THE FUNCTIONAL TREATMENT OF PARSING

THE KLUWER INTERNATIONAL SERIES IN ENGINEERING AND COMPUTER SCIENCE

NATURAL LANGUAGE PROCESSING AND MACHINE TRANSLATION

Consulting Editor

Jaime Carbonell

Other books in the series:

NATURAL LANGUAGE PROCESSING: THE PLNLP APPROACH, Karen Jensen, George E. Heidorn, Stephen D. Richardson
ISBN: 0-7923-9279-5
ADAPTIVE PARSING: Self-Extending Natural Language Interfaces, J. F. Lehman
ISBN: 0-7923-9183-7
GENERALIZED L. R. PARSING, M. Tomita
ISBN: 0-7923-9201-9
CONCEPTUAL INFORMATION RETRIEVAL: A Case Study in Adaptive Partial Parsing, M. L. Mauldin
ISBN: 0-7923-9214-0
CURRENT ISSUES IN PARSING TECHNOLOGY, M. Tomita
ISBN: 0-7923-9131-4
NATURAL LANGUAGE GENERATION IN ARTIFICIAL INTELLIGENCE AND COMPUTATIONAL LINGUISTICS, C. L. Paris, W. R. Swartout, W. C. Mann
ISBN: 0-7923-9098-9
UNDERSTANDING EDITORIAL TEXT: A Computer Model of Argument Comprehension, S. J. Alvarado
ISBN: 0-7923-9123-3
NAIVE SEMANTICS FOR NATURAL LANGUAGE UNDERSTANDING, K. Dahlgren
ISBN: 0-89838-287-4
INTEGRATED NATURAL LANGUAGE DIALOGUE: A Computational Model, R. E. Frederking
ISBN: 0-89838-255-6
A NATURAL LANGUAGE INTERFACE FOR COMPUTER AIDED DESIGN, T. Samad
ISBN: 0-89838-222-X
EFFICIENT PARSING FOR NATURAL LANGUAGE: A Fast Algorithm for Practical Systems, M. Tomita
ISBN: 0-89838-202-5

THE FUNCTIONAL TREATMENT OF PARSING

by

René Leermakers

Institute for Perception Research,
Eindhoven, The Netherlands

SPRINGER SCIENCE+BUSINESS MEDIA, B.V.

Library of Congress Cataloging-in-Publication Data

Leermakers, René
The functional treatment of parsing / by René Leermakers.
p. cm. -- (Kluwer international series in engineering and computer science ; v. 242)
Includes bibliographical references and index.

1. Natural language processing (Computer science) 2. Parsing (Computer grammar) 3. Functional programming (Computer science) I. Title. II. Series: Kluwer international series in engineering and computer science ; SECS 242.
QA76.9.N38L42 1993
005.13'1--dc20 93-22799

ISBN 978-1-4613-6397-2 ISBN 978-1-4615-3186-9 (eBook)
DOI 10.1007/978-1-4615-3186-9

Printed on acid-free paper

All Rights Reserved
© 1993 Springer Science+Business Media Dordrecht
Originally published by Kluwer Academic Publishers in 1993
Softcover reprint of the hardcover 1st edition 1993
No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without written permission from the copyright owner.

To my family
but in particular
to my son Arjeh
who made his first steps
as this book went to press
and to my daughter Mirjam

FOREWORD

Formal-language theory, theoretical linguistics and computational linguistics have shared roots in the 1950s, with the seminal work of Kleene, Chomsky, Miller and Bar-Hillel on regular languages and phrase-structure grammars. However, various social, cultural and technological factors have since then conspired to split those disciplines and weaken their understanding and appreciation of each other. Efficiency considerations and the fact that programming languages are human artifacts may partly justify the focus on deterministic languages and parsers in the theory of context-free parsing. However, natural languages are highly ambiguous and thus non-deterministic, making much of that theory seem irrelevant to natural-language parsing. It has thus been difficult to convince the computational linguist of the importance of context-free-parsing theory, if not for specific algorithms, then for concepts and techniques essential to the rigorous analysis of natural-language parsers. Looking in the other direction, the formal-language theorist is mostly unaware of the special problems of natural-language parsing, and thus not only misses a potentially rich area for new research but also fails to appreciate the efforts of computational linguists. For these reasons, the publication of *The Functional Treatment of Parsing* is doubly welcome.

For the computational linguist, René Leermakers's book brings out the relevance of the theory of context-free parsing to natural-language parsing. His innovative use of functional notation makes algorithms and their derivation less mysterious, and eliminates much of the need for the laborious inductive proofs of correctness found in other parsing theory texts. In addition, the functional approach ties well with the widespread acquaintance of current and recent students with the functional programming paradigm through languages such as Scheme and ML. Delicate data structure issues in parsing are clearly located in elegant abstractions representing nondeterminism and result reuse.

For the computer scientist, *The Functional Treatment of Parsing* offers a fresh and unified perspective on a variety of parsing algorithms, some well-known and some less so. This new perspective offers much simpler proofs

of correctness and computational complexity, and eliminates the artificial distinction between stack-based and tabular parsers. The use of equational reasoning rather than special-purpose inductive proofs to relate algorithms to their specifications is an excellent application of an approach to program derivation and verification that has received strong support in the works of Boyer, Moore, Dijkstra and Gries. From a more practical angle, Leermakers's approach provides a theoretical basis for the parsing component of interactive language-development environments, for which the standard deterministic parsing methods have been proven unwieldy.

Parsing theory has many subtleties, requiring attentive and thoughtful study. While the present book does not excuse the student from those obligations, it will provide ample rewards to readers at all levels. In addition to a self-contained and elegant treatment of all the main ideas of context-free parsing, it brings out the underlying unity of the subject as no other book I know of, and offers a wealth of conceptual and technical riches, of which I particularly enjoyed the application of Lambek types to the analysis of grammatical covers and attribute grammars.

There have been increasing signs in the research literature of a long-overdue convergence between formal-language theory and computational linguistics, in particular in the area of context-free parsing. *The Functional Treatment of Parsing* not only demonstrates that convergence for the first time in book form, but also revives context-free parsing theory as an interesting and relevant topic for computational linguists and computer scientists alike.

Fernando C.N. Pereira.

CONTENTS

FOREWORD *by Fernando Pereira* vii

PREFACE xiii

1 CONTEXT-FREE GRAMMARS 1

2 BUNCH NOTATION 7
2.1 Bunches 8
2.2 Algorithmic interpretation 12

3 GRAMMAR INTERPRETATIONS 15
3.1 The natural interpretation 15
3.2 Derivation 20
3.3 The Lambek types 23
3.4 Recognition functions 26
3.5 Generation 28
3.6 Summary of interpretations 29

4 RECURSIVE DESCENT 33
4.1 The functional interpretation 33
4.2 Termination 35
4.3 Complexity and memoization 35
4.4 Look ahead 38
4.5 Error recovery 42

5 GRAMMAR TRANSFORMATIONS 45
5.1 Making grammars bilinear 45

5.2	Recursive descent for E_G	48
5.3	Partial elimination of left recursion	49
5.4	Recursive descent for F_G	57
6	**RECURSIVE ASCENT**	**61**
6.1	The algorithm	62
6.2	Termination	65
6.3	A variant that works with strings	66
6.4	Complexity	68
6.5	EBNF grammars	69
7	**PARSE FOREST**	**75**
7.1	Informal introduction	75
7.2	The grammar E'_G	77
7.3	Forest for bilinear grammars	78
7.4	The set Q	82
7.5	Standard Earley parser	85
7.6	Earley versus Earley	87
8	**ATTRIBUTE GRAMMARS**	**91**
8.1	Notational conventions	92
8.2	Attribute functions	94
8.3	Example	96
8.4	Function graphs	100
8.5	Attribute grammar parser	103
8.6	Direct attribute evaluation	104
9	**LR PARSERS**	**115**
9.1	LR(0) recognizer	115
9.2	The deterministic case	120
9.3	Implementation with stacks	123
9.4	Some variants	126
9.5	Look ahead	128
9.6	Attributes	130
9.7	Continuations	131
9.8	Error recovery	134

9.9 The methods by Lang and Tomita 137
9.10 Evaluation w.r.t. standard approaches 138
9.11 Earley versus LR 140

10 SOME NOTES 143
10.1 Context-free grammars 143
10.2 Names 144
10.3 Bunches 145
10.4 Functional programming 145
10.5 Grammar transformations 146
10.6 Memo-functions 146
10.7 Parse forests 147
10.8 Earley 147
10.9 Attribute grammars 147
10.10 Natural language 148
10.11 Other applications 148
10.12 LR parsing 149
10.13 EBNF 149
10.14 Conclusion 150

REFERENCES 151

INDEX 157

PREFACE

The theory of parsing with respect to context-free grammars is one of the old and established parts of computer science. The first contributions were rather theoretical treatises within automata theory. Later on, contributions to the field came from people who were interested in practical applications such as compiler construction and natural language processing. The programming language community developed a vast amount of knowledge about deterministic (LL(k), PLR(k), LR(k), LALR(k), operator precedence, recursive descent ...) parsers. For analyzing natural language, these parsers are not useful. Instead, for the latter purpose, a large number of general parsing algorithms have been used. Among them are the CYK, Earley, chart, Sheil, and Tomita parsers.[1] The scientific communities of computational linguistics and compiler theory are rather different. It is my experience that the average researcher in either area underestimates the problems on the other side of the fence. One professor of computer science, for instance, once confided to me that it escaped him why "beautiful compiler construction tools like parser generators (he mentioned a specific one) are not being used for natural language processing." This professor had never realized that there are differences between artificial languages and natural languages that have grave consequences for parsing, such as the fact that natural language sentences are syntactically ambiguous.

This book might help bring both parsing communities closer together. In any case, it brings together the techniques that are used on either side. The current state of parsing theory reflects the status quo regarding the two contributing communities. It suffers from a dichotomy that does not befit a mature field of knowledge. Parsers in compilers are typically implemented as deterministic push-down automata, the central data structure of which is a so-called *parse stack*. General parsing algorithms are mostly tabular, which means that a *parse matrix* is the central data structure. In this book, by contrast, deterministic and general parsing algorithms are treated in a unified fashion. This is accomplished by adopting a functional formulation of

[1] References are described in the last chapter.

parsing theory. In the new theory, factors that distinguish various parsing algorithms, such as stacks and parse matrices, are banned. Stacks are replaced by recursive functions, and parse matrices by memoizing functions (functions that remember past invocations). Along the way, some deficient parts of the existing body of knowledge are identified and repaired. A notable example of such a deficiency in the standard theory is the absence of simple functional implementations of LR parsers.

Many books on parsing theory are dedicated to the study of many classes of grammars (such as LL(k), LR(k), ...). Such a class is determined by the requirement that a corresponding parser behaves deterministically. In this book, whether or not a parser is deterministic is considered to be a marginal question. Our emphasis is on algorithms, not on grammar classes. Our main topic is theory, but the application of the theory is never far away, and the results are of direct practical relevance. Parsing theory is presented in a mathematical way, but the style of mathematics is not too rigorous. The correctness of most algorithms is formally established, with proofs of calculational nature.

The theory presented in this book leads to a new technique for implementing parsers, which has been named *recursive ascent parsing*. The history of recursive ascent parsing is quite interesting. The standard formulation of LR parsing uses the concepts of automata theory. Looking for efficient implementations, Pennello came up with a technique borrowed from efficient recursive descent implementations, and in this way created, in a hidden way, the first recursive ascent LR parser. Based on this work, Roberts presented an implementation at a higher level of abstraction. Independently from Roberts and Pennello, Kruseman Aretz and Barnard and Cordy almost simultaneously proposed very similar ideas, with motivations of a theoretical kind. Their starting point was the standard LR parser. Figure 0.1 is a pictorial representation of this history and displays the work reported here at the level of automata theory. More precisely, our theory is the functional equivalent of the theory of nondeterministic pushdown automata (NPDA), the class of automata that recognizes exactly the context-free languages. The basic idea is simple but extraordinary powerful: a NPDA *state* is 'implemented' as a *function*. Then, state *transitions* correspond to function *calls*, and stack *pops* to function *returns*.

The expert reader may be surprised when he or she finds his favorite parsing algorithm described in this book. LR parsers, for example, are defined not only without parse stacks but also without parsing tables. The functional approach to parsing requires a new way of thinking, and in this respect it may

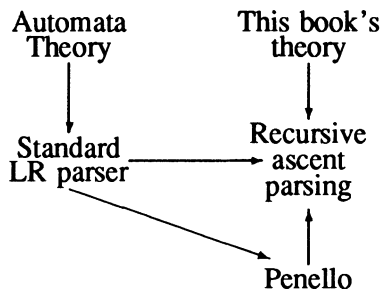

Figure 0.1 History of recursive ascent parsing

sometimes be an advantage to be unacquainted with standard approaches.

