence for each time we turn the crank; in fact, in this case real sentences will get scarcer and scarcer. The reason is of course that during the process more and more side lines develop, which all require equal attention. Still, we can be certain that every sentence that can be produced, will in the end be produced: we leave no stone unturned. This way of doing things is called *breadth-first production*; computers are better at it than people.

Step	Queue	Result
1	`S`	
2	`abc` `aSQ`	`abc`
3	`aSQ`	
4	`aabcQ` `aaSQQ`	
5	`aaSQQ` `aabQc`	
6	`aabQc` `aaabcQQ` `aaaSQQQ`	
7	`aaabcQQ` `aaaSQQQ` `aabbcc`	
8	`aaaSQQQ` `aabbcc` `aaabQcQ`	
9	`aabbcc` `aaabQcQ` `aaaabcQQQ` `aaaaSQQQQ`	`aabbcc`
10	`aaabQcQ` `aaaabcQQQ` `aaaaSQQQQ`	
11	`aaaabcQQQ` `aaaaSQQQQ` `aaabbccQ` `aaabQQc`	
...	...	

Fig. 2.17. The first couple of steps in producing for $\texttt{a}^n\texttt{b}^n\texttt{c}^n$

It is tempting to think that it is unnecessary to replace *all* left-hand sides that we found in the top-most sentential form. Why not just replace the first one and wait for the resulting sentential form to come up again and then do the next one? This is wrong, however, since doing the first one may ruin the context for doing the second one. A simple example is the grammar

`S`$_\text{S}$ --> `AC`
`A` --> `b`
`AC` --> `ac`

First doing `A-->b` will lead to a blind alley and the grammar will produce nothing. Doing both possible substitutions will lead to the same blind alley, but then there will also be a second sentential form, `ac`. This is also an example of a grammar for which the queue will get empty after a (short) while.

If the grammar is context-free (or regular) there is no context to ruin and it is quite safe to just replace the first (or only) match.

There are two remarks to be made here. First, it is not at all certain that we will indeed obtain a sentence for all our effort: it is quite possible that every new sentential form again contains non-terminals. We should like to know this in advance by examining the grammar, but it can be proven that it is impossible to do so for PS grammars. The formal-linguist says “It is *undecidable* whether a PS grammar produces the empty set”, which means that there cannot be an algorithm that will for every PS grammar correctly tell if the grammar produces at least one sentence. This does not mean that we cannot prove for some given grammar that it generates nothing, if that is the case. It means that the proof method used will not work for *all* grammars: we could have a program that correctly says Yes in finite time if the answer is Yes but that takes infinite time if the answer is No. In fact, our generating procedure above is such an algorithm that gives the correct Yes/No answer in infinite time (although we can have an algorithm that gives a Yes/Don’t know answer in finite time). Although it is true that because of some deep property of formal languages we cannot always get exactly the answer we want, this does not prevent us from obtaining all kinds of useful information that gets close. We shall see that this is a recurring phenomenon. The computer scientist is aware of but not daunted by the impossibilities from formal linguistics.

The second remark is that when we do get sentences from the above production process, they may be produced in an unexploitable order. For non-monotonic grammars the sentential forms may grow for a while and then suddenly shrink again, perhaps even to the empty string. Formal linguistics proves that there cannot be an algorithm that for all PS grammars produces their sentences in increasing (actually “non-decreasing”) length. In other words, the parsing problem for PS grammars is *unsolvable*. (Although the terms are used interchangeably, it seems reasonable to use “undecidable” for yes/no questions and “unsolvable” for problems.)

2.4.2 The CS Case

The above language-generating procedure is also applicable to CS grammars, except for the parts about undecidability. Since the sentential forms under development can never shrink, the strings are produced in monotonic order of increasing length. This means that if the empty string is not the first string, it will never appear and the CS grammar does not produce ε. Also, if we want to know if a given string w is in the language, we can just wait until we see it come up, in which case the answer is Yes, or until we see a longer string come up, in which case the answer is No.

Since the strings in a CS language can be recognized by a possibly recursive algorithm, CS languages are also called *recursive sets*.

2.4.3 The CF Case

When we generate sentences from a CF grammar, many things are a lot simpler. It can still happen that our grammar will never produce a sentence, but now we can test for that beforehand, as follows. First scan the grammar to find all non-terminals that have a right-hand side that contains terminals only or is empty. These non-terminals are guaranteed to produce something. Now scan again to find non-terminals that have a right-hand side that consists of only terminals and non-terminals that are guaranteed to produce something. This will give us new non-terminals that are guaranteed to produce something. Repeat this until we find no more new such non-terminals. If we have not met the start symbol this way, it will not produce anything.

Furthermore we have seen that if the grammar is CF, we can afford to just rewrite the leftmost non-terminal every time (provided we rewrite it into all its alternatives). Of course we can also consistently rewrite the rightmost non-terminal. Both approaches are similar but different. Using the grammar

0.	`N`	->	`t` | `d` | `h`
1.	$\texttt{S}_s$	->	`N` | `L & N`
2.	`L`	->	`N , L` | `N`

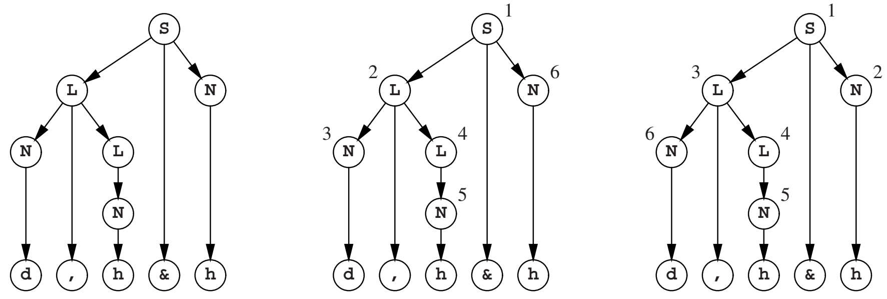
let us follow the adventures of the sentential form that will eventually result in **`d,h&h`**. Although it will go up and down the production queue several times, we only depict here what changes are made to it. Figure 2.18 shows the sentential forms for leftmost and rightmost substitution, with the rules and alternatives involved; for example, (1b) means rule 1 alternative b, the second alternative.

	`S`		`S`
1b		1b	
	`L&N`		`L&N`
2a		0c	
	`N,L&N`		`L&h`
0b		2a	
	`d,L&N`		`N,L&h`
2b		2b	
	`d,N&N`		`N,N&h`
0c		0c	
	`d,h&N`		`N,h&h`
0c		0b	
	`d,h&h`		`d,h&h`

Fig. 2.18. Sentential forms leading to **`d,h&h`**, with leftmost and rightmost substitution

The sequences of production rules used are not as similar as we would expect. Of course in grand total the same rules and alternatives are applied, but the sequences are neither equal nor each other's mirror image, nor is there any other obvious relationship. Both sequences define the same production tree (Figure 2.19(*a*)), but if we number the non-terminals in it in the order they were rewritten, we get different numberings, as shown in (*b*) and (*c*).

The sequence of production rules used in leftmost rewriting is called the *leftmost derivation* of a sentence. We do not have to indicate at what position a rule must be applied, nor do we need to give its rule number. Just the alternative is sufficient; the position and the non-terminal are implicit. A *rightmost derivation* is defined in a similar way.