People who love functional programming will enjoy the new applications of this style of writing algorithms. For others, the same style may be a (hopefully temporary) stumbling block. For that reason, every now and then an example is given in an imperative style that can be translated into any imperative programming language without much ado. The reader is advised to actually do so whenever the level of abstraction becomes a problem.

My hope is that this book will be of use to students, teachers, scientists, and programmers. Parsing theory does not need heavy mathematics, and only some standard mathematics skills are presupposed. The book is self-contained. References to the literature are avoided in the content chapters. The last chapter is contemplative and contains many bibliographic notes. The results are presented without long explanatory elaborations, but this could make the book a bit terse for students. As an encouragement, it may be said that working through this book is a quick way to become an expert on parsing theory. The book can be used to teach parsing, but it also contains interesting topics to touch upon in courses on formal language theory, compiler construction, functional programming, computational linguistics, or program derivation.

This book was written to explain the new theory of chapters 6 to 9. The earlier chapters are a prelude, providing background knowledge that is useful for building up some intuition about recursive ascent parsing algorithms.

We start with an introduction to context-free grammars. Various interpretations of context-free grammars are given. One such interpretation involves a mapping from grammar symbols to multiple-valued functions, and leads directly to the recursive descent parsing method. Since this method has some limitations, we are led to study grammar transformations as a way to adapt arbitrary grammars to these limitations. Historically, many results of chapters 6 and 9 have been obtained by grammar transformations. However, chapters 6 to 9 do not depend on all of the preceding text, and in particular they *can* be understood without knowing about grammar transformations. One may therefore skip chapter 5 on the first reading of this book, if one is not interested in what is "behind it all".

In practical applications, context-free grammars are often used in conjunction with some form of attribute evaluation. Chapter 8 treats attribute grammars functionally, in such a way that parsing with respect to attribute grammars is a simple extension to pure context-free parsing. Two attribute grammar formalisms are presented. One is typically useful in compiler technology, where attributes are used for semantic purposes. The other is meant for applications in which attributes play an important syntactic role, as in natural language parsing.

I am indebted to my family for letting me work so many nights, and to various colleagues at Philips Research for having been instrumental to this work. Jan Landsbergen stimulated me to express my views in a book. Frans Kruseman Aretz invented recursive ascent parsing and through his stimulating, if critical, comments contributed to this work in its early stages. Lex Augusteijn profoundly influenced both my ideas and the way they appear in this book. I thank Jaime Carbonell, Theo Norvell, Fernando Pereira, Wim Pijls and three anonymous referees for their suggestions and for finding errors.

René Leermakers.

1

CONTEXT-FREE GRAMMARS

Context-free grammars are a language for defining languages. Not all languages can be defined by a context-free grammar - only the (yes) context-free ones. Suppose we want to define the language of a small child, who continually says sentences like "want cookie". His or her sentences consist of a verb followed by a noun. A context-free grammar rule expressing this is

$$< sentence > \rightarrow < verb > < noun > .$$

Here the symbols between angular brackets, i.e., $< sentence >$, $< verb >$ and $< noun >$, are *grammar symbols*, of which $< sentence >$ is the *start* symbol. Instead of $< sentence >$, the symbol S is commonly used as the start symbol, and we will also use this convention from now on. Suppose our child grows older and starts saying "want yellow dress." This means that his or her sentences contain complex noun phrases like "yellow dress," which are composed of an adjective and a noun. Our grammar may grow, with the child, as follows:

$$S \rightarrow < verb > < nounphrase >,$$
$$< nounphrase > \rightarrow < noun >,$$
$$< nounphrase > \rightarrow < adjective > < noun > .$$

This grammar says that a sentence is a verb followed by a noun phrase. A noun phrase in its turn either is a noun or is an adjective followed by a noun. If the first rule for $< nounphrase >$ is used in combination with the rule for S, one gets sentences like "want dress". The second rule for $< nounphrase >$ leads to "want yellow dress". Still older, the child masters

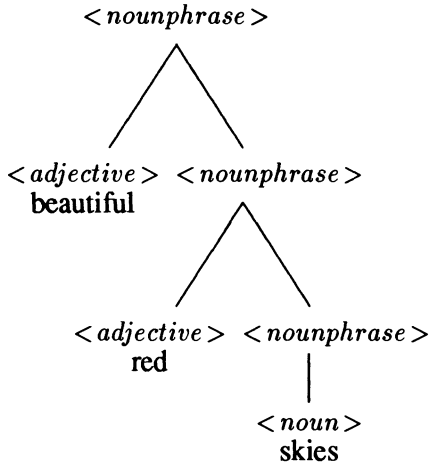

Figure 1.1 The structure of "beautiful red skies"

recursion; he or she starts using any number of adjectives before a noun: "Daddy paints beautiful red skies." This means that a noun phrase can be an adjective followed by (here is the recursion) a noun phrase. A corresponding grammar is (by now the child uses noun phrases as subjects of its sentences)

$$\begin{aligned} &S \rightarrow < nounphrase >< verbphrase >, \\ &< verbphrase > \rightarrow < verb >< nounphrase >, \\ &< nounphrase > \rightarrow < noun >, \\ &< nounphrase > \rightarrow < adjective >< nounphrase > . \end{aligned} \tag{1.1}$$

The noun phrase "beautiful red skies" is now dissected into the adjective "beautiful" and the noun phrase "red skies" by the second grammar rule which has the left-hand side $< nounphrase >$. Subsequently, "red skies" is, by the same rule, the adjective "red," followed by the noun phrase "skies." Lastly, the noun phrase "skies" is analyzed as being a noun by the first grammar rule for $< nounphrase >$. Such phrase analysis can be summarized nicely by the picture of a syntax tree, in this case figure 1.1.

The main verb need not always be followed by a noun phrase, as in "I paint." One could add this possibility to grammar (1.1) by adding the rule

$$< nounphrase > \rightarrow \epsilon,$$

where ϵ denotes "nothing," the empty string of grammar symbols. The grammar rule says that a noun phrase can be empty. Adding this so-called ϵ-rule to our last grammar implies that sentences also need not have a subject. This is true: imperative sentences have no overt subject. The reader is warned not to conclude from our little exercise that natural language grammars are easy to come by. Though writing a small grammar for a fragment of some natural language is often simple, writing comprehensive grammars is cumbersome.

The above makes it clear that, given a context-free grammar, it can be decided whether or not a sentence or phrase is well formed. A program that makes such judgments is called a *recognizer*. If a sentence or phrase is well formed, its structure may be established in the form of a syntax tree. A program that combines the recognition task with the production of syntax trees is called a *parser*.

A rule of a context-free grammar is often called a rewriting rule: it tells you how to rewrite the left-hand side of the rule. Accordingly, a context-free grammar is a *rewriting system*. In a rewriting system, structures (such as sentences) are obtained by *derivations*, which go as follows. The first step is that the start symbol is rewritten, i.e., replaced by the right-hand side of one of its rewriting rules. (In our example there is only one rule for the start symbol S, so we get $< nounphrase >< verbphrase >$.) In the result, the left-hand side of a rule is looked for (a grammar symbol in the case of context-free grammars) and replaced by the right-hand side of one of its rules (a string of grammar symbols in the case of context-free grammars), et cetera, until nothing can be rewritten anymore. As a rewriting system, a context-free grammar is one in a large family: one can rewrite strings, formulas, trees, graphs, shapes, et cetera.

The set of grammar symbols is denoted by V. It has a subset V_N of *nonterminal* symbols. A nonterminal symbol can be rewritten: it is the left-hand side of some grammar rule. The other grammar symbols are *terminal* symbols: they cannot be rewritten. The set of terminal symbols is V_T. The right-hand sides of grammar rules are (possibly empty) strings of grammar symbols. The set of all such strings is denoted by V^*. Formally, one has

- $\epsilon \in V^*$,
- $V \subset V^*$,
- if $\alpha \in V^*$ and $\beta \in V^*$, then $\alpha\beta \in V^*$,

and similar definitions for V_T^*, V_N^*. The length of a string α, i.e., the number of grammar symbols, is denoted by $|\alpha|$. This length is zero in the case of the empty string ϵ. One has $\alpha\epsilon \equiv \epsilon\alpha \equiv \alpha$, for all $\alpha \in V^*$.

When talking about arbitrary grammars, one needs variables. We use typed variables: a given variable always denotes the same type of entity. The following are our conventions:

$$\begin{array}{lll} A, B, C \in V_N & x, y, z \in V_T & X, Y, Z \in V \\ \alpha, \beta, \gamma, \delta, \mu, \nu, \lambda \in V^* & \xi, \eta, \zeta, \rho \in V_T^* & \end{array}$$

Thus, $A \to \alpha$ is an arbitrary grammar rule and $A \to X\alpha$ is an arbitrary grammar rule with nonempty right-hand side. We use Greek letters as variables over strings of grammar symbols, but we also use an alternative notation: one can write string variables with the help of an ordered set of variables $X_i \in V$, with integer i. A string $X_{i+1}...X_j$ denotes elements of V^* with length $j - i$. It is therefore natural to let expressions of the form $X_{i+1}...X_i$ denote strings with zero length, i.e., ϵ. One may easily verify that, with this convention,

$$X_{i+1}...X_j = X_{i+1}...X_k X_{k+1}...X_j \tag{1.2}$$

is true for all $i \leq k \leq j$.

Here is another example of a context-free grammar. It has $V_N = \{S\}$ and $V_T = \{x, y\}$ and grammar rules

$$\begin{array}{l} S \to xy, \\ S \to xSy. \end{array}$$

Application of only the first rule gives us that the string xy is produced by the grammar. If the second rule is applied first and then the first one, the string $xxyy$ is produced, and so on. The language produced by this grammar consists of all finite strings of any number of x's followed by the same number of y's.

The above gives already a fairly complete picture of context-free grammars - complete enough, in fact, for understanding much of this book. Chapter 3

describes the same concept in a pedantic way. It stresses the natural semantics of context-free grammars, in which grammar symbols are interpreted as (variables over) languages: $< nounphrase >$ is the language of all noun phrases, $< verb >$ is the language of all verbs, et cetera. In this interpretation, grammar rule $A \rightarrow XY$ means that if one has a sentence of language X and another one of language Y, the concatenation of the two is a sentence of language A. The nice thing about the natural semantics is that a rule can be understood locally: its meaning does not depend on the rest of the grammar. Interpretations with such property are often called *declarative*.

But first we need a bit of nonstandard mathematics: bunch notation.

2

BUNCH NOTATION

A context-free grammar can be interpreted formally in a number of ways. This chapter prepares the ground for discussing such interpretations. It gives a variant of set notation, called *bunch notation*. There are two major differences between sets and bunches. One difference is that a bunch with one element is identified with that one element (the singleton property). Moreover, a function or operator that is defined on some domain may be applied to a bunch of elements that belong to that domain. Such an application, say $f(x)$, causes the function f to be applied to each separate element of the bunch x, after which the results are combined in a bunch, which is the result of $f(x)$. This is called the distributivity property of bunches.

In this book, bunches are used for various purposes. Firstly, a language is defined as a bunch rather than as a set. Secondly, multiple-valued functions are implemented as bunch-valued functions. The reason for replacing sets, in some contexts, by bunches is a matter of elegance, not of principle. Bunches are used whenever they simplify notation. The singleton property of bunches simplifies the formalization of the natural interpretation of grammars and smoothes the transition from parsing algorithms for general grammars to deterministic algorithms for subclasses of grammars. The distributivity property also simplifies the discussion on the natural interpretation. Moreover, this property makes it possible to write succinct formulae in bunch notation that "blow up" if translated into set notation.

Another advantage of the notation that we will develop for defining bunch-valued functions is that it resembles traditional notation for specifying programs. In particular, we will make extensive use of a construct akin to Dijkstra's guarded commands (see [Dijkstra,1976]).

2.1 BUNCHES

In standard mathematics, a (total) *function* f with domain $\mathcal{A}$ and range $\mathcal{B}$ associates one element of $\mathcal{B}$ with each element of $\mathcal{A}$. A function is a special case of a *relation*, which may associate any number of elements of $\mathcal{B}$ with each element of $\mathcal{A}$. Conversely, each relation can also be seen as a special kind of function: a set-valued function that, if applied to $a \in \mathcal{A}$, yields the set of elements of $\mathcal{B}$ associated with a by the relation. Alternatively, a relation may be viewed as a nondeterministic function: of all elements of $\mathcal{B}$ related to some element $a \in \mathcal{A}$, the nondeterministic function arbitrarily picks one.

The set-valuedness of functions associated with a relation has one peculiar consequence. Take a function $f : \mathcal{A} \mapsto \mathcal{B}$, and view it as a relation with the special property that it relates only one element of $\mathcal{B}$ to each element of $\mathcal{A}$. Next, use the mapping from relations to set-valued or nondeterministic functions to view the relation as a function again. Then one would expect to reobtain the original function f. If relations are mapped to nondeterministic functions, this is indeed the case: the function that corresponds to the relation associated with f happens to be deterministic and is equal to f. Using standard sets, however, the set-valued function that corresponds to the relation associated with f produces a set with exactly one element (a singleton) where f produces that element. This suggests that it is better not to see relations as set-valued functions, but rather as *bunch*-valued functions. A bunch is a set with some nonstandard properties, so that it can be interpreted in an alternative way: a bunch is also a process that nondeterministically produces one of its values. The alternative interpretation implies the following three properties of bunches:

1. The process that corresponds to a bunch with one value (a singleton) is deterministic: it can only produce that one value. Therefore: a singleton is identified with its only element.