(*a*) (*b*) (*c*)

Fig. 2.19. Production tree (*a*) with leftmost (*b*) and rightmost (*c*) derivation order

A leftmost production step can be indicated by using an arrow marked with a small l: $\texttt{N,L\&N}\underset{l}{\rightarrow}\texttt{d,L\&N}$, and the leftmost production sequence

$$\texttt{S} \underset{l}{\rightarrow} \texttt{L\&N} \underset{l}{\rightarrow} \texttt{N,L\&N} \underset{l}{\rightarrow} \texttt{d,L\&N} \underset{l}{\rightarrow} \texttt{d,N\&N} \underset{l}{\rightarrow} \texttt{d,h\&N} \underset{l}{\rightarrow} \texttt{d,h\&h}$$

can be abbreviated to $\texttt{S}\overset{*}{\underset{l}{\rightarrow}}\texttt{d,h\&h}$. Likewise, the rightmost production sequence

$$\texttt{S} \underset{r}{\rightarrow} \texttt{L\&N} \underset{r}{\rightarrow} \texttt{L\&h} \underset{r}{\rightarrow} \texttt{N,L\&h} \underset{r}{\rightarrow} \texttt{N,N\&h} \underset{r}{\rightarrow} \texttt{N,h\&h} \underset{r}{\rightarrow} \texttt{d,h\&h}$$

can be abbreviated to $\texttt{S}\overset{*}{\underset{r}{\rightarrow}}\texttt{d,h\&h}$. The fact that `S` produces `d,h&h` in any way is written as $\texttt{S}\overset{*}{\rightarrow}\texttt{d,h\&h}$.

The task of parsing is to reconstruct the derivation tree (or graph) for a given input string. Some of the most efficient parsing techniques can be understood more easily if viewed as attempts to reconstruct a left- or rightmost derivation process of the input string; the derivation tree then follows automatically. This is why the notion “[left|right]-most derivation” occurs frequently in this book (note the FC grammar used here).

2.5 To Shrink or Not To Shrink

In the previous paragraphs, we have sometimes been explicit as to the question if a right-hand side of a rule may be shorter than its left-hand side and sometimes we have been vague. Type 0 rules may definitely be of the shrinking variety, monotonic rules definitely may not, and Type 2 and 3 rules can shrink only by producing empty (ε); that much is sure.

The original Chomsky hierarchy (Chomsky [385]) was very firm on the subject: only Type 0 rules are allowed to make a sentential form shrink. Type 1, 2 and 3 rules are all monotonic. Moreover, Type 1 rules have to be of the context-sensitive variety, which means that only one of the non-terminals in the left-hand side is actually allowed to be replaced (and then not by ε). This makes for a proper hierarchy in which each next class is a proper subset of its parent and in which all derivation graphs except for those of Type 0 grammars are actually derivation trees.

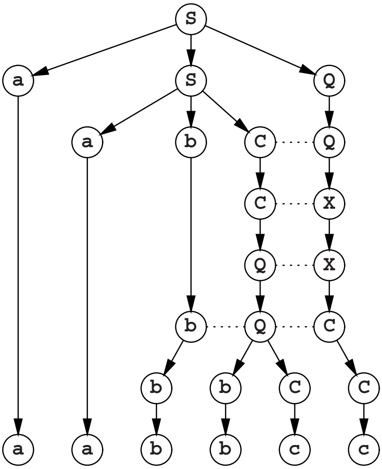
As an example consider the grammar for the language $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ given in Figure 2.7:

1. $\mathtt{S_S} \rightarrow$ **abc** | **aSQ**
2. **bQc** $\rightarrow$ **bbcc**
3. **cQ** $\rightarrow$ **Qc**

which is monotonic but not context-sensitive in the strict sense. It can be made CS by expanding the offending rule 3 and introducing a non-terminal for **c**:

1. $\mathtt{S_S} \rightarrow$ **abC** | **aSQ**
2. **bQC** $\rightarrow$ **bbCC**
3a. **CQ** $\rightarrow$ **CX**
3b. **CX** $\rightarrow$ **QX**
3c. **QX** $\rightarrow$ **QC**
4. **C** $\rightarrow$ **c**

Now the production graph of Figure 2.8 turns into a production tree:



There is an additional reason for shunning ε-rules: they make both proofs and parsers more complicated, sometimes much more complicated; see, for example, Section 9.5.4. So the question arises why we should bother with ε-rules at all; the answer is that they are very convenient for the grammar writer and user.

If we have a language that is described by a CF grammar with ε-rules and we want to describe it by a grammar without ε-rules, then that grammar will almost always be more complicated. Suppose we have a system that can be fed bits of information, like: “Amsterdam is the capital of the Netherlands”, “Truffles are expensive”, and can then be asked a question. On a very superficial level we can define its input as:

input$_S$**: zero-or-more-bits-of-info question**

or, in an extended notation

input$_S$**: bit-of-info*** **question**

Since **`zero-or-more-bits-of-info`** will, among other strings, produce the empty string, at least one of the rules used in its grammar will be an ε-rule; the * in the extended notation already implies an ε-rule somewhere. Still, from the user's point of view, the above definition of input neatly fits the problem and is exactly what we want.

Any attempt to write an ε-free grammar for this input will end up defining a notion that comprises some of the later **`bits-of-info`** together with the **`question`** (since the **`question`** is the only non-empty part, it must occur in all rules involved!) But such a notion does not fit our problem at all and is an artifact:

```
                     input_S:  question-preceded-by-info
 question-preceded-by-info:  question
                            | bit-of-info
                                 question-preceded-by-info
```

As a grammar becomes more and more complicated, the requirement that it be ε-free becomes more and more of a nuisance: the grammar is working against us, not for us.

This presents no problem from a theoretical point of view: any CF language can be described by an ε-free CF grammar and ε-rules are never needed. Better still, any grammar with ε-rules can be mechanically transformed into an ε-free grammar for the same language. We saw an example of such a transformation above and details of the algorithm are given in Section 4.2.3.1. But the price we pay is that of any grammar transformation: it is no longer our grammar and it reflects the original structure less well.

The bottom line is that the practitioner finds the ε-rule to be a useful tool, and it would be interesting to see if there exists a hierarchy of non-monotonic grammars alongside the usual Chomsky hierarchy. To a large extend there is: Type 2 and Type 3 grammars need not be monotonic (since they can always be made so if the need arises); it turns out that context-sensitive grammars with shrinking rules are equivalent to unrestricted Type 0 grammars; and monotonic grammars with ε-rules are also equivalent to Type 0 grammars. We can now draw the two hierarchies in one picture; see Figure 2.20. Drawn lines separate grammar types with different power. Conceptually different grammar types with the same power are separated by blank space. We see that if we insist on non-monotonicity, the distinction between Type 0 and Type 1 disappears.

A special case arises if the language of a Type 1 to Type 3 grammar itself contains the empty string. This cannot be incorporated into the grammar in the monotonic hierarchy since the start symbol already has length 1 and no monotonic rule can make it shrink. So the empty string has to be attached as a special property to the grammar. No such problem occurs in the non-monotonic hierarchy.

Many parsing methods will in principle work for ε-free grammars only: if something does not produce anything, you can't very well see if it's there. Often the parsing method can be doctored to handle ε-rules, but that invariably increases the complexity of the method. It is probably fair to say that this book would be at least 30%

		Chomsky (monotonic) hierarchy		non-monotonic hierarchy
global production	Type 0	unrestricted phrase structure grammars	monotonic grammars with ε-rules	unrestricted phrase structure grammars
	Type 1	context-sensitive grammars	monotonic grammars, no ε-rules	context-sensitive grammars with non-monotonic rules
local production	Type 2	context-free ε-free grammars		context-free grammars
	Type 3	regular (ε-free) grammars		regular grammars, regular expressions
no production	Type 4	finite-choice		

Fig. 2.20. Summary of grammar hierarchies

thinner if ε-rules did not exist — but then grammars would lose much more than 30% of their usefulness!

2.6 Grammars that Produce the Empty Language

Roughly 1500 years after the introduction of zero as a number by mathematicians in India, the concept is still not well accepted in computer science. Many programming languages do not support records with zero fields, arrays with zero elements, or variable definitions with zero variables; in some programming languages the syntax for calling a routine with zero parameters differs from that for a routine with one or more parameters; many compilers refuse to compile a module that defines zero names; and this list could easily be extended. More in particular, we do not know of any parser generator that can produce a parser for the empty language, the language with zero strings.

All of which brings us to the question of what the grammar for the empty language would look like. First note that the empty language differs from the language that consists of only the empty string, a string with zero characters. This language is easily generated by the grammar $S_s \rightarrow \varepsilon$, and is handled correctly by the usual *lex-yacc* pipeline. Note that this grammar has no terminal symbols, which means that V_T in Section 2.2 is the empty set.

For a grammar to produce nothing, the production process cannot be allowed to terminate. This suggests one way to obtain such a grammar: $S_s \rightarrow S$. This is ugly, however, for two reasons. From an algorithmic point of view the generation process now just loops and no information about the emptiness of the language is obtained; and the use of the symbol S is arbitrary.

Another way is to force the production process to get stuck by not having any production rules in the grammar. Then R in Section 2.2 is empty too, and the form of the grammar is $(\{S\}, \{\}, S, \{\})$. This is not very satisfactory either, since now we have a non-terminal without a defining rule; and the symbol S is still arbitrary.

A better way is never to allow the production process to get started: have no start symbol. This can be accommodated by allowing a *set* of start symbols in the definition of a grammar rather than a single start symbol. There are other good reasons for doing so. An example is the grammar for a large programming language which has multiple "roots" for module specifications, module definitions, etc. Although these differ at the top level, they have large segments of the grammar in common. If we extend the definition of a CF grammar to use a set of start symbols, the grammar for the empty language obtains the elegant and satisfactory form ({}, {}, {}, {}).

Also on the subject of zero and empty: it might be useful to consider grammar rules in which the *left*-hand side is empty. Terminal productions of the right-hand sides of such rules may appear anywhere in the input, thus modeling noise and other every-day but extraneous events.

Our preoccupation with empty strings, sets, languages, etc. is not frivolous, since it is well known that the ease with which a system handles empty cases is a measure of its cleanliness and robustness.

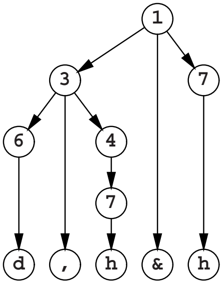
2.7 The Limitations of CF and FS Grammars

When one has been working for a while with CF grammars, one gradually gets the feeling that almost anything could be expressed in a CF grammar. That there are, however, serious limitations to what can be said by a CF grammar is shown by the famous *uvwxy* theorem, which is explained below.

2.7.1 The *uvwxy* Theorem

When we have obtained a sentence from a CF grammar, we may look at each (terminal) symbol in it, and ask: How did it get here? Then, looking at the production tree, we see that it was produced as, say, the *n*-th member of the right-hand side of rule number *m*. The left-hand side of this rule, the parent of our symbol, was again produced as the *p*-th member of rule *q*, and so on, until we reach the start symbol. We can, in a sense, trace the *lineage* of the symbol in this way. If all rule/member pairs in the lineage of a symbol are different, we call the symbol *original*, and if all the symbols in a sentence are original, we call the sentence "original".

For example, the lineage of the first **h** in the production tree



produced by the grammar

1. $\mathbf{S}_S$ → **L & N**
2. $\mathbf{S}_S$ → **N**
3. **L** → **N , L**
4. **L** → **N**
5. **N** → **t**
6. **N** → **d**
7. **N** → **h**

is **h** of **7,1** of **4,1** of **3,3** of **1,1**. Here the first number indicates the rule and the second number is the member number in that rule. Since all the rule/member pairs are different the **h** is original.

Now there is only a finite number of ways for a given symbol to be original. This is easy to see as follows. All rule/member pairs in the lineage of an original symbol must be different, so the length of its lineage can never be more than the total number of different rule/member pairs in the grammar. There are only so many of these, which yields only a finite number of combinations of rule/member pairs of this length or shorter. In theory the number of original lineages of a symbol can be very large, but in practice it is very small: if there are more than, say, ten ways to produce a given symbol from a grammar by original lineage, your grammar will be very convoluted indeed!

This puts severe restrictions on original sentences. If a symbol occurs twice in an original sentence, both its lineages must be different: if they were the same, they would describe the same symbol in the same place. This means that there is a maximum length to original sentences: the sum of the numbers of original lineages of all symbols. For the average grammar of a programming language this length is in the order of some thousands of symbols, i.e., roughly the size of the grammar. So, since there is a longest original sentence, there can only be a finite number of original sentences, and we arrive at the surprising conclusion that any CF grammar produces a finite-size kernel of original sentences and (probably) an infinite number of unoriginal sentences!

What do "unoriginal" sentences look like? This is where we come to the *uvwxy* theorem. An unoriginal sentence has the property that it contains at least one symbol in the lineage of which a repetition occurs. Suppose that symbol is a q and the repeated rule is A. We can then draw a picture similar to Figure 2.21, where w is the part produced by the most recent application of A, vwx the part produced by the other application of A and $uvwxy$ is the entire unoriginal sentence. Now we can immediately find another unoriginal sentence, by removing the smaller triangle headed by A and replacing it by a copy of the larger triangle headed by A; see Figure 2.22.

This new tree produces the sentence $uvvwxxy$ and it is easy to see that we can, in this way, construct a complete family of sentences uv^nwx^ny for all $n \geq 0$. This form shows the w nested in a number of v and x brackets, in an indifferent context of u and y.

The bottom line is that when we examine longer and longer sentences in a context-free language, the original sentences become exhausted and we meet only families of closely related sentences telescoping off into infinity. This is summarized

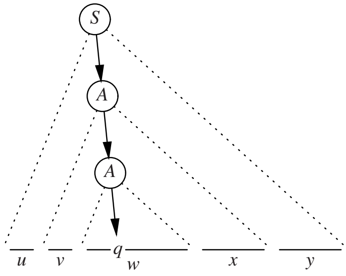


Fig. 2.21. An unoriginal sentence: *uvwxy*

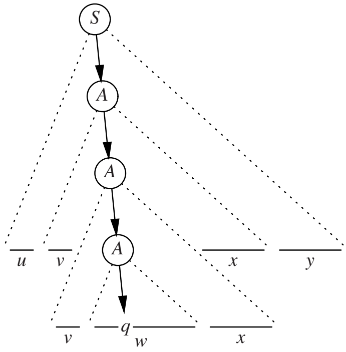


Fig. 2.22. Another unoriginal sentence, uv^2wx^2y

in the *uvwxy* theorem: any sentence generated by a CF grammar that is longer than the longest original sentence from that grammar can be cut into five pieces u, v, w, x and y, in such a way that uv^nwx^ny is a sentence from that grammar for all $n \geq 0$. The *uvwxy* theorem is also called the *pumping lemma for context-free languages* and has several variants.