2. The process that corresponds to a bunch produces definite values. So: elements of bunches cannot be bunches with cardinality $\neq 1$.

3. If f is a function and x is a bunch that can produce the values $e_1 ... e_k$, then $f(x)$ can take the values $f(e_1) ... f(e_k)$. Therefore: functions distribute over bunches.

With these properties, a bunch simultaneously allows two interpretations. In

the set-valued interpretation, it is just a collection of values. In the nondeterministic interpretation, one value is randomly taken out of this collection.

Bunches are the result of bunch expressions. Given two bunch expressions x and y, their bunch union $x|y$ denotes a process that could either produce a value of x or a value of y. Bunch union has the same properties as set union: it is associative, commutative, and idempotent. The main difference with sets is that a bunch is not "one thing" if it has more than one element. This is why a bunch with many elements cannot be one element of another bunch: it can only be many elements of another bunch. This is also why a bunch with many elements cannot be passed to a function or operator as one thing. Here are a few examples of equalities for bunch expressions that illustrate the above:

$$3 + (1|2) \equiv 4|5$$
$$(3|4) + (1|2) \equiv 4|5|5|6 \equiv 4|5|6$$
$$cos(\pi|0) \equiv -1|1$$
$$(1|2) > 3 \equiv false|false \equiv false$$

If e is one of the values a bunch expression x can take, we write $e \leftarrow x$. Here and henceforth, e is a definite value or, what is the same, a singleton bunch. Since definite values are also bunches, and elements of bunches cannot be bunches with cardinality $\neq 1$, the distinction between $\in$ and $\subseteq$ is no longer needed: if, for all e, $e \leftarrow x$ implies $e \leftarrow y$ then we write $x \leftarrow y$. In words, x is a subbunch of y, or, x is smaller than y. Below we will need the following property:

$$\text{Consider bunch expressions } z \text{ such that } x \leftarrow z \wedge y \leftarrow z; \text{ then the smallest of all such } z \text{ is given by } z = x|y. \tag{2.1}$$

Bunch expressions can be simple or complex. The simplest simple bunch expression is the empty bunch *null*. It is the identity of bunch union. Other simple bunch expressions are enumerations. The following is a formal definition of simple bunch expressions with elements from a (possibly infinite) set of definite values:

1. *null* is a simple bunch expression,
2. if e is a definite value, then e is a simple bunch expression,

3. if x and y are simple bunch expressions, then $x|y$ is a simple bunch expression.

A simple bunch expression may be rewritten into an equivalent expression using $(x|y)|z \equiv x|(y|z) \equiv x|y|z$, $x|y \equiv y|x$, $x|x \equiv x$, and $x|null \equiv x$. Assuming some ordering on the set of definite values, it is not difficult to define a canonical form for each simple bunch expression, which may serve as a unique representation of the bunch denoted. The bunch expression *all* denotes the smallest bunch such that $e \leftarrow all$ for all definite values e.

Complex bunch expressions are constructed with variables. Unless stated otherwise, variables have types that consist of definite values only. Such variables are called definite; they cannot be bound to bunches with cardinality greater than one. Given a proposition P and bunch expressions x and y, the expression

$$\textbf{if } P \textbf{ then } x \textbf{ else } y \tag{2.2}$$

is a complex bunch expression. It contains free variables if P, x, or y contain free variables (nontrivial P do). As will be clear, for each assignment of values to the variables, the complex expression (2.2) is equivalent to x if P is true and to y otherwise. Free variables in bunch expressions can be bound by λ-abstraction. If i is a variable and x is a bunch expression, then

$$\lambda i \cdot x$$

is a bunch-valued function. For any definite value e,

$$\lambda i \cdot x(e) \stackrel{def}{\equiv} \text{Substitute } e \text{ for free variables } i \text{ everywhere in } x.$$

This definition holds only for definite values e. Note, therefore, that it is important to distinguish between functions and expressions. In expression x in $\lambda i \cdot x$, variable i may occur more than once. If function $\lambda i \cdot x$ is applied to a bunch y, then the distributivity of functions over bunches means that the function applies to each $e \leftarrow y$ separately. That is, if $x = i*i$ then $\lambda i \cdot x(2|3)$ produces $2*2 \mid 3*3$, which is not the same as $(2|3) * (2|3)$. In jargon, our functional language is characterized as having a semantics such that functions

are not *unfoldable*: a function invocation cannot be textually replaced by the expression that defines the function, if function parameters are not definite. This is also called *call time choice* semantics of functions.

This is about all we need to know about bunches. We just add some notations:

$$P \rhd x \stackrel{def}{=} \textbf{if } P \textbf{ then } x \textbf{ else } \mathit{null},$$
$$\textbf{let } i \cdot x \stackrel{def}{=} \lambda i \cdot x(\mathit{all}).$$

The bunch *all* will in general be infinite, so a function that distributes over it might produce an infinite bunch as well. In our application, however, the structure of bunch expressions will be such that **let**'s produce only finite bunches.

We will repeatedly use the laws

$$(P_1 \wedge P_2) \rhd x \equiv P_1 \rhd (P_2 \rhd x), \tag{2.3}$$

$$(P_1 \vee P_2) \rhd x \equiv (P_1 \rhd x)|(P_2 \rhd x), \tag{2.4}$$

$$\textbf{let } i \cdot (i \leftarrow x \rhd f(i)) \equiv f(x). \tag{2.5}$$

In (2.5) it is assumed that i does not occur free in x. These laws are easy to prove: the first two follow from the definition of $\rhd$, and the third follows from the distributivity of function application over bunches.

Normally, set-valued functions are defined using set comprehension according to the schema

$$f(X) = \{A(X,Y) | \exists_Z P(X,Y,Z)\}, \tag{2.6}$$

where P is a predicate, A is a function, and X, Y, Z are variables or sets of variables. Now let us define a related bunch-valued function, called f_b:

$$f_b = \lambda X \cdot (\textbf{let } Y \cdot (\textbf{let } Z \cdot (P(X,Y,Z) \rhd A(X,Y)))). \tag{2.7}$$

It follows that f and f_b are equivalent if the latter is interpreted as producing a set. A nice aspect of (2.7) is that its algorithmic content is more explicit than the algorithmic content of (2.6); because **let** is defined as a function application to *all*, it is explicit that (2.7) involves searching over all values of Y, Z.

In this book we will use a notational convention that removes the λ's and the **let**'s from definitions such as (2.7). Instead of (2.7) we write

$$f_b(X) = \quad P(X,Y,Z) \rhd A(X,Y). \tag{2.8}$$

So $\lambda X \cdot$ has been changed into a formal argument on the left-hand side, and we take the convention that free variables on the right-hand side (here Y, Z) are bound by **let**'s. The scope of such an implicit **let** is in practice always clear: usually it is from the first occurrence of the variable until "|", or else until the end of the bunch expression. Thus, whenever an expression $P \rhd x$ is encountered in this book, with some free variables, its meaning is that all possible values of the free variables must be tried to make the guard P true and all results x must be combined in one bunch. An example that is typical for our application is the function

$$f(X) = \quad X \rightarrow \alpha \rhd \alpha.$$

Here α is a free variable, so there is an implicit **let** over α. The function produces the bunch of all right-hand sides of grammar rules with left-hand side X. If one takes the grammar (1.1), one has, for instance,

$$f(< nounphrase >) =< noun > \mid < adjective >< nounphrase >,$$
$$f(< noun >) = null.$$

2.2 ALGORITHMIC INTERPRETATION

In this section we describe how definitions of bunch-valued functions should be interpreted algorithmically. With this interpretation it is straightforward to implement any of the algorithms developed in this book in imperative programming languages. Though it would be possible to give a thorough and formal treatment of the computational properties of bunch expressions,

we refrain from doing so. The description that will be given is sufficiently complete for practical purposes.

In the following, bunch-valued functions either are known computable functions or have definitions with the following general format:

$$
\begin{aligned}
f(X) = \quad & P_1(X,Y_1) \rhd A_1(X,Y_1) \mid \\
& P_2(X,Y_2) \rhd A_2(X,Y_2) \mid \\
& \quad \vdots \\
& P_k(X,Y_k) \rhd A_k(X,Y_k),
\end{aligned}
$$

where X is a set of input parameters and Y_i are sets of variables subject to **let** quantification. P_i are predicates and A_i are bunch-valued functions. Both P_i and A_i may involve other applications of bunch-valued functions. Unless stated otherwise, the intention is that bunches are interpreted as collections. Function f then has a simple algorithmic (imperative) interpretation that makes use of a bunch-valued variable "result":

```
f(X) =  result:=null;
        for all Y1 such that P1(X,Y1) do
            result:=result | A1(X,Y1)
        od;
        for all Y2 such that P2(X,Y2) do
            result:=result | A2(X,Y2)
        od;
            .
            .
            .
        for all Yk such that Pk(X,Yk) do
            result:=result | Ak(X,Yk)
        od;
        return result.
```

The invocations $A_i(X,Y_i)$ and function applications inside P_i are to be computed in the same vein. In this algorithmic interpretation, a function may or may not terminate. If it does not terminate, $f(X)$ does not define an algorithm. This may happen if the definition of $f(X)$ is circular, i.e., if the computation of some P_i or A_i involves $f(X)$ again. In our application the

predicates P_i are such that for every X it can be determined constructively which values for Y_i satisfy $P_i(X, Y_i)$.

The above function f is *deterministic* if, for each X, at most one proposition $P_i(X, Y_i)$ can be true, for only one value of Y_i, and function A_i is deterministic. If one wants to prove that a recursively defined function f has some property, e.g. that it implements some specification or that it is deterministic, one may do so inductively:

1. Assume that f has the property if applied to the arguments with which it occurs in the defining expression of itself.
2. Show that this assumption entails that f has the property for any value of its formal parameters.

This reasoning may be circular, but if it is, f will not terminate. Therefore, provided that f terminates, the above inductive step suffices to prove that f has the required property.

Lastly, note that the equivalences expressed by the laws (2.3) and (2.4) refer to the bunches denoted by the equivalent expressions, and not to the algorithmic interpretation of these expressions. On the contrary, the purpose of these laws is to change the algorithmic interpretation. A similar phenomenon is that (2.8) stands for the same bunch as

$$f_b(X) = \exists_Z(P(X, Y, Z)) \rhd A(X, Y), \tag{2.9}$$

although the algorithmic interpretation of both formulae is not exactly the same. In this case, however, it will be convenient to associate with (2.8) the algorithmic interpretation of (2.9). In other words, if a variable appears only in a guard, like Z in (2.8), we take the convention that it is implicitly subject to existential quantification.

3

GRAMMAR INTERPRETATIONS

We are equipped now to give a presentation of context-free grammars that is a bit more formal than that of the first chapter. At first, a context-free grammar is only some four-tuple, without any meaning. This formal system is brought to life (is given an interpretation) via a mapping from the elements of the formal system to some domain. In the natural interpretation, we map grammar symbols to languages and grammar rules to inequalities.

From the natural interpretation, rewritings (or derivations) are presented as a notion needed for constructing proofs. In the interpretation of grammars as rewriting systems, a nonterminal grammar symbol is simply a bookkeeping device.

An important third interpretation of context-free grammars is that of abstract programs, via a homomorphism from grammar symbols to function spaces.

3.1 THE NATURAL INTERPRETATION

Within the family of rewriting systems, context-free grammars have a distinguishing property: they have a declarative meaning. This means that a grammar can be understood not only by producing a sample of trial sentences with it but also by viewing it as a collection of static statements about the language that is to be defined. This is the underlying reason for the intelligibility and usefulness of context-free grammars. It is also the reason for putting, as we do, the natural interpretation first and deriving the rewriting interpretation from it. In the natural interpretation, grammar symbols

are seen as variables over languages and grammar rules as stipulations of relations between these variables. Together, these relations put enough constraints on the possible values of the variables that they may be determined, albeit not entirely uniquely. A grammar, in this view, is analogous to a set of arithmetic inequalities with variables. Take, for instance, the following inequalities:

$$k \geq l + 3,\ l \geq 5.$$

A formal interpretation of this is that there are two symbols k and l here, and that there is some assignment function h from these symbols to numbers, and that the inequalities restrict the possible values of h, via

$$h(k) \geq h(l) + 3,\ h(l) \geq 5.$$

Of course, there are still many functions h that satisfy these constraints, but there is one that assigns the smallest possible numbers to the symbols: $h(k) = 8, h(l) = 5$.