Two remarks must be made here. The first is that if a language keeps on providing longer and longer sentences without reducing to families of nested sentences, there cannot be a CF grammar for it. We have already encountered the context-sensitive language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ and it is easy to see (but not quite so easy to prove!) that it does not decay into such nested sentences, as sentences get longer and longer. Consequently, there is no CF grammar for it. See Billington [396] for a general technique for such proofs.

The second is that the longest original sentence is a property of the grammar, not of the language. By making a more complicated grammar for a language we can increase the set of original sentences and push away the border beyond which we are forced to resort to nesting. If we make the grammar infinitely complicated, we can push the border to infinity and obtain a phrase structure language from it. How we can make a CF grammar infinitely complicated is described in the section on two-level grammars, 15.2.1.

2.7.2 The *uvw* Theorem

A simpler form of the *uvwxy* theorem applies to regular (Type 3) languages. We have seen that the sentential forms occurring in the production process for a FS grammar all contain only one non-terminal, which occurs at the end. During the production of a very long sentence, one or more non-terminals must occur two or more times, since there are only a finite number of non-terminals. Figure 2.23 shows what we see when we list the sentential forms one by one. The substring v has been produced from one occurrence of A to the next, u is a sequence that allows us to reach A, and w is a sequence that allows us to terminate the production process. It will be clear that, starting from the second A, we could have followed the same path as from the first A, and thus have produced $uvvw$. This leads us to the *uvw* theorem, or the *pumping lemma for regular languages*: any sufficiently long string from a regular language can be cut into three pieces u, v and w, so that uv^nw is a string in the language for all $n \geq 0$.

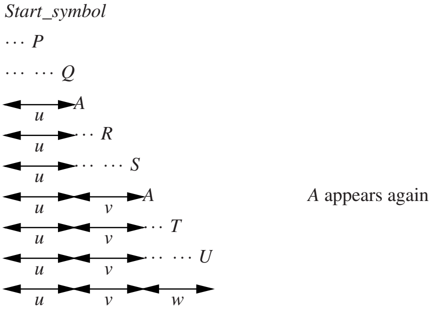


Fig. 2.23. Repeated occurrence of A may result in repeated occurrence of v

2.8 CF and FS Grammars as Transition Graphs

A *transition graph* is a directed graph in which the arrows are labeled with zero or more symbols from the grammar. The idea is that as you follow the arrows in the graph you produce one of the associated symbols, if there is one, and nothing otherwise. The nodes, often unlabeled, are resting points between producing the symbols. If there is more than one outgoing arrow from a node you can choose any to follow. So the transition graph in Figure 2.24 produces the same strings as the sample grammar on page 23.

It is fairly straightforward to turn a grammar into a set of transition graphs, one for each non-terminal, as Figure 2.25 shows. But it contains arrows marked with non-terminals, and the meaning of "producing" a non-terminal associated with an arrow is not directly clear. Suppose we are at node n_1, from which a transition (arrow) labeled with non-terminal N leads to a node n_2, and we want to take that transition.

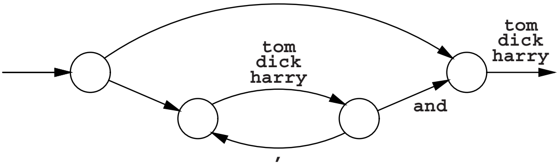


Fig. 2.24. A transition graph for the `[tdh]` language

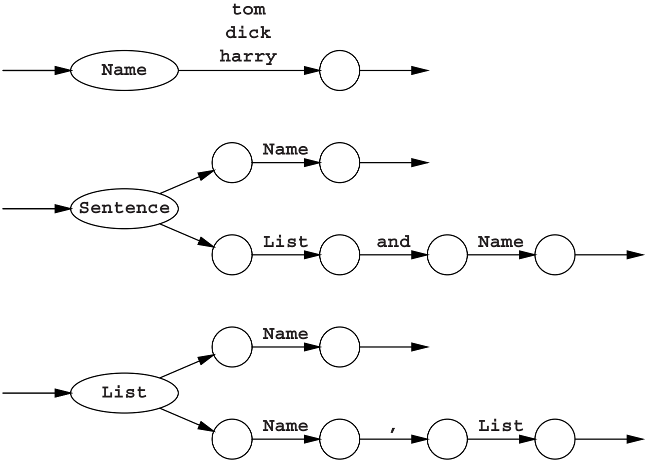


Fig. 2.25. A recursive transition network for the sample grammar on page 23

Rather than producing *N* by appending it to the output, we push node n_2 on a stack, and continue our walk at the entrance to the transition graph for *N*. And when we are leaving the transition graph for *N*, we pop n_2 from the stack and continue at node n_2. This is the *recursive transition network* interpretation of context-free grammars: the set of graphs is the transition network, and the stacking mechanism provides the recursion.

Figure 2.26 shows the right-regular rules of the FS grammar Figure 2.14(*a*) as transition graphs. Here we have left out the unmarked arrows at the exits of the graphs and the corresponding nodes; we could have done the same in Figure 2.25, but doing so would have complicated the stacking mechanism.

We see that we have to produce a non-terminal only when we are just leaving another, so we do not need to stack anything, and can interpret an arrow marked with a non-terminal *N* as a jump to the transition graph for *N*. So a regular grammar corresponds to a (non-recursive) transition network.

If we connect each exit marked *N* in such a network to the entrance of the graph for *N* we can ignore the non-terminals, and obtain a transition graph for the corresponding language. When we apply this short-circuiting to the transition network of Figure 2.26 and rearrange the nodes a bit, we get the transition graph of Figure 2.24.

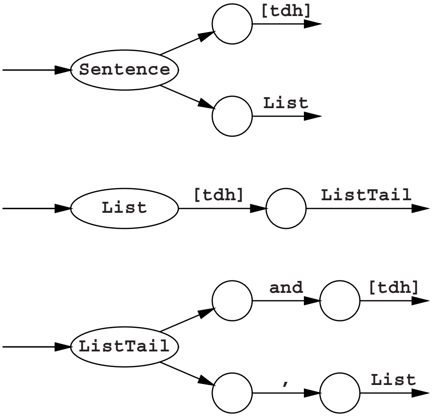


Fig. 2.26. The FS grammar of Figure 2.14(*a*) as transition graphs

2.9 Hygiene in Context-Free Grammars

All types of grammars can contain *useless rules*, rules that cannot play a role in any successful production process. A production process is *successful* when it results in a terminal string. Production attempts can be unsuccessful by getting stuck (no further substitution possible) or by entering a situation in which no substitution sequence will ever remove all non-terminals. An example of a Type 0 grammar that can get stuck is

1. $\mathtt{S}_s \rightarrow \mathtt{A}\ \mathtt{B}$
2. $\mathtt{S} \rightarrow \mathtt{B}\ \mathtt{A}$
3. $\mathtt{S} \rightarrow \mathtt{C}$
4. $\mathtt{A}\ \mathtt{B} \rightarrow \mathtt{x}$
5. $\mathtt{C} \rightarrow \mathtt{C}\ \mathtt{C}$

When we start with the first rule for `S`, all goes well and we produce the terminal string `x`. But when we start with rule 2 for `S` we get stuck, and when we start with rule 3, we get ourselves in an infinite loop, producing more and more `C`s. Rules 2, 3 and 5 can never occur in a successful production process: they are useless rules, and can be removed from the grammar without affecting the language produced.

Useless rules are not a fundamental problem: they do not obstruct the normal production process. Still, they are dead wood in the grammar, and one would like to remove them. Also, when they occur in a grammar specified by a programmer, they probably point at some error, and one would like to detect them and give warning or error messages.

The problems with the above grammar were easy to understand, but it can be shown that in general it is undecidable whether a rule in a Type 0 or 1 grammar is useless: there cannot be an algorithm that does it correctly in all cases. For context-free grammars the situation is different, however, and the problem is rather easily solved.

Rules in a context-free grammar can be useless through three causes: they may contain undefined non-terminals, they may not be reachable from the start symbol, and they may fail to produce anything. We will now discuss each of these ailments in more detail; an algorithm to rid a grammar of them is given in Section 2.9.5.

2.9.1 Undefined Non-Terminals

The right-hand side of some rule may contain a non-terminal for which no production rule is given. Such a rule will never have issue and can be removed from the grammar. If we do this, we may of course remove the last definition of another non-terminal, which will then in turn become undefined, etc.

We will see further on (for example in Section 4.1.3) that it is occasionally useful to also recognize undefined terminals. Rules featuring them in their right-hand sides can again be removed.

2.9.2 Unreachable Non-Terminals

If a non-terminal cannot be reached from the start symbol, its defining rules will never be used, and it cannot contribute to the production of any sentence. Unreachable non-terminals are sometimes called "unused non-terminals". But this term is a bit misleading, because an unreachable non-terminal A may still occur in some right-hand side $B \rightarrow \cdots A \cdots$, making it look useful, provided B is unreachable; the same applies of course to B, etc.

2.9.3 Non-Productive Rules and Non-Terminals

Suppose X has as its only rule $X \rightarrow aX$ and suppose X can be reached from the start symbol. Now X will still not contribute anything to the sentences of the language of the grammar, since once X is introduced, there is no way to get rid of it: X is a non-productive non-terminal. In addition, any rule which has X in its right-hand side is non-productive. In short, any rule that does not in itself produce a non-empty sublanguage is non-productive. If all rules for a non-terminal are non-productive, the non-terminal is non-productive.

In an extreme case *all* non-terminals in a grammar are non-productive. This happens when all right-hand sides in the grammar contain at least one non-terminal. Then there is just no way to get rid of the non-terminals, and the grammar itself is non-productive.

These three groups together are called *useless non-terminals*.

2.9.4 Loops

The above definition makes "non-useless" all rules that can be involved in the production of a sentence, but there still is a class of rules that are not really useful: rules of the form $A \rightarrow A$. Such rules are called *loops*. Loops can also be indirect: $A \rightarrow B$,

$B \rightarrow C$, $C \rightarrow A$; and they can be hidden: $A \rightarrow PAQ$, $P \xrightarrow{*} \varepsilon$, $Q \xrightarrow{*} \varepsilon$, so a production sequence $A \rightarrow PAQ \rightarrow \ldots A \ldots \rightarrow A$ is possible.

A loop can legitimately occur in the production of a sentence, and if it does, there is also a production of that sentence without the loop. Loops do not contribute to the language and any sentence the production of which involves a loop is *infinitely ambiguous*, meaning that there are infinitely many production trees for it. Algorithms for loop detection are given in Section 4.1.2.

Different parsers react differently to grammars with loops. Some (most of the general parsers) faithfully attempt to construct an infinite number of derivation trees, some (for example, the CYK parser) collapse the loop as described above and some (most deterministic parsers) reject the grammar. The problem is aggravated by the fact that loops can be concealed by ε-rules: a loop may only become visible when certain non-terminals produce ε.

A grammar without useless non-terminals and loops is called a *proper grammar*.

2.9.5 Cleaning up a Context-Free Grammar

Normally, grammars supplied by people do not contain undefined, unreachable or non-productive non-terminals. If they do, it is almost certainly a mistake (or a test!), and we would like to detect and report them. Such anomalies can, however, occur normally in generated grammars or be introduced by some grammar transformations, in which case we wish to detect them to "clean up" the grammar. Cleaning the grammar is also very important when we obtain the result of parsing as a parse-forest grammar (Section 3.7.4, Chapter 13, and many other places).

The algorithm to detect and remove useless non-terminals and rules from a context-free grammar consists of two steps: remove the non-productive rules and remove the unreachable non-terminals. Surprisingly it is not necessary to remove the useless rules due to undefined non-terminals: the first step does this for us automatically.

```
Ss  →  A B | D E
 A  →  a
 B  →  b C
 C  →  c
 D  →  d F
 E  →  e
 F  →  f D
```

Fig. 2.27. A demo grammar for grammar cleaning

We will use the grammar of Figure 2.27 for our demonstration. It looks fairly innocent: all its non-terminals are defined and it does not exhibit any suspicious constructions.

2.9.5.1 Removing Non-Productive Rules

We find the non-productive rules by finding the productive ones. Our algorithm hinges on the observation that a rule is productive if its right-hand side consists of symbols all of which are productive. Terminal symbols are productive since they produce terminals and empty is productive since it produces the empty string. A non-terminal is productive if there is a productive rule for it, but the problem is that initially we do not know which rules are productive, since that is exactly the thing we are trying to find out.

We solve this dilemma by first marking all rules and non-terminals as “Don’t know”. We now go through the grammar of Figure 2.27 and for each rule for which we do know that all its right-hand side members are productive, we mark the rule and the non-terminal it defines as “Productive”. This yields markings for the rules `A→a`, `C→c`, and `E→e`, and for the non-terminals `A`, `C` and `E`.

Now we know more and apply this knowledge in a second round through the grammar. This allows us to mark the rule `B→bC` and the non-terminal `B`, since now `C` is known to be productive. A third round gives us `S→AB` and `S`. A fourth round yields nothing new, so there is no point in a fifth round.

We now know that `S`, `A`, `B`, `C`, and `E` are productive, but `D` and `F` and the rule `S→DE` are still marked “Don’t know”. However, now we know more: we know that we have pursued all possible avenues for productivity, and have not found any possibilities for `D`, `F` and the second rule for `S`. That means that we can now upgrade our knowledge “Don’t know” to “Non-productive”. The rules for `D`, `F` and the second rule for `S` can be removed from the grammar; the result is shown in Figure 2.28. This makes `D` and `F` undefined, but `S` stays in the grammar since it is productive, in spite of having a non-productive rule.

```
S_s  →  A B
 A   →  a
 B   →  b C
 C   →  c
 E   →  e
```

Fig. 2.28. The demo grammar after removing non-productive rules

It is interesting to see what happens when the grammar contains an undefined non-terminal, say U. U will first be marked “Don’t know”, and since there is no rule defining it, it will stay “Don’t know”. As a result, any rule R featuring U in its right-hand side will also stay “Don’t know”. Eventually both will be recognized as “Non-productive”, and all rules R will be removed. We see that an “undefined non-terminal” is just a special case of a “non-productive” non-terminal: it is non-productive because there is no rule for it.

The above knowledge-improving algorithm is our first example of a *closure algorithm*. Closure algorithms are characterized by two components: an *initialization*, which is an assessment of what we know initially, partly derived from the situation and partly “Don’t know”; and an inference rule, which is a rule telling how knowledge from several places is to be combined. The inference rule for our problem was:

> For each rule for which we do know that all its right-hand side members are productive, mark the rule and the non-terminal it defines as “Productive’.’