The formal system

A context-free grammar is a four-tuple $G = (V_N, V_T, P, S)$, where S, V_N, V_T where introduced above and P composes the set of grammar rules. A grammar rule is a pair (A, α), which is said to be a grammar rule for nonterminal A. Remember that $V = V_N \cup V_T$.

Languages

A language is a bunch of strings of terminals, i.e., a bunch of elements of V_T^*. Concatenation is an operation that is defined for (pairs of) strings. Therefore, it distributes over languages L and M, if they are concatenated:

$$LM = \ \xi \leftarrow L \wedge \rho \leftarrow M \ \rhd \xi\rho. \tag{3.1}$$

This equation is referred to as the definition of *language multiplication*, although it is not really a definition: it follows from the distributivity property.

The interpretation

A nonterminal can be seen as a variable of type language (like k, l are variables of type integer); a terminal is a constant language (like 3,5 are constant integers). Just as in the arithmetic example, we assume an assignment function that performs the mapping from symbols to their interpretation. We call this function L_G, since its value will be determined by the grammar G. Take, for example, the grammar rule (S, xSy). In the natural interpretation this rule means

$$xL_G(S)y \leftarrow L_G(S).$$

That is, the grammar rule is a constraint on L_G. In principle, L_G need only apply to nonterminals, but it is convenient to extend it, via

$$L_G(x) = x, \text{ for all } x \in V_T,$$

to all grammar symbols. Moreover, we further extend it to arbitrary strings of grammar symbols, via

$$\begin{aligned} &L_G(\alpha\beta) = L_G(\alpha)L_G(\beta), \\ &L_G(\epsilon) = \epsilon, \end{aligned} \tag{3.2}$$

so that the following equalities hold true:

$$xL_G(S)y = L_G(x)L_G(S)L_G(y) = L_G(xSy).$$

Equation (3.2) expresses that L_G not only maps grammar-symbol strings into languages: it also maps an operation on its input objects (concatenation) to an operation on its output objects (language multiplication). Mappings that have such property are called *homomorphisms*. Keep in mind that it is sufficient to show how L_G is defined for nonterminals.

The interpretation of any grammar rule (A, α) now reads

$$L_G(\alpha) \leftarrow L_G(A).$$

In other words, the language associated with A and the language associated with α are related: the latter is a subbunch of the former.

The smallest solution

Given a grammar G, the restrictions that correspond to grammar rules do not determine L_G uniquely. However, there is always one *smallest* L_G: the one that assigns the smallest possible languages to grammar symbols, while satisfying all constraints. Without further qualifications, *the* homomorphism L_G is this smallest one. The language of primary interest is $L_G(S)$, the language that corresponds to the start symbol. For most grammars, $L_G(S)$ is an infinite bunch and therefore cannot be computed explicitly. The most one can do is solve the *recognition* problem associated with G. A solution to this problem is a method for determining whether or not an arbitrary string is an element of $L_G(S)$.

An example

Specifying a grammar amounts to listing the set of grammar-rule constraints. If, for example, $P = \{(S, xy), (S, xSy)\}$, these constraints are

$$L_G(xy) \leftarrow L_G(S),\ L_G(xSy) \leftarrow L_G(S).$$

This simple grammar has only one nonterminal S and two terminal symbols, x and y. To formally figure out the language associated with this grammar, we first point out that it follows from the grammar's constraints that

$$L_G(x^n y^n) \leftarrow L_G(S), \tag{3.3}$$

for any $n \geq 1$ (x^n is the string of n x's). The proof is by induction. If $n = 1$, (3.3) is simply the first grammar-rule constraint. The induction hypothesis is that (3.3) holds true for some positive n. Substituting (3.3) in the second grammar rule, one gets

$$L_G(xx^n y^n y) \leftarrow L_G(xSy) \leftarrow L_G(S), \tag{3.4}$$

and hence, $L_G(x^{n+1} y^{n+1}) \leftarrow L_G(S)$. Because L_G maps terminals to themselves, it follows that $x^n y^n \leftarrow L_G(S)$, for any $n \geq 1$. So, either

$$L_G(S) = \quad n \geq 1 \ \triangleright x^n y^n \tag{3.5}$$

or $L_G(S)$ is something bigger. To see whether (3.5) holds true, we must check whether it is consistent with the constraints. Substituting it in the constraints, the resulting inequalities

$$xy \leftarrow (n \geq 1 \ \triangleright x^n y^n),$$
$$(n \geq 2 \ \triangleright x^n y^n) \leftarrow (n \geq 1 \ \triangleright x^n y^n)$$

are clearly valid. Hence, (3.5) holds true.

The derivation of (3.4) is an instance of a *cut-rule* application. A general definition of this rule is

$$\text{if } L_2 L_3 L_4 \leftarrow L_1 \text{ and } L_5 \leftarrow L_3 \text{ then } L_2 L_5 L_4 \leftarrow L_1. \tag{3.6}$$

The recognition problem

The example illustrates the general mechanism. From the grammar-rule constraints one may obtain derived constraints using the cut rule (3.6). Note that the cut rule implies transitivity of bunch inclusion (take $L_2 = L_4 = \epsilon$), so that transitivity is not an independent property that could be used to obtain derived constraints. If $L_G(\beta) \leftarrow L_G(\alpha)$ is a constraint derived from the grammar rules with the cut rule, we write $\alpha \geq^G \beta$. For example, if $(A, \alpha B \beta) \in P$ and $(B, \gamma) \in P$, then we write $A \geq^G \alpha B \beta$, $B \geq^G \gamma$, but we may also substitute the latter into the former and conclude $A \geq^G \alpha\gamma\beta$. Because $L_G(\alpha) \leftarrow L_G(\alpha)$ for all G, α, one has that $\alpha \geq^G \alpha$ for all G, α. Hence, $\geq^G$ is a reflexive partial order on V^*. If $A \geq^G \xi$, then clearly $\xi \leftarrow L_G(A)$. Since the cut rule is the only means by which conclusions $\xi \leftarrow L_G(A)$ can be drawn from the rule constraints, it follows that for each A,

$$L_G(A) = \quad A \geq^G \xi \ \triangleright \xi$$

states the smallest possible value of $L_G(A)$ consistent with the constraints.

3.2 DERIVATION

The process of deriving propositions $\alpha \geq^G \beta$ with the cut rule can be formalized by introducing the *derives* relation $\rightarrow$:

$$\alpha \rightarrow \beta \equiv \exists_{\delta\gamma A\mu}(\alpha = \delta A\gamma \wedge \beta = \delta\mu\gamma \wedge (A,\mu) \in P). \tag{3.7}$$

The reflexive and transitive closure of the derives relation, denoted by $\xrightarrow{*}$, is defined recursively as

$$\begin{aligned} \alpha \xrightarrow{*} \beta &\equiv (\alpha = \beta) \vee (\alpha \xrightarrow{+} \beta), \\ \alpha \xrightarrow{+} \beta &\equiv \exists_\gamma(\alpha \rightarrow \gamma \wedge \gamma \xrightarrow{*} \beta). \end{aligned} \tag{3.8}$$

Verbalized, this means that $\alpha \xrightarrow{*} \beta$ if and only if either $\alpha = \beta$ or β can be derived from α using one or more grammar rules. By construction, one has the equivalence

$$\alpha \geq^G \beta \equiv \alpha \xrightarrow{*} \beta.$$

If $\alpha \xrightarrow{*} \beta$ we say that α rewrites to β, and we describe $\alpha \rightarrow \beta$ by saying that α rewrites *directly* to β.

Above, we introduced context-free grammar rules with arrow notation. In the previous section, we shifted to the more formal pair notation. It follows from (3.7) that $A \rightarrow \mu$ is synonymous to $(A,\mu) \in P$. In the following we will write rules again with the more convenient arrow notation, i.e., as direct rewritings. A consequence is that we will no longer refer to P explicitly: all formulae in which the derives relation appears are defined with respect to an implicit reference grammar.

Rewriting

A proof of $\alpha \xrightarrow{*} \beta$ is often represented as a *derivation*, usually a *leftmost* derivation. A derivation is a series of *rewritings* of strings of grammar symbols. Rewriting a string β with rule $A \rightarrow \alpha$ means finding an A in β and replacing it by α. In a next step, the result can be rewritten again, and so

on. In a leftmost derivation, at each step the leftmost nonterminal is being replaced. Formulated in rewriting terminology, the recognition problem is to decide whether or not S can be rewritten, in any number of steps, into $x_1 ... x_n$. An example of a (leftmost) derivation is $S \rightarrow AB \rightarrow xB \rightarrow xCz \rightarrow xyz$, where the rules $S \rightarrow AB$, $A \rightarrow x$, $B \rightarrow Cz$, $C \rightarrow y$ are applied in this order. There are other, more or less equivalent, ways to represent derivations. In the following two paragraphs, we discuss such alternatives.

Parse trees

The most interesting derivations are the ones starting from a nonterminal, i.e., ones that prove $A \xrightarrow{*} \beta$. Such a derivation has the following structure. According to (3.8), $A \xrightarrow{*} \beta$ is true iff either $A = \beta$ or there is a rule $A \rightarrow \gamma$ and one can prove $\gamma \xrightarrow{*} \beta$. Now, let

$$\gamma = X_1 ... X_k \text{ and } \beta = Y_1 ... Y_l.$$

Then, a proof of $\gamma \xrightarrow{*} \beta$ can be split into k subproofs, using (1.2):

$$\begin{array}{l} X_1 \xrightarrow{*} Y_1 ... Y_{i_1} \\ X_2 \xrightarrow{*} Y_{i_1+1} ... Y_{i_2} \\ \quad ... \\ X_k \xrightarrow{*} Y_{i_{k-1}+1} ... Y_l \end{array}$$

where $0 \leq i_1 \leq i_2 \leq ... i_{k-1} \leq l$. In other words, a proof of $A \xrightarrow{*} \beta$ is trivial if $A = \beta$, and else it consists of selecting a grammar rule for A and giving proofs that the grammar symbols that appear on the right-hand side of this rule derive consecutive parts of β. A derivation of $A \xrightarrow{*} \beta$ thus has a tree structure and in fact is often depicted as a so-called *syntax* tree, or (synonymously) a *parse* tree. An example is figure 3.1, which represents the same derivation that was just given as a series of rewritings.

Rule applications

Derivational proofs can also be represented as sets of *rule applications*, which are "dressed" grammar rules. An example of a rule application is

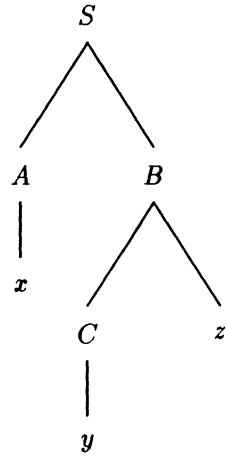

Figure 3.1 Syntax tree

$$(A, \eta\rho) \rightarrow (X, \eta)(Y, \rho),$$

which is an application of $A \rightarrow XY$ and says that $\eta\rho \leftarrow L_G(A)$ is deduced from $\eta \leftarrow L_G(X)$ and $\rho \leftarrow L_G(Y)$. It will be clear how this extends to non-binary rules (see also chapter 7). In the rule-applications style, the above example derivation is represented by the set

$$\{(S, xyz) \rightarrow (A, x)(B, yz), (A, x) \rightarrow (x, x),$$
$$(B, yz) \rightarrow (C, y)(z, z), (C, y) \rightarrow (y, y)\}.$$

Note that each pair on a right-hand side is either a pair of identical terminals or the left-hand side of another rule application.

The parsing problem

Whereas the recognition problem was to decide whether or not there is at least one derivation

$$S \xrightarrow{*} x_1 ... x_n$$

for a given string $x_1 ... x_n$, the *parsing* problem associated with a grammar is to produce *all* such derivations.

3.3 THE LAMBEK TYPES

This section introduces expressions over grammar symbols that are called Lambek types. Lambek types will be used in chapters 5 and 8. We employ Lambek types to formulate a little theorem that will be of use in chapter 5.

The product operation (3.1) is associative but not commutative. Therefore, it has two inverse operations:

right division $L/M = \forall_{\xi \leftarrow M}(\rho\xi \in L) \rhd \rho$

left division $M\backslash L = \forall_{\xi \leftarrow M}(\xi\rho \in L) \rhd \rho$

The set T_V of all Lambek types over V consists of the empty type ϵ and expressions over the alphabet $V \cup \{/, \backslash\}$. T_V is defined as the smallest set such that

- $\epsilon \in T_V$
- $V \subset T_V$
- if $a \in T_V$ and $b \in T_V$, then ab, a/b, and $b\backslash a$ are in T_V

It follows that $V^* \subset T_V$. We use variables a, b, c to range over T_V. Types are written with parentheses to indicate their structure, if they would otherwise be ambiguous, e.g., $a/(b/c)$ would be ambiguous without parentheses. To reduce the number of parentheses, multiplication is taken to bind stronger than division, e.g., $a/bc \equiv a/(bc)$.