It is implicit in a closure algorithm that the inference rule(s) are repeated until nothing changes any more. Then the preliminary “Don’t know” can be changed into a more definitive “Not X”, where “X” was the property the algorithm was designed to detect.

Since it is known beforehand that in the end all remaining “Don’t know” indications are going to be changed into “Not X”, many descriptions and implementations of closure algorithms skip the whole “Don’t know” stage and initialize everything to “Not X”. In an implementation this does not make much difference, since the meaning of the bits in computer memory is not in the computer but in the mind of the programmer, but especially in text-book descriptions this practice is unelegant and can be confusing, since it just is not true that initially all the non-terminals in our grammar are “Non-productive”.

We will see many examples of closure algorithms in this book; they are discussed in more detail in Section 3.9.

2.9.5.2 Removing Unreachable Non-Terminals

A non-terminal is called *reachable* or *accessible* if there exists at least one sentential form, derivable from the start symbol, in which it occurs. So a non-terminal A is reachable if $S \xrightarrow{*} \alpha A \beta$ for some α and β.

We found the non-productive rules and non-terminals by finding the “productive” ones. Likewise, we find the unreachable non-terminals by finding the reachable ones. For this, we can use the following closure algorithm. First, the start symbol is marked “reachable”; this is the initialization. Then, for each rule in the grammar of the form $A \rightarrow \alpha$ with A marked, all non-terminals in α are marked; this is the inference rule. We continue applying the inference rule until nothing changes any more. Now the unmarked non-terminals are not reachable and their rules can be removed.

The first round marks **`A`** and **`B`**; the second marks **`C`**, and the third produces no change. The result — a clean grammar — is in Figure 2.29. We see that rule **`E-->e`**, which was reachable and productive in Figure 2.27 became isolated by removing the non-productive rules, and is then removed by the second step of the cleaning algorithm.

$$\begin{array}{rcl} \mathtt{S_S} & \rightarrow & \mathtt{A\ B} \\ \mathtt{A} & \rightarrow & \mathtt{a} \\ \mathtt{B} & \rightarrow & \mathtt{b\ C} \\ \mathtt{C} & \rightarrow & \mathtt{c} \end{array}$$

Fig. 2.29. The demo grammar after removing all useless rules and non-terminals

Removing the unreachable rules cannot cause a non-terminal N used in a reachable rule to become undefined, since N can only become undefined by removing all its defining rules but since N is reachable, the above process will not remove any rule for it. A slight modification of the same argument shows that removing the unreachable rules cannot cause a non-terminal N used in a reachable rule to become non-productive: N, which was productive or it would not have survived the previous clean-up step, can only become non-productive by removing some of its defining rules but since N is reachable, the above process will not remove any rule for it. This shows conclusively that after removing non-productive non-terminals and then removing unreachable non-terminals we do not need to run the step for removing non-productive non-terminals again.

It is interesting to note, however, that first removing unreachable non-terminals and then removing non-productive rules may produce a grammar which again contains unreachable non-terminals. The grammar of Figure 2.27 is an example in point.

Furthermore it should be noted that cleaning a grammar may remove *all* rules, including those for the start symbol, in which case the grammar describes the empty language; see Section 2.6.

Removing the non-productive rules is a bottom-up process: only the bottom level, where the terminal symbols live, can know what is productive. Removing unreachable non-terminals is a top-down process: only the top level, where the start symbol(s) live(s), can know what is reachable.

2.10 Set Properties of Context-Free and Regular Languages

Since languages are sets, it is natural to ask if the standard operations on sets — union, intersection, and negation (complement) — can be performed on them, and if so, how.

The *union* of two sets S_1 and S_2 contains the elements that are in either set; it is written $S_1 \cup S_2$.. The *intersection* contains the elements that are in both sets; it is written $S_1 \cap S_2$. And the *negation* of a set S contains those in Σ^* but not in S; it is written $\neg S$. In the context of formal languages the sets are defined through grammars, so actually we want to do the operations on the grammars rather than on the languages.

Constructing the grammar for the union of two languages is trivial for context-free and regular languages (and in fact for all Chomsky types): just construct a new start symbol $S' \rightarrow S_1|S_2$, where S_1 and S_2 are the start symbols of the two grammars that describe the two languages. (Of course, if we want to combine the two grammars into one we must make sure that the names in them differ, but that is easy to do.)

Intersection is a different matter, though, since the intersection of two context-free languages need not be context-free, as the following example shows. Consider the two CF languages $L_1 = \texttt{a}^n\texttt{b}^n\texttt{c}^m$ and $L_2 = \texttt{a}^m\texttt{b}^n\texttt{c}^n$ described by the CF grammars

$$\begin{array}{lll} L_{1\mathrm{s}} & \rightarrow & \texttt{A P} \\ \texttt{A} & \rightarrow & \texttt{a A b} \mid \varepsilon \\ \texttt{P} & \rightarrow & \texttt{c P} \mid \varepsilon \end{array} \qquad \text{and} \qquad \begin{array}{lll} L_{2\mathrm{s}} & \rightarrow & \texttt{Q C} \\ \texttt{Q} & \rightarrow & \texttt{a Q} \mid \varepsilon \\ \texttt{C} & \rightarrow & \texttt{b C c} \mid \varepsilon \end{array}$$

When we take a string that occurs in both languages and thus in their intersection, it will have the form $\mathtt{a}^p\mathtt{b}^q\mathtt{c}^r$ where $p = q$ because of L_1 and $q = r$ because of L_2. So the intersection language consists of strings of the form $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ and we know that that language is not context-free (Section 2.7.1).

The intersection of CF languages has weird properties. First, the intersection of two CF languages always has a Type 1 grammar — but this grammar is not easy to construct. More remarkably, the intersection of three CF languages is more powerful than the intersection of two of them: Liu and Weiner [390] show that there are languages that can be obtained as the intersection of k CF languages, but not of $k-1$. In spite of that, *any* Type 1 language, and even any Type 0 language, can be constructed by intersecting just two CF languages, provided we are allowed to erase all symbols in the resulting strings that belong to a set of *erasable symbols*.

The CS language we will use to demonstrate this remarkable phenomenon is the set of all strings that consist of two identical parts: ww, where w is any string over the given alphabet; examples are `aa` and `abbababbab`. The two languages to be intersected are defined by

```
L3s  -->  A P                           L4s  -->  Q C
  A  -->  a A x | b A y | ε     and       Q  -->  a Q | b Q | ε
  P  -->  a P | b P | ε                   C  -->  x C a | y C b | ε
```

where `x` and `y` are the erasable symbols. The first grammar produces strings consisting of three parts, a sequence A_1 of `a`s and `b`s, followed by its “dark mirror” image M_1, in which `a` corresponds to `x` and `b` to `y`, followed by an arbitrary sequence G_1 of `a`s and `b`s. The second grammar produces strings consisting of an arbitrary sequence G_2 of `a`s and `b`s, a “dark” sequence M_2, and its mirror image A_2, in which again `a` corresponds to `x` and `b` to `y`. The intersection forces $A_1 = G_2$, $M_1 = M_2$, and $G_1 = A_2$. This makes A_2 the mirror image of the mirror image of A_1, in other words equal to A_1. An example of a string in the intersection is `abbabyxyyxabbab`, where we see the mirror images `abbab` and `yxyyx`. We now erase the erasable symbols `x` and `y` and obtain our result `abbababbab`.

Using a massive application of the above mirror-mirror trick, one can relatively easily prove that any Type 0 language can be constructed as the intersection of two CF languages, plus a set of erasable symbols. For details see, for example, Révész [394].

Remarkably the intersection of a context-free and a *regular* language is always a context-free language, and, what’s more, there is a relatively simple algorithm to construct a grammar for that intersection language. This gives rise to a set of unusual parsing algorithms, which are discussed in Chapter 13.

If we cannot have intersection of two CF languages and stay inside the CF languages, we certainly cannot have negation of a CF language and stay inside the CF languages. If we could, we could negate two languages, take their union, negate the result, and so obtain their intersection. In a formula: $L_1 \cap L_2 = \neg((\neg L_1) \cup (\neg L_2))$; this formula is known as De Morgan’s Law.

In Section 5.4 we shall see that union, intersection and negation of regular (Type 3) languages yield regular languages.

It is interesting to speculate what would have happened if formal languages had been based on set theory with all the set operations right from the start, rather than on the Chomsky hierarchy. Would context-free languages still have been invented?

2.11 The Semantic Connection

Sometimes parsing serves only to check the correctness of a string; that the string conforms to a given grammar may be all we want to know, for example because it confirms our hypothesis that certain observed patterns are indeed correctly described by the grammar we have designed for it. Often, however, we want to go further: we know that the string conveys a meaning, its semantics, and this semantics is directly related to the structure of the production tree of the string. (If it is not, we have the wrong grammar!)

Attaching semantics to a grammar is done in a very simple and effective way: to each rule in the grammar, a *semantic clause* is attached which relates the semantics of the members of the right-hand side of the rule to the semantics of the left-hand side, in which case the semantic information flows from the leaves of the tree upwards to the start symbol; or the other way around, in which case the semantic information flows downwards from the start symbol to the leaves; or both ways, in which case the semantic information may have to flow up and down for a while until a stable situation is reached. Semantic information flowing down is called *inherited*: each rule inherits it from its parent in the tree. Semantic information flowing up is called *derived*: each rule derives it from its children.

There are many ways to express semantic clauses. Since our subject is parsing and syntax rather than semantics, we will briefly describe only two often-used and well-studied techniques: attribute grammars and transduction grammars. We shall explain both using the same simple example, the language of sums of one-digit numbers; the semantics of a sentence in this language is the value of the sum. The language is generated by the grammar of Figure 2.30. One of its sentences is, for example, **3+5+1**; its semantics is 9.

1. $\texttt{Sum}_S \rightarrow \texttt{Digit}$
2. $\texttt{Sum} \rightarrow \texttt{Sum + Digit}$
3. $\texttt{Digit} \rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9}$

Fig. 2.30. A grammar for sums of one-digit numbers

2.11.1 Attribute Grammars

The semantic clauses in an attribute grammar assume that each node in the production tree has room for one or more *attributes*, which are just values (numbers, strings or anything else) sitting in nodes in production trees. For simplicity we restrict ourselves to attribute grammars with only one attribute per node. The semantic clause of a rule in such a grammar contains some formulas which compute the attributes of some of the non-terminals in that rule (represented by nodes in the production tree) from those of other non-terminals in that same rule. These *semantic actions* connect only attributes that are local to the rule: the overall semantics is composed as the result of all the local computations.

If the semantic action of a rule R computes the attribute of the left-hand side of R, that attribute is *derived*. If it computes an attribute of one of the non-terminals in the right-hand side of R, say A, then that attribute is *inherited* by A. Derived attributes are also called “synthesized attributes”. The attribute grammar for our example is:

1. `Sum`$_S$ `->` `Digit` $\{A_0 := A_1\}$
2. `Sum` `->` `Sum + Digit` $\{A_0 := A_1 + A_3\}$
3a. `Digit` `->` `0` $\{A_0 := 0\}$
··· ···
3j. `Digit` `->` `9` $\{A_0 := 9\}$

The semantic clauses are given between curly brackets. A_0 is the (derived) attribute of the left-hand side; $A_1, \ldots, A_n$ are the attributes of the members of the right-hand side. Traditionally, terminal symbols in a right-hand side are also counted in determining the index of A, although they do not (normally) carry attributes; the **`Digit`** in rule 2 is in position 3 and its attribute is A_3. Most systems for handling attribute grammars have less repetitive ways to express rule 3a through 3j.

The initial production tree for **`3+5+1`** is given in Figure 2.31. First only the attributes for the leaves are known, but as soon as all attributes in a right-hand side of a production rule are known, we can use its semantic clause to compute the attribute of its left-hand side. This way the attribute values (semantics) percolate up the tree, finally reach the start symbol and provide us with the semantics of the whole sentence, as shown in Figure 2.32. Attribute grammars are a very powerful method of handling the semantics of a language. They are discussed in more detail in Section 15.3.1.

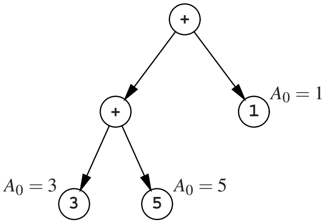


Fig. 2.31. Initial stage of the attributed production tree for **`3+5+1`**

2.11.2 Transduction Grammars

Transduction grammars define the semantics of a string (the “input string”) as another string, the “output string” or “translation”, rather than as the final attribute of

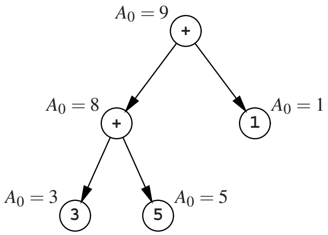


Fig. 2.32. Fully attributed production tree for `3+5+1`

the start symbol. This method is less powerful but much simpler than using attributes and often sufficient. The semantic clause in a production rule is just the string that should be output for the corresponding node. We assume that the string for a node is output just after the strings for all its children. Other variants are possible and in fact usual. We can now write a transduction grammar which translates a sum of digits into instructions to compute the value of the sum.

```
1.     Sum_S  -->  Digit          "make it the result"
2.      Sum   -->  Sum + Digit    "add it to the previous result"
3a.   Digit   -->  0              "take a 0"
          ...  ...
3j.   Digit   -->  9              "take a 9"
```

This transduction grammar translates `3+5+1` into:

```
take a 3
make it the result
take a 5
add it to the previous result
take a 1
add it to the previous result
```

which is indeed what `3+5+1` “means”.

2.11.3 Augmented Transition Networks

Semantics can be introduced in a recursive transition network (Section 2.8) by attaching actions to the transitions in the graphs. These actions can set variables, construct data structures, etc. A thus augmented recursive transition network is known as an *Augmented Transition Network* (or *ATN*) (Woods [378]).

2.12 A Metaphorical Comparison of Grammar Types

Text books claim that “Type n grammars are more powerful than Type $n+1$ grammars, for $n = 0, 1, 2$”, and one often reads statements like “A regular (Type 3) grammar is not powerful enough to match parentheses”. It is interesting to see what kind of power is meant. Naively, one might think that it is the power to generate larger and larger sets, but this is clearly incorrect: the largest possible set of strings, Σ^*, is easily generated by the Type 3 grammar

$$\mathtt{S}_\mathrm{s} \rightarrow [\Sigma]\ \mathtt{S} \mid \varepsilon$$

where $[\Sigma]$ is an abbreviation for the symbols in the language. It is just when we want to restrict this set, that we need more powerful grammars. More powerful grammars can define more complicated boundaries between correct and incorrect sentences. Some boundaries are so fine that they cannot be described by any grammar (that is, by any generative process).