If L is a homomorphism from V^* to languages, then it is extended to types by $L(a\backslash b) = L(a)\backslash L(b)$, $L(a/b) = L(a)/L(b)$, and $L(ab) = L(a)L(b)$. Some propositions involving languages $L(a)$ are logically true for all homomorphisms L. These are called *tautologies*. One has, for example,

$$\begin{aligned} &L(bc) \leftarrow L(a) \Leftrightarrow L(c) \leftarrow L(b\backslash a), \\ &L(bc) \leftarrow L(a) \Leftrightarrow L(b) \leftarrow L(a/c). \end{aligned} \tag{3.9}$$

If a proposition $L(b) \leftarrow L(a)$ is logically true for all L, we write $a \geq b$, which is called a *type tautology*. Examples are

$$\begin{aligned} a &\geq b(b\backslash a), \\ a &\geq (a/b)b. \end{aligned} \tag{3.10}$$

Note the similarity to arithmetic equalities. The tautologies can be verified by substituting the definitions of product and left and right division. For instance, according to the definitions,

$$L(L\backslash M) = \quad \rho \leftarrow L \wedge \forall_{\xi \leftarrow L}(\xi\eta \leftarrow M) \rhd \rho\eta.$$

Now, if $\xi\eta$ in M for all ξ in L, and ρ in L, then $\rho\eta$ in M, which proves the first line of (3.10).

The relation $\geq^G$ can be extended to types: if a and b are types, then $a \geq^G b$ iff one can prove $L_G(b) \leftarrow L_G(a)$ using tautologies, grammar rule constraints, and the cut rule (3.6). To give an idea, if $A \rightarrow XY$, corresponding to the constraint $L_G(XY) \leftarrow L_G(A)$, then, for all $a \in T_V$, $(X\backslash A)/(a\backslash Y) \geq^G a$. The proof is not difficult. If $L_G(XY) \leftarrow L_G(A)$, then the first line of (3.9) implies $L_G(Y) \leftarrow L_G(X\backslash A)$. With the type tautology $Y \geq a(a\backslash Y)$ and transitivity of bunch inclusion, one gets $L_G(a(a\backslash Y)) \leftarrow L_G(X\backslash A)$. With the second line of (3.9) it follows that $L_G(a) \leftarrow L_G((X\backslash A)/(a\backslash Y))$. With the definition of $\geq^G$, this may be written as $(X\backslash A)/(a\backslash Y) \geq^G a$.□

Now comes the little theorem that will be used in chapter 5. Assume a grammar H that shares with G the start symbol and the terminals, i.e., $H = (V'_N, V_T, P', S)$, with $V' = V'_N \cup V_T$. Let there be a mapping $\phi : V' \mapsto T_V$ extended to strings of elements of V' by stipulating $\phi(\alpha\beta) = \phi(\alpha)\phi(\beta)$ and $\phi(\epsilon) = \epsilon$. Let $\phi(S) = S$ and $\phi(x) = x$ if $x \in V_T$. Then:

Theorem If $\phi(A) \geq^G \phi(\alpha)$ for all $A \rightarrow \alpha$ in H, then $L_H(S) \leftarrow L_G(S)$.

That is, if all grammar rules of H map to type tautologies of G, via the mapping ϕ, then the language defined by grammar H is a subbunch of the language defined by G. The proof is simple: each derivation $S \stackrel{*}{\rightarrow} x_1 ... x_n$ in H corresponds via the mapping ϕ, using the cut rule, to a proof of $\phi(S) \geq^G \phi(x_1)...\phi(x_n)$. Because ϕ maps S and terminals to themselves, $S \geq^G x_1 ... x_n$. □

Example: Let G be given by $S \rightarrow xSy$, $S \rightarrow xy$, and H by $S \rightarrow xA$, $A \rightarrow y$, $A \rightarrow Sy$. Both grammars define the same language. This can now be proven

as follows. $L_H(S) \leftarrow L_G(S)$ follows from the theorem by taking $\phi(A) = x\backslash S$, and noting that $S \geq^G x(x\backslash S)$, $x\backslash S \geq^G y$, $x\backslash S \geq^G Sy$. The reverse is trivial: $L_G(S) \leftarrow L_H(S)$ follows because the rules of G correspond to $S \geq^H xSy$ and $S \geq^H xy$, which both hold true. □

The language inclusion problem, i.e., the question whether the language generated by one grammar is a subbunch of the language generated by another, is very difficult. In general, it is even undecidable: no algorithm can exist that determines for any two context-free grammars whether the first defines a language that is included in the language defined by the other [Salomaa,1973]. To solve the problem in particular cases, one has to try to find a relation between the symbols of one grammar and symbols of the other. In the simplest case, there is a 1-1 relationship between grammar symbols, i.e., symbols just have different names. The above theorem enables one to try more complex relations between the symbols of two grammars: the symbols of one grammar are mapped to types over symbols of the other. As we will see, it occurs quite often that a language inclusion problem can be settled in this way.

The above could be generalized by using language operators other than multiplication and division: bunch union, for instance. Then the applicability of the above theorem can be stretched by using new tautologies.

Language multiplication and division are *Galois connected* operations. The general definition of Galois connections is the following. Let $\mathcal{A}$ and $\mathcal{B}$ be two partially ordered sets, the partial-order relations being notated as $\leq_{\mathcal{A}}$, $\leq_{\mathcal{B}}$. Then if, for all $a \in \mathcal{A}$ and $b \in \mathcal{B}$,

$$F(a) \leq_{\mathcal{B}} b \Leftrightarrow a \leq_{\mathcal{A}} G(b), \tag{3.11}$$

for two functions F (from $\mathcal{A}$ to $\mathcal{B}$) and G (from $\mathcal{B}$ to $\mathcal{A}$), then the pair (F, G) is a Galois connection. In our case, $\mathcal{A}$ and $\mathcal{B}$ are identical and equal to the set of all languages, which is partially ordered by "$\leftarrow$". If F is defined as multiplication to the right with M, $F(L) = LM$, and G as right division with same language, $G(L) = L/M$, then (3.11) corresponds to the second equivalence in (3.9). Likewise, the first equivalence of (3.9) reflects that multiplication to the left with a language is Galois connected to left division by the same language. Many type tautologies can be derived from the theory of Galois connections. For instance, it is generally true for the Galois connection of (3.11) that

$$F(G(b)) \leq_{\mathcal{B}} b,$$

which leads to the first type tautology of (3.10), if F is left multiplication with a language and G is left division by the same language.

Another theorem for Galois connections is

$$a \leq_{\mathcal{A}} G(F(a)). \tag{3.12}$$

Its translation to type tautologies is left as an exercise.

The above tautologies can also be seen as theorems of a logic. This becomes manifest if we write $\vdash$ instead of $\leftarrow$, remove all the L's in (3.9), and write $a \wedge b$ instead of ab, and $a \rightarrow b$ instead of $a \backslash b$. Then, for instance,

$$L(bc) \leftarrow L(a) \Leftrightarrow L(c) \leftarrow L(b \backslash a) \text{ reads } b \wedge c \vdash a \Leftrightarrow c \vdash (b \rightarrow a).$$

Similarly, $a \geq b(b \backslash a)$ translates into $b \wedge (b \rightarrow a) \vdash a$. What does not correspond to a tautology is a theorem like

$$a \wedge a \vdash a.$$

The absence of such theorems is typical of *linear logic*. Linear logic has only theorems in which every conjunct in the premise (before the $\vdash$) is used exactly once in a proof of the conclusion (after the $\vdash$).

3.4 RECOGNITION FUNCTIONS

Given some input string ξ of terminal symbols, a grammar determines for each string of grammar symbols α whether or not $\alpha \xrightarrow{*} \xi$. Also, for substrings η of ξ it may be determined whether or not $\alpha \xrightarrow{*} \eta$. Let us define for each α a bunch-valued *recognition function* $[\alpha]$, of type $V_T^* \mapsto V_T^*$, as follows:

$$[\alpha](\xi) = \quad \alpha \xrightarrow{*} \eta \wedge \xi = \eta\rho \rhd \rho. \tag{3.13}$$

Stated differently, this defines a function $[\cdot]$ that operates on two strings of grammar symbols, such that $[\cdot](\alpha, \xi) = [\alpha](\xi)$.

In (3.13), the argument is split into two parts, the first of which is derivable from α. The second part is output by the function. It follows, for all α and β, that

$$\begin{aligned}
[\alpha\beta](\xi) = \;& \alpha\beta \xrightarrow{*} \eta \wedge \xi = \eta\rho \;\rhd\; \rho \\
= \;& \alpha \xrightarrow{*} \eta_1 \wedge \xi = \eta_1\rho_1 \wedge \beta \xrightarrow{*} \eta_2 \wedge \rho_1 = \eta_2\rho \;\rhd\; \rho \\
= \;& \alpha \xrightarrow{*} \eta_1 \wedge \xi = \eta_1\rho_1 \;\rhd\; (\beta \xrightarrow{*} \eta_2 \wedge \rho_1 = \eta_2\rho \;\rhd\; \rho) \\
= \;& \rho_1 \leftarrow [\alpha](\xi) \;\rhd\; [\beta](\rho_1) \\
= \;& [\beta]([\alpha](\xi)).
\end{aligned}$$

Here (2.3) and (2.5) were used in the second and fourth transitions, respectively. Thus, $[\alpha\beta] = [\beta][\alpha]$, where $[\beta][\alpha]$ is the composition of functions $[\beta]$ and $[\alpha]$, defined by

$$([\beta][\alpha])(\xi) = [\beta]([\alpha](\xi)).$$

In other words, $\alpha = X_1 ... X_k$ implies $[\alpha] = [X_k]...[X_1]$ and $[\epsilon](\xi) = \xi$. In algebraic terms, the mapping $[\cdot]$ is a homomorphism from V^* to a function space of bunch-valued functions. Since the functions $[\alpha]$ are compositions of functions $[X]$, an implementation for the latter implies an implementation of the former. Now,

$$\begin{aligned}
[X](\xi) = \;& X \xrightarrow{*} \eta \wedge \xi = \eta\rho \;\rhd\; \rho \\
= \;& ((X \in V_T \wedge X = \eta) \vee (X \rightarrow \alpha \wedge \alpha \xrightarrow{*} \eta)) \wedge \xi = \eta\rho \;\rhd\; \rho \\
= \;& (X \in V_T \wedge \xi = X\rho \;\rhd\; \rho) | \\
& (X \rightarrow \alpha \wedge \alpha \xrightarrow{*} \eta \wedge \xi = \eta\rho \;\rhd\; \rho) \\
= \;& (X \in V_T \wedge \xi = X\rho \;\rhd\; \rho) | \\
& (X \rightarrow \alpha \;\rhd\; (\alpha \xrightarrow{*} \eta \wedge \xi = \eta\rho \;\rhd\; \rho)) \\
= \;& (X \in V_T \wedge \xi = X\rho \;\rhd\; \rho) | \\
& (X \rightarrow \alpha \;\rhd\; [\alpha](\xi)).
\end{aligned}$$

Here (2.4) was used to eliminate the disjunction $\vee$ and (2.3) to eliminate a conjunction $\wedge$. We could have arrived at

$$\begin{aligned}
[X](\xi) = \;& X \in V_T \wedge \xi = X\rho \;\rhd\; \rho \;| \\
& X \rightarrow \alpha \;\rhd\; [\alpha](\xi)
\end{aligned} \tag{3.14}$$

in another way too: Since $\alpha \xrightarrow{*} \eta$ implies $A \xrightarrow{*} \eta$ if there is a rule $A \rightarrow \alpha$, it follows from (3.13) that

$$[\alpha](\xi) \leftarrow [A](\xi). \tag{3.15}$$

The smallest function $[A]$ that satisfies such inequalities is, in accordance with (2.1),

$$[A](\xi) = \quad A \rightarrow \alpha \ \triangleright [\alpha](\xi),$$

whereas it follows directly from (3.13) that

$$[x](\xi) = \quad \xi = x\rho \ \triangleright \rho$$

if $x \in V_T$. To summarize, (3.14) is the smallest function that satisfies (3.15) for all grammar rules $A \rightarrow \alpha$. Note that the status of the inequality $[\alpha](\xi) \leftarrow [A](\xi)$ resembles that of $L_G(\alpha) \leftarrow L_G(A)$.

Note that $\epsilon \leftarrow [S](\xi)$, equivalent to $S \xrightarrow{*} \xi$, means that ξ is a correct sentence. For recognition purposes, the nondeterministic interpretation of $\epsilon \leftarrow [S](\xi)$ is not of much use. It says that ϵ is one of the possible results of a computation of $[S](\xi)$. However, the computation might also produce a finite string if some prefix of ξ is a correct sentence as well. Therefore, we interpret a bunch-valued recognition function as producing all of its results simultaneously, as in section (2.2).