This idea has been depicted metaphorically in Figure 2.33, in which a rose is approximated by increasingly finer outlines. In this metaphor, the rose corresponds to the language (imagine the sentences of the language as molecules in the rose); the grammar serves to delineate its silhouette. A regular grammar only allows us straight horizontal and vertical line segments to describe the flower; ruler and T-square suffice, but the result is a coarse and mechanical-looking picture. A CF grammar would approximate the outline by straight lines at any angle and by circle segments; the drawing could still be made using the classical tools of compasses and ruler. The result is stilted but recognizable. A CS grammar would present us with a smooth curve tightly enveloping the flower, but the curve is too smooth: it cannot follow all the sharp turns, and it deviates slightly at complicated points; still, a very realistic picture results. An unrestricted phrase structure grammar can represent the outline perfectly. The rose itself cannot be caught in a finite description; its essence remains forever out of our reach.

A more prosaic and practical example can be found in the successive sets of Java[5] programs that can be generated by the various grammar types.

- The set of all lexically correct Java programs can be generated by a regular grammar. A Java program is lexically correct if there are no newlines inside strings, comments are terminated before end-of-file, all numerical constants have the right form, etc.
- The set of all syntactically correct Java programs can be generated by a context-free grammar. These programs conform to the (CF) grammar in the manual.
- The set of all semantically correct Java programs can be generated by a CS grammar. These are the programs that pass through a Java compiler without drawing error messages.
- The set of all Java programs that would terminate in finite time when run with a given input can be generated by an unrestricted phrase structure grammar. Such a grammar would, however, be very complicated, since it would incorporate detailed descriptions of the Java library routines and the Java run-time system.
- The set of all Java programs that solve a given problem (for example, play chess) cannot be generated by a grammar (although the description of the set is finite).

Note that each of the above sets is a subset of the previous set.

---

[5] We use the programming language Java here because we expect that most of our readers will be more or less familiar with it. Any programming language for which the manual gives a CF grammar will do.

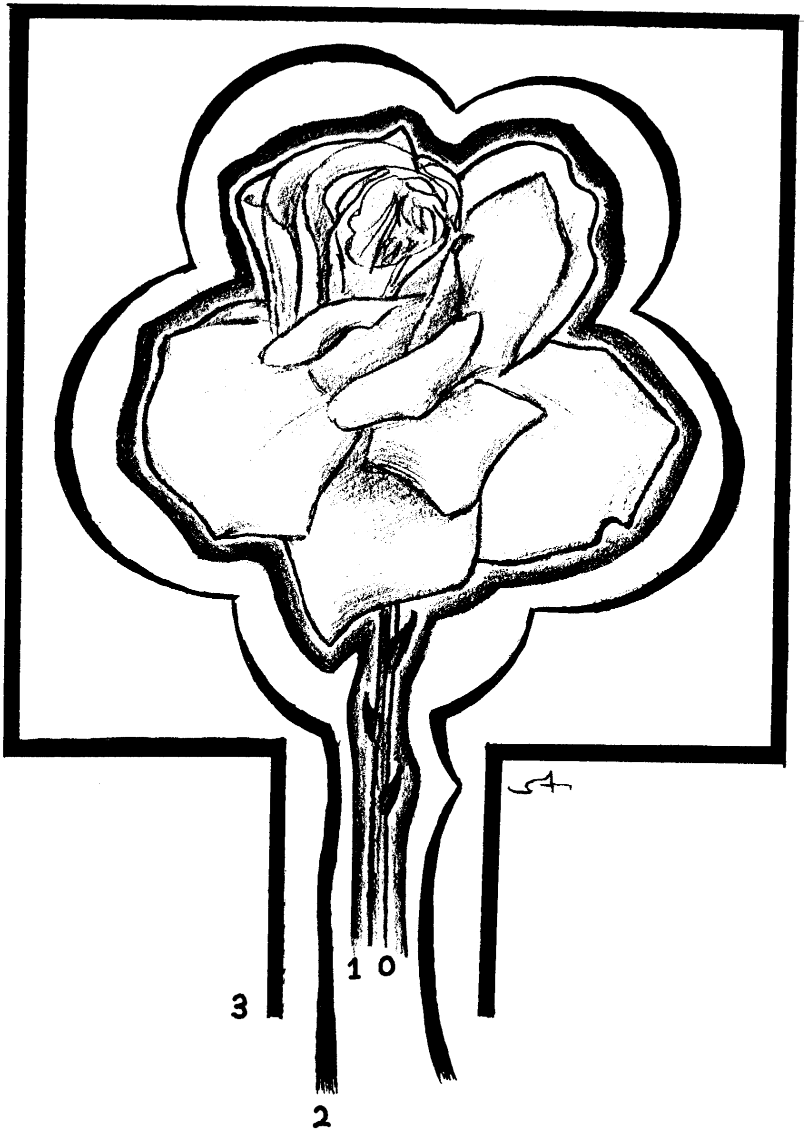


Fig. 2.33. The silhouette of a rose, approximated by Type 3 to Type 0 grammars

2.13 Conclusion

A Chomsky grammar is a finite mechanism that produces a usually infinite set of strings, a "language." Unlike many other set generation mechanisms, this production process assigns a structure to the produced string, which can be utilized to attach semantics to it. For context-free (Type 2) grammars, this structure is a tree, which allows the semantics to be composed from the semantics of the branches. This is the basis of the importance of context-free grammars.

Problems

Problem 2.1: The diagonalization procedure on page 11 seems to be a finite description of a language not on the list. Why is the description not on the list, which contains all finite descriptions after all?

Problem 2.2: In Section 2.1.3.4 we considered the functions n, $n+10$, and $2n$ to find the positions of the bits that should differ from those in line n. What is the general form of these functions, i.e., what set of functions will generate languages that do not have finite descriptions?

Problem 2.3: Write a grammar for Manhattan turtle paths in which the turtle is never allowed to the west of its starting point.

Problem 2.4: Show that the monotonic Type 1 grammar of Figure 2.7 produces all strings of the form $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ for $n \geq 1$, and no others. Why is $n = 0$ excluded?

Problem 2.5: Write a Type 1 grammar that produces the language of all strings that consists of two identical parts: ww, where w is any string over the given alphabet (see Section 2.10).

Problem 2.6: On page 34 we have the sentence production mechanism add the newly created sentential forms to the end of the queue, claiming that this realizes breadth-first production. When we put them at the start of the queue, the mechanism uses depth-first production. Show that this does not work.

Problem 2.7: The last paragraph of Section 2.4.1 contains the words "in increasing (actually 'non-decreasing') length". Explain why "non-decreasing" is enough.

Problem 2.8: Relate the number of strings in the finite language produced by a grammar without recursion (page 25) to the structure of that grammar.

Problem 2.9: Refer to Section 2.6. Find more examples in your computing environment where zero as a number gets a second-class treatment.

Problem 2.10: In your favorite parser generator system, write a parser for the language $\{\varepsilon\}$. Same question for the language $\{\}$.

Problem 2.11: Use the uvw theorem (Section 2.7.2) to show that there is no Type 3 grammar for the language $\mathtt{a}^i\mathtt{b}^i$.

Problem 2.12: In Section 2.9 we write that useless rules can be removed from the grammar without affecting the language produced. This seems to suggest that "not affecting the language by its removal" is the actual property we are after, rather than just uselessness. Comment.

Problem 2.13: Write the Chomsky production process of Section 2.2.2 as a closure algorithm.