3.5 GENERATION

A grammar may also be used for generating sentences rather than recognizing them, and here the nondeterministic interpretation of bunch-valued functions is very natural. Indeed, it is in general impossible to generate all sentences defined by a grammar, but it is possible to produce any one sentence nondeterministically. The following variant of (3.13) randomly produces a string of terminals produced by α and adds this string to its argument:

$$[\alpha]'(\xi) = \quad \alpha \xrightarrow{*} \eta \ \triangleright \xi\eta. \tag{3.16}$$

With this definition, $[S]'(\epsilon)$ nondeterministically generates a correct sentence, if it terminates. One can prove $[\alpha\beta]' = [\beta]'[\alpha]'$, and again we have a homomorphism from V^* to bunch-valued functions.

3.6 SUMMARY OF INTERPRETATIONS

Context-free grammars are usually presented as one level in the Chomsky hierarchy of rewriting grammars [Chomsky,1959]. This is a very one-sided way to depict context-free grammars, however. The formalism of context-free grammars is like a piece of art that can be placed in any of a great number of art styles but never really fits in. As we already saw, context-free grammars can be looked upon from different viewpoints. Each interpretation of context-free grammars suggests its own generalizations of the formalism.

Above we encountered four ways to interpret grammar rules. Let us recapitulate them here. Generalizations that are natural in the contexts of various interpretations aptly illustrate the differences between the interpretations. If $A \rightarrow \alpha$ is a grammar rule, then

- in the rewriting interpretation, A may be rewritten into α. More generally, $\beta A \gamma$ may be rewritten into $\beta\alpha\gamma$. So a grammar rule is a prescription for deriving strings of grammar symbols. The natural generalization is to allow rules $\gamma \rightarrow \alpha$, with strings of grammar symbols at their left-hand sides. Rewriting a string β with a rule $\gamma \rightarrow \alpha$ means replacing an occurrence of γ in β by α. Grammars with rules $\gamma \rightarrow \alpha$ in general have no natural semantics, in the sense that it is not possible to consistently assign languages to grammar symbols. Although, from this point of view, such grammar formalisms are not attractive, they have been studied extensively in theoretical contexts. In practice, however, they are not very useful. As an illustration, consider the following grammar:

  $$\begin{array}{l} S \rightarrow xSBC \\ S \rightarrow xyC \\ CB \rightarrow BC \\ yB \rightarrow yy \\ yC \rightarrow yz \\ zC \rightarrow zz \end{array}$$

  The language generated by this grammar is $(n \geq 1 \triangleright x^n y^n z^n)$, which is a language that cannot be generated by a context-free grammar. A

derivation in which all rules are used exactly once, in the order in which they are listed above, is

$$\begin{aligned} S &\rightarrow xSBC \\ &\rightarrow xxyCBC \\ &\rightarrow xxyBCC \\ &\rightarrow xxyyCC \\ &\rightarrow xxyyzC \\ &\rightarrow xxyyzz. \end{aligned}$$

Small as it is, this grammar is very difficult to comprehend, as the reader will undoubtedly be able to verify. The reason is that the grammar symbols have no interpretation other than that of bookkeeping devices, so the meaning of grammar rules cannot be understood locally.

- in the natural interpretation, $L_G(\alpha) \leftarrow L_G(A)$. Inspired by the arithmetic analogue, one might write X instead of $L_G(X)$, and insist that $A \rightarrow \alpha$ means $\alpha \leftarrow A$: α is a subbunch of A. Rules $A \rightarrow \alpha_1$, ...,$A \rightarrow \alpha_k$, for the same nonterminal, are often abbreviated as $A \rightarrow \alpha_1|...|\alpha_k$. This is very natural here: it means that $\alpha_1|...|\alpha_k$ is a subbunch of A. Natural generalizations come from allowing other operations on languages in the rules, like intersection. As an illustration, consider the following grammar rules:

$$\begin{aligned} S_1 &\rightarrow xS_1y \\ S_1 &\rightarrow xy \\ S_2 &\rightarrow yS_2z \\ S_2 &\rightarrow yz \end{aligned}$$

Clearly, we have $L_G(S_1) = (n \geq 1 \rhd x^n y^n)$ and $L_G(S_2) = (n \geq 1 \rhd y^n z^n)$. If we add the rules

$$\begin{aligned} S &\rightarrow (S_1C) \cap (AS_2) \\ C &\rightarrow Cz \\ C &\rightarrow z \\ A &\rightarrow Ax \\ A &\rightarrow x \end{aligned}$$

with

$$L_G(\alpha \cap \beta) = \quad \xi \leftarrow L_G(\alpha) \wedge \xi \leftarrow L_G(\beta) \rhd \xi,$$

it follows from $L_G(S_1C) = (n \geq 1 \wedge k \geq 1 \rhd x^n y^n z^k)$ and $L_G(AS_2) = (n \geq 1 \wedge k \geq 1 \rhd x^k y^n z^n)$ that

$$L_G(S) = \quad n \geq 1 \triangleright x^n y^n z^n.$$

So we here have a grammar for the same language as above. Whereas the previous grammar for this language lacks all perspicuity, this one has a clear declarative meaning. Nevertheless, grammars of this kind never made it to practical applications.

- in the recognition function interpretation, $[\alpha](\xi) \leftarrow [A](\xi)$. All grammar rules for A together define the smallest function $[A]$ that is compatible with the restrictions $[\alpha](\xi) \leftarrow [A](\xi)$. Here the natural generalization is to allow the functions to have extra input and output parameters. This leads to attribute grammars, which are very useful in practice, both in compiler technology and in computational linguistics. The subject of attribute grammars is elaborated in chapter 8.
- in the generation function interpretation, $[\alpha]'(\xi) \leftarrow [A]'(\xi)$. This case is a variant of the recognition function interpretation.

This list is not complete. For instance, a grammar rule $A \rightarrow XY$ could be understood as a function that maps two parse trees t_1 and t_2, with roots X and Y, to a parse tree with root A and sons t_1 and t_2. In this view, a terminal is a tree with only one node. Clearly, such a function is reversible: from the resulting tree, the right-hand side trees may be reobtained by removing the root. A natural generalization is obtained if more powerful reversible functions on trees are allowed. Reversibility is required because a grammar must be suitable for analysis as well as generation of sentences. Tree-based generalizations of context-free grammars are being used in computational linguistics. The reason is that the tree concept has deep roots in linguistics. For more on this see [Rosetta].

The picture should be clear: there are many ways to look upon context-free grammars and to generalize them. Such generalizations are not an academic matter: pure context-free grammars have few practical applications. Separate investigations of context-free grammars are nevertheless worthwhile because of the amenability of context-free grammars to mathematical techniques. Being about the kernel of many practical formalisms, the results of such studies are likely to generalize to practical results. In particular, parsers for some extension of context-free grammars often can be extensions to parsers for context-free grammars. In chapter 8 we set out to illustrate this for the case of attribute grammars.

Exercises

1. If $a \geq b$ and $b \geq a$ are type identities, then we write $a \equiv b$, a *type identity*. Prove the type identities

$$(bc)\backslash a \equiv c\backslash(b\backslash a)$$
$$b\backslash(a/c) \equiv (b\backslash a)/c$$

2. Translate (3.12) into at least one type tautology. Prove this tautology.
3. Prove that the extension of $\geq^G$ to Lambek types is conservative, i.e., that the equivalence

$$a \geq^G b \equiv a \xrightarrow{*} b$$

is true whenever a and b are elements of V^*.

4. Prove $[\alpha\beta]' = [\beta]'[\alpha]'$, on the basis of (3.16).
5. Implement (3.16) just like (3.14).

4

RECURSIVE DESCENT

The term *recursive descent* refers to a technique for implementing parsers that directly follows from the (bunch-)functional interpretation of grammars. A recursive descent parser consists of a collection of mutually recursive procedures or functions. Many compilers are implemented with the recursive descent technique, with a so-called look-ahead facility. In this chapter we discuss the technique with and without look ahead and study its properties when applied to arbitrary grammars.

4.1 THE FUNCTIONAL INTERPRETATION

The input and output of recognition functions, defined in (3.13), are strings of terminal symbols. Assume that there is some reference string $x_1 \ldots x_n$ of terminal symbols. Then, instead of working with strings, one can work with numbers. With each number i, $0 \leq i \leq n$, associate the string $x_{i+1} \ldots x_n$. Then the definition (3.13) may be reformulated as

$$[\alpha](i) = \quad \alpha \xrightarrow{*} x_{i+1} \ldots x_j \ \rhd j. \tag{4.1}$$

We will mainly work with numbers from now on, because it is sometimes technically more convenient. The results of the previous chapter carry over to numbers straightforwardly. For instance, the inequalities (3.15) become

$$[\alpha](i) \leftarrow [A](i), \tag{4.2}$$

and the function (3.14) becomes

$$[X](i) = \begin{array}{l} X = x_{i+1} \rhd i+1 \mid \\ X \rightarrow \alpha \rhd [\alpha](i). \end{array} \tag{4.3}$$

Thus, if $A \rightarrow BCD$ is the only grammar rule for A, the following is an implementation of $[A]$:

$$[A](i) = [BCD](i) = [D][C][B](i) = [D]([C]([B](i))). \tag{4.4}$$

It may be useful to explicitly state the algorithmic interpretation of this function, without making use of the fact that functions distribute over bunches. Equation (4.4) may be rewritten, using law (2.5) twice, as

$$[A](i) = \quad k \leftarrow [B](i) \rhd (m \leftarrow [C](k) \rhd [D](m)).$$

The prescription in section 2.2 entails that this is to be computed as

```
[A](i) :  result:=null;
          for all k←[B](i) do
             for all m←[C](k) do
                result:= result | [D](m)
             od
          od;
          return result;
```

Note how the nesting of $\rhd$ translates to an isomorphic nesting of "for all."

The mapping from grammar symbols to functions is the basis of the functional parsing algorithms discussed in this book. With such functions, the correctness of a sentence can be determined according to

$$S \xrightarrow{*} x_1 \ldots x_n \equiv n \leftarrow [S](0).$$

4.2 TERMINATION

The recursive definition of the recognition functions $[X]$ can be seen as a functional algorithm, a *recognizer*, if it terminates. Termination is a problem if the grammar is left recursive, i.e., if it has a derivation $A \xrightarrow{+} A\alpha$. For such grammars, function $[A](i)$ will call itself with the same argument. A simple example of this is grammar $S \rightarrow Sx$, $S \rightarrow \epsilon$. According to above prescriptions, its recognizer would be

$$\begin{aligned} [S](i) = &\quad [x]([S](i)) \mid i \\ [x](i) = &\quad x = x_{i+1} \rhd i+1 \end{aligned}$$

and this is a cyclic definition: $[S](i)$ is defined in terms of itself.

4.3 COMPLEXITY AND MEMOIZATION

The definition of $[X]$ in (4.3) is nondeterministic, in general: the right-hand side of (4.3) denotes a bunch that will have more than one value. The time resources needed to solve a recognition problem with (4.3) is an exponential function of the length n of the string that is to be recognized - that is, if the algorithm is applied with arbitrary grammars that are not left recursive. There is a way to get around this exponentiality. The idea is simple. The algorithm is exponential although there is a fixed number of functions, which can be invoked with only $O(n)$ possible values of their arguments. This means that the average function is recomputed an exponential number of times on the same arguments. Now suppose we change the implementation of the functions in such a way that they store previously computed argument-result pairs. With such knowledge about previous invocations, a function needs to compute results only when called with parameter values that did not occur before. Otherwise, a previously computed result can be returned without recomputing it. Functions implemented in this way are called *memo-functions*, and the technique *memoization*.

There is a class of grammars, encompassing most grammars for programming languages, that can be parsed with a memoized version of (4.3) in linear time. This is the case if the grammar is such that each function $[X]$ produces a bunch with at most one element. The calculation of the amount of time needed in worst-case situations is simple. There are $O(n)$ different function

invocations. If l_A is the total number of symbols occurring on the right-hand sides of grammar rules for A, i.e., $l_A = \sum_{A \to \alpha} |\alpha|$, then in function $[A]$ at most l_A functions $[X_j]$ are called. This means that each invocation of function $[A]$ takes an amount of time that does not depend on n. Therefore, the overall time complexity is linear. Also the space resources are linear, since a linear number of different function invocations occur, and each one leads to the storage of the result, which by assumption is a bunch with at most one integer.

In the general case, the complexity is worse. Let $A \to X_1 ... X_m$ be the rule with the longest right-hand side. Then in $[A](i)$ an expression

$$\begin{aligned} & i_1 \leftarrow [X_1](i) \rhd (i_2 \leftarrow [X_2](i_1) \rhd \\ & \quad ... \rhd (i_{m-1} \leftarrow [X_{m-1}](i_{m-2}) \rhd [X_m](i_{m-1}))...) \end{aligned}$$

occurs, without making use of bunch distributivity. There are $O(n^{m-1})$ sequences $i \leq i_1 ... \leq i_{m-1}$ with fixed i. Each sequence $i \leq i_1 ... \leq i_{m-1}$ may lead to a bunch of $O(n)$ size. Merging all these bunches may be done in $O(n^m)$ time on a random access machine. Because there are $O(n)$ different invocations of this most complex function, the overall time complexity of the recognizer is $O(n^{m+1})$. The space complexity is quadratic: $O(n)$ different invocations each store a bunch of size $O(n)$. A more refined analysis shows that the complexity does not so much depend on the maximum length of grammar rules, but rather on the maximum number of nonterminals on the right-hand side of grammar rules.

It is possible to rewrite a given grammar such that the right-hand sides in the rewritten grammar have at most two grammar symbols. We call such grammars *bilinear*. If a grammar is bilinear, (4.3) may be written as

$$\begin{aligned} [x](i) = \quad & x = x_{i+1} \rhd i + 1 \\ [A](i) = \quad & A \to \epsilon \rhd i \mid \\ & A \to X \rhd [X](i) \mid \\ & A \to XY \rhd [Y]([X](i)). \end{aligned} \tag{4.5}$$

The functions $[x]$ can be eliminated. In chapter 5 we will encounter a bilinear grammar in which terminals appear only as the first of two symbols on the right-hand sides of grammar rules. A recognizer for such a grammar is

$$
\begin{aligned}
[A](i) = \quad & A \rightarrow \epsilon \ \triangleright \ i \ | \\
& A \rightarrow B \ \triangleright \ [B](i) \ | \\
& A \rightarrow xB \wedge x = x_{i+1} \ \triangleright \ [B](i+1) \ | \\
& A \rightarrow BC \ \triangleright \ [C]([B](i)).
\end{aligned}
\tag{4.6}
$$

For bilinear grammars that are not left recursive, the time complexity of the last two algorithms is cubic. This fact is a motivation for studying grammar transformations, in the next chapter, to make grammars bilinear and not left-recursive.

The algorithms in this chapter can be understood and implemented in two ways. Firstly, one may adopt an *interpretative* view. In this view, there is only one function $[\cdot]$ with two arguments, a grammar symbol and an integer. In the *compiled* view, there are as many (compiled) functions $[X]$ as there are grammar symbols (or nonterminals), each with one integer argument. The reason for using a circumfix operator $[\cdot]$ to map nonterminals to functions is to enhance the compiled view in the reader's mind. The notation $[X](i)$ is of course mathematically equivalent to $[\cdot](X, i)$, but it emphasizes the compiled view, displaying X as part of the function name rather than as argument of a generic function. In the language of functional programming, the compiled functions are the result of partial evaluations of the function $[\cdot]$, one for each value of its grammar symbol argument. The interpretative implementations are small but slow, and hence useful in prototypes. The compiled implementations are larger but faster. The reason is that in interpretative implementations a lot of computations are unnecessarily repeated during the recognition of each string. In the compiled parser, all computation that depends only on the structure of the grammar has been done while creating and optimizing the functions, and needs not be repeated for each string. For this reason, whenever time efficiency is an issue - and it often is - compiled implementations are the best.

4.3.1 Example

It may be helpful to see one more concrete example. Consider the grammar $S \rightarrow xS$, $S \rightarrow y$. A recognizer for this grammar is

$$
\begin{aligned}
[S](i) = \quad & x_{i+1} = x \ \triangleright \ [S](i+1) \ | \\
& x_{i+1} = y \ \triangleright \ i+1.
\end{aligned}
$$

There are no free variables in this function, subject to implicit **let** quantification. Therefore, its algorithmic interpretation does not involve "for all" loops. Moreover, the two guards are mutually exclusive:

```
[S](i) =  result:=null;
          if x_{i+1} = x then result:=[S](i + 1)
          else if x_{i+1} = y then result:=i + 1
          fi;
          return result;
```

So our recognition functions can be implemented directly, not only in a functional programming language, but with only a little more difficulty also in imperative programming languages that allow one to define functions recursively.

4.3.2 A way to get cubic complexity

Above we established that if $[X_1 ... X_k]$ is viewed as a shorthand for $[X_k]...[X_1]$, the complexity of the recursive descent recognizer grows with the length of grammar rules. However, one may also decide to create separate functions $[\alpha]$ for strings, one for each postfix of a rule's right-hand side, and express them in terms of each other according to $[X_1 ... X_k] = [X_2 ... X_k][X_1]$. This may be seen as an implicit bilinearization of the grammar, and it makes the recognizer cubic, if it terminates and if the functions $[\alpha]$ are memoized. A complete definition of recursive descent recognition, with separate functions for strings with length unequal to one, is

$$
\begin{aligned}
[x](i) &= \quad x = x_{i+1} \rhd i + 1 \\
[A](i) &= \quad A \rightarrow \alpha \rhd [\alpha](i) \\
[XY\beta](i) &= \quad [Y\beta]([X](i)) \\
[\epsilon](i) &= \quad i
\end{aligned}
$$

4.4 LOOK AHEAD

The function

$$[A](i) = \quad A \rightarrow \alpha \rhd [\alpha](i)$$

of (4.3) is nondeterministic, because in general there will be more than one α such that $A \rightarrow \alpha$. More than one $[\alpha](i)$ may produce a nonempty result and, being the merger of these results, $[A](i)$ will be truly multiple-valued, in general. For some grammars, no function $[A](i)$ can ever produce a bunch with more than one element, i.e., for each possible continuation of the input after position i at most one grammar rule for A is applicable. Even in such a case, the above function $[A]$ invokes $[\alpha]$ for every $A \rightarrow \alpha$. This means that a lot of invocations are wasted, and it would be desirable to know beforehand which ones are doomed to produce *null*. This can be achieved with look-ahead techniques. Below, we describe the most common version of look ahead, in which the choice of calling a function $[\alpha](i)$ is made dependent on the value of x_{i+1}.

Let us construct for each α the set $first(\alpha)$ of terminals it can begin with:

$$first(\alpha) = \{x | x \in V_T \wedge \alpha \xrightarrow{*} x\beta\}. \tag{4.7}$$

(Note that we left out existential quantification over β. We take the convention that free variables in set constructions are always bound by implicit existential quantifiers.) Then it follows that

$$[A](i) = \quad A \rightarrow \alpha \wedge (x_{i+1} \in first(\alpha) \vee \alpha \xrightarrow{*} \epsilon) \ \triangleright \ [\alpha](i). \tag{4.8}$$

Suppose the grammar has been constructed in such a way that for all A, x there is at most one α such that $A \rightarrow \alpha$ and $x \in first(\alpha)$ and that $\alpha \xrightarrow{*} \epsilon$ for no right-hand-side α. Then in (4.8) a unique value of α is always determined, as desired.

For example, in the grammar $S \rightarrow A$, $A \rightarrow yx$, $S \rightarrow By$, $B \rightarrow xS$, the situation is that $first(A) = \{y\}$ and $first(By) = \{x\}$. Thus, $[S]$ may be written as

$$\begin{aligned} [S](i) = \quad & x_{i+1} = y \ \triangleright \ [A](i) \ | \\ & x_{i+1} = x \ \triangleright \ [y]([B](i)), \end{aligned}$$

which, together with the deterministic functions $[A]$ and $[B]$, forms a deterministic recognizer: no function can ever produce more than one result, and no invoked function produces *null* unless the sentence is ill formed.

If a nonterminal A rewrites directly to α, and $\alpha \xrightarrow{*} \epsilon$, and A has another rule as well, then (4.8) is not deterministic. To make the choice of a rule $A \rightarrow \alpha$ with $\alpha \xrightarrow{*} \epsilon$ conditional, we need the set

$$\begin{aligned} follow(X) = \{x | x \in first(\beta) \wedge A \rightarrow \alpha X \beta\} \cup \\ \{x | x \in follow(A) \wedge A \rightarrow \alpha X \beta \wedge \beta \xrightarrow{*} \epsilon\}. \end{aligned} \tag{4.9}$$

This definition is recursive and does not define $follow$ uniquely. We intend to use the smallest possible solution. It may be obtained by approximating it from below with a series $follow_0$, $follow_1$, $follow_2$, etc., by iterating

$$\begin{aligned} follow_{i+1}(X) = \{x | x \in first(\beta) \wedge A \rightarrow \alpha X \beta\} \cup \\ \{x | x \in follow_i(A) \wedge A \rightarrow \alpha X \beta \wedge \beta \xrightarrow{*} \epsilon\}, \end{aligned}$$

starting with $follow_0(X) = \emptyset$, for all X. Because there are only finitely many terminals, it must be that $follow_{i+1} = follow_i$, for some i: the series converges to the minimal solution of (4.9).

It is convenient to add to each grammar the rule $S' \rightarrow S \perp$, where $S', \perp$ are new symbols that appear only in this rule. S' is the new start symbol and $\perp$ is formally added to V_T. Of course, any correct input must now end with $\perp$. The above then implies that $\perp \in follow(S)$, and it is guaranteed that $follow(X)$ is not empty if $\exists_{\alpha\beta}(S \xrightarrow{*} \alpha X \beta)$. If X is one of the added symbols $S', \perp$ then $follow(X) = \emptyset$.

The set $follow(A)$ consists of all followers of A: the terminals that may follow after recognition of A. With followers sets one may modify (4.8), if $A \xrightarrow{*} \epsilon$, to become potentially deterministic:

$$\begin{aligned} &[A](i) = \\ &\quad A \rightarrow \alpha \wedge (x_{i+1} \in first(\alpha) \vee (\alpha \xrightarrow{*} \epsilon \wedge x_{i+1} \in follow(A))) \rhd [\alpha](i). \end{aligned}$$

Only if $\exists_{B\alpha}(A \xrightarrow{*} \alpha B \wedge B \xrightarrow{*} \epsilon)$ function $[A]$ now no longer satisfies specification (4.1), so $[S']$ still does. If for all $A \neq S'$ and every x_{i+1} at most one α exists that makes the guard true, the choice of grammar rule is always unique. Assuming that it is not left recursive, the grammar is then called LL(1). The last example grammar is an LL(1) grammar. For LL(1) grammars each invocation $[X](i)$ occurs at most once. It follows that memoization is superfluous

and that each function delivers at most one value. To improve efficiency, function argument i may be replaced by a global variable; the recognition functions then become procedures that act on this global variable, e.g., the function of the last example becomes the procedure

$$
\begin{aligned}
[S] = \ & \text{if } x_{i+1} = y \text{ then } [A] \\
& \text{else if } x_{i+1} = x \text{ then } [B];\ [y] \\
& \text{else } \textit{error} \\
& \text{fi}
\end{aligned}
$$

where procedure *error* is called if the function would have produced *null*. The procedure for B reads

$$[B] = \ \text{if } x_{i+1} = x \text{ then } i := i + 1;\ [S] \text{ else } \textit{error} \text{ fi}.$$

It is conventional terminology to use a variable *sym* for x_{i+1} and to increment the counter i by a procedure *nextsym*. Then procedure $[B]$ becomes

$$[B] = \ \text{if } \textit{sym}{=}x \text{ then } \textit{nextsym};\ [S] \text{ else } \textit{error} \text{ fi}.$$

Deterministic LL(1) recognizers run in linear time. Their space consumption is given by the maximal size of the recursion stack, which is also proportional to the input length.

The above can be generalized in a straightforward way by looking ahead more than one symbol. Grammars that can be parsed deterministically with a k-symbol look ahead are called LL(k) grammars. In compiler practice, parsers with $k \neq 1$ are rarely used. In natural language parsers, even $k \neq 0$ is never used: natural language grammars are very ambiguous and the ambiguities cannot be resolved by looking ahead a number of words.

A practical advantage of recursive descent programs, which was already mentioned in section 3.6, is the possibility of adding parameters to the functions. Formally, the resulting programs are recognizers for *attribute grammars*, which are context-free grammars with grammar rules that are parametrized with attributes. The relations between attributes are stipulated per grammar 'le. Attribute grammars form an important subject of parsing theory, which ' be touched upon in chapter 8.

4.5 ERROR RECOVERY

If a user-friendly compiler encounters an error, it tries to resume parsing to find other possible errors in the same compilation run. This is called error recovery. Below we present a form of error recovery that is based on the definition of a set *landmarks* of terminal symbols that start some syntactic construction. After an error has occurred, input is skipped until a terminal in *landmarks* is found. At that point an attempt is made to resume parsing. We require that $\perp \in landmarks$.

The following is a deterministic LL(1) parser with error recovery. The new parameter F is a set of landmark terminals at which parsing can be resumed after an error.

$[X](F,i) =$
  if $X = x_{i+1}$ then $i+1$
  else if there is a rule $X \rightarrow \alpha$ such that
    $x_{i+1} \in first(\alpha) \vee (\alpha \xrightarrow{*} \epsilon \wedge x_{i+1} \in follow(X))$ then $[\alpha](F,i)$
  else $error(X,F,i)$
  fi

and

$[\epsilon](F,i) = \quad i$
$[XY\beta](F,i) = \quad [Y\beta](F,[X](F \cup F_{Y\beta},i))$

Here F_α is the following set of landmark terminals:

$$F_\alpha = \{x | x \in landmarks \wedge \alpha = \gamma Z \delta \wedge x \in first(Z)\}.$$

Hence, $F_{Y\beta}$ contains terminals that can be handled by some Z in $Y\beta$, should $[X]$ encounter an error.

The function *error* gives an error message and deletes some input:

error(X,F,i)=
  output('error at "i","X" expected');
  return *delete*(F,i)

The function *delete* is given by

$$delete(F, i) = \text{if } x_{i+1} \in F \text{ then } i \text{ else } delete(F, i+1).$$

It skips terminals until one is found that is in F.

The recognizer should be invoked as $[S](\{\perp\}, 0)$. The above is not the best possible error recovery method. On the other hand, where it comes to error recovery one necessarily has to settle for less than perfect performance. Either wrong error messages will be generated, when parsing is resumed in a too naive way, or too much text will be skipped before parsing is resumed. With a judiciously chosen set *landmarks*, the above will work satisfactorily for many applications.

Exercises

1. Implement the above recognition algorithm for $S \rightarrow xS$, $S \rightarrow y$ in Pascal or C.

5

GRAMMAR TRANSFORMATIONS

The recursive descent algorithm terminates only if the grammar is not left recursive and has cubic time-complexity if the grammar is bilinear. Suppose we could transform a given grammar, transforming away left recursion and grammar rules $A \rightarrow \alpha$ with $|\alpha| > 2$, without affecting the language generated by the grammar. Then a general recognizer could be constructed as the composition of such a grammar transformation and a recursive descent recognizer. Transformations with the desired properties exist. Below we find one, assuming that G does not have ϵ-rules and that it is not cyclic, which means that $A \stackrel{+}{\rightarrow} A$ is impossible for all A.

5.1 MAKING GRAMMARS BILINEAR

First we describe a simple method to make a grammar bilinear. Let $A \rightarrow X_1 ... X_n$ be a grammar rule. The (possibly long) right-hand side may be rewritten with the introduction of some extra nonterminals. These nonterminals have to be fresh, and it is convenient to have a naming convention to guarantee this. For this reason the transformation employs so-called *items*, which are grammar rules with a dot somewhere in the right-hand side. For each rule $A \rightarrow X_1 ... X_n$, with n symbols at the right-hand side, there are $n+1$ items $A \rightarrow X_1 ... X_k \cdot X_{k+1} ... X_n$ (k taking the values $0...n$). The single item for a rule $A \rightarrow \epsilon$ is written as $A \rightarrow \cdot$, or sometimes as $A \rightarrow \epsilon \cdot$ or $A \rightarrow \cdot \epsilon$. Items are standard notation in parsing theory to denote partially recognized grammar rules. Just as anything between square brackets is a function in this book, we adopt the convention that anything between angular brackets is a nonterminal. So, to emphasize that an item is to be viewed as a nonterminal,

we write $<A \rightarrow X_1 ... X_k \cdot X_{k+1} ... X_n>$.

The grammar rule $A \rightarrow X_1 ... X_n$ can be rewritten as follows:

$$\begin{array}{l} A \rightarrow <A \rightarrow \cdot X_1 ... X_n> \\ <A \rightarrow \cdot X_1 ... X_n> \rightarrow X_1 <A \rightarrow X_1 \cdot X_2 ... X_n> \\ <A \rightarrow X_1 \cdot X_2 ... X_n> \rightarrow X_2 <A \rightarrow X_1 X_2 \cdot X_3 ... X_n> \\ \qquad\qquad\qquad\qquad \vdots \\ <A \rightarrow X_1 ... X_{n-1} \cdot X_n> \rightarrow X_n <A \rightarrow X_1 ... X_n \cdot> \\ <A \rightarrow X_1 ... X_n \cdot> \rightarrow \epsilon. \end{array} \tag{5.1}$$

It will be clear that the ensuing grammar is equivalent to the original, in the sense that each application of a grammar rule in a derivation in the original grammar corresponds to a simple subderivation in the transformed grammar. We will refer to the transformed grammar as E_G, if the original grammar is G.

Characterization of E_G

The grammar E_G can be specified succinctly. It consists of the following rules:

$$A \rightarrow <A \rightarrow \cdot \alpha>, \text{ for all } A \rightarrow \alpha \text{ in } G \tag{5.2}$$

$$<A \rightarrow \alpha \cdot X\beta> \rightarrow X <A \rightarrow \alpha X \cdot \beta>, \text{ for all } A \rightarrow \alpha X \beta \text{ in } G \tag{5.3}$$

$$<A \rightarrow \alpha \cdot> \rightarrow \epsilon, \text{ for all } A \rightarrow \alpha \text{ in } G \tag{5.4}$$

Note that a grammar rule $A \rightarrow \epsilon$ of G appears in E_G as a pair $A \rightarrow <A \rightarrow \cdot>$, $<A \rightarrow \cdot> \rightarrow \epsilon$.
For example, let G be given by $S \rightarrow xy$, $S \rightarrow xSy$. Then E_G is

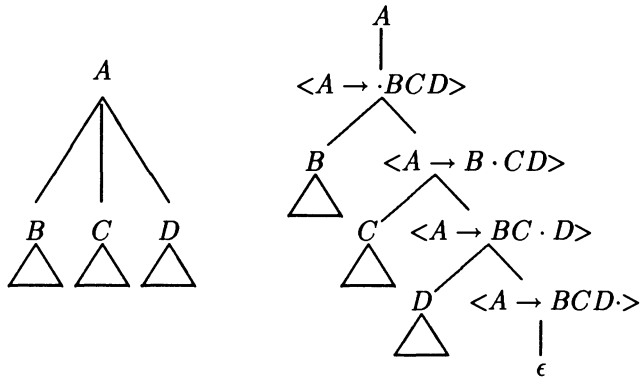

Figure 5.1 Corresponding derivations in G and E_G

$S \rightarrow <S \rightarrow \cdot xy>$
$<S \rightarrow x \cdot y> \rightarrow y <S \rightarrow xy\cdot>$
$S \rightarrow <S \rightarrow \cdot xSy>$
$<S \rightarrow x \cdot Sy> \rightarrow S <S \rightarrow xS \cdot y>$
$<S \rightarrow xSy\cdot> \rightarrow \epsilon$

$<S \rightarrow \cdot xy> \rightarrow x <S \rightarrow x \cdot y>$
$<S \rightarrow xy\cdot> \rightarrow \epsilon$
$<S \rightarrow \cdot xSy> \rightarrow x <S \rightarrow x \cdot Sy>$
$<S \rightarrow xS \cdot y> \rightarrow y <S \rightarrow xSy\cdot>$

□

Derivations in G correspond to derivations in E_G as displayed in figure 5.1. The triangles are arbitrary subtrees.

Equivalence of G and E_G

The little theorem of section 3.3 here has its first application, providing an alternative way to prove the equivalence of G and E_G. To prove $L_{E_G}(S) \leftarrow L_G(S)$ we must find a mapping ϕ from the nonterminals of E_G to types over V, the grammar symbols of G, which are also among the grammar symbols of E_G. We take $\phi(X) = X$ for $X \in V$, whereas the application of ϕ to item nonterminals is defined by

$$\phi(<A \rightarrow \alpha \cdot \beta>) = \alpha \backslash A. \tag{5.5}$$

Then the rules (5.2) map to $A \geq^G A$, which is a type tautology. Likewise, the rules (5.3) map to $\alpha \backslash A \geq^G X(\alpha X \backslash A)$, which is also a type tautology. Lastly, the rules (5.4) map to $\alpha \backslash A \geq^G \epsilon$, which is true because $A \geq^G \alpha$ if

$A \rightarrow \alpha$. Conversely, to prove $L_G(S) \leftarrow L_{E_G}(S)$, let $\phi(X) = X$. Then, each rule $A \rightarrow \alpha$ of G maps to $A \geq^{E_G} \alpha$, which holds true because each grammar rule of G corresponds to a derivation in E_G, as displayed in (5.1). It thus follows that the language defined by E_G is equal to the language defined by G.

5.2 RECURSIVE DESCENT FOR E_G

The grammar E_G is such that the recursive descent algorithm (4.5) is applicable, and leads to a collection of functions $[x]$, with $x \in V_T$, $[A]$ with $A \in V_N$, and $[<A \rightarrow \alpha \cdot X\beta>]$, which we write without angular brackets as $[A \rightarrow \alpha \cdot X\beta]$:

$$\begin{aligned}
&[x](i) = \quad x = x_{i+1} \rhd i+1,\\
&[A](i) = \quad A \rightarrow \alpha \rhd [A \rightarrow \cdot\alpha](i),\\
&[A \rightarrow \alpha \cdot X\beta](i) = \quad [A \rightarrow \alpha X \cdot \beta]([X](i)),\\
&[A \rightarrow \alpha\cdot](i) = \quad i.
\end{aligned}$$

The functions $[x]$ and $[A]$ can be eliminated by substituting their recursive definitions, where they are called

$$\begin{aligned}
&[A \rightarrow \alpha \cdot x\beta](i) = \quad x = x_{i+1} \rhd [A \rightarrow \alpha x \cdot \beta](i+1),\\
&[A \rightarrow \alpha \cdot B\beta](i) = \quad B \rightarrow \gamma \rhd [A \rightarrow \alpha B \cdot \beta]([B \rightarrow \cdot\gamma](i)),\\
&[A \rightarrow \alpha\cdot](i) = \quad i.
\end{aligned} \tag{5.6}$$

The correctness of this algorithm follows from the truthfulness of the grammar transformation and the correctness of the recursive descent method. There is a much shorter way to verify (5.6), however, by proving that its functions are implementations of the specification

$$[A \rightarrow \alpha \cdot \beta](i) = \quad \beta \stackrel{*}{\rightarrow} x_i \ldots x_j \rhd j. \tag{5.7}$$

Assuming that the functions that appear on the right-hand sides of (5.6) satisfy this specification, one can simply calculate by substitution that the left-hand sides satisfy it as well. Let us verify this for each type of function. From (5.7) it follows that

$$\begin{aligned}[A \to \alpha \cdot x\beta](i) &= x\beta \xrightarrow{*} x_{i+1}...x_j \ \rhd j \\ &= x = x_{i+1} \wedge \beta \xrightarrow{*} x_{i+2}...x_j \ \rhd j.\end{aligned}$$

By assumption, $j \leftarrow [A \to \alpha x \cdot \beta](i+1)$ is equivalent to $\beta \xrightarrow{*} x_{i+2}...x_j$. Substituting this and applying law (2.5), the first line of (5.6) appears.
From (5.7) it follows that

$$\begin{aligned}[A \to \alpha \cdot B\beta](i) &= B\beta \xrightarrow{*} x_{i+1}...x_j \ \rhd j \\ &= B \xrightarrow{*} x_{i+1}...x_k \wedge \beta \xrightarrow{*} x_{k+1}...x_j \ \rhd j \\ &= B \to \gamma \wedge \gamma \xrightarrow{*} x_{i+1}...x_k \wedge \beta \xrightarrow{*} x_{k+1}...x_j \ \rhd j.\end{aligned}$$

Substituting $k \leftarrow [B \to \cdot\gamma](i)$ for its assumed equivalent $\gamma \xrightarrow{*} x_{i+1}...x_k$ and likewise $j \leftarrow [A \to \alpha B \cdot \beta](k)$ for $\beta \xrightarrow{*} x_{k+1}...x_j$, and subsequently using (2.5) twice, the second line of (5.6) follows.
From (5.7) it follows that

$$[A \to \alpha\cdot](i) = (\epsilon \xrightarrow{*} x_{i+1}...x_j \ \rhd j) = i,$$

as $\epsilon \xrightarrow{*} x_{i+1}...x_j \equiv (i = j)$. This proves the third line of (5.6).

If one adds a new start symbol S' to grammar G, with $S' \to S$ as its only rule, it follows from (5.7) that

$$S \xrightarrow{*} x_1...x_n \equiv n \leftarrow [S' \to .S](0),$$

so that the recognition problem can be solved again. As before, left recursion leads to termination problems, as the transformation $G \mapsto E_G$ maps a left-recursive grammar to a left-recursive grammar.

5.3 PARTIAL ELIMINATION OF LEFT RECURSION

The principle used to eliminate left recursion is best explained by an example. Consider the left-recursive grammar segment $A \to AY$, $A \to x$. Clearly,

using these rules, the terminal-symbol strings generated by A must start with x, which we can express by assuming as a rule for A

$$A \rightarrow x <x, A>, \tag{5.8}$$

where $<x, A>$ is a new nonterminal that is to derive all strings that A derives, with the initial x cut off. We use the notation $<X, A>$ to denote a nonterminal that derives the language[1]

$$\xi \in V_T^* \wedge A \xrightarrow{*} X\xi \rhd \xi,$$

"$\xrightarrow{*}$" referring to the original grammar. In other words, if one concatenates members of the languages derived by X and $<X, A>$ one gets a member of the language derived by A. What are the rules for $<x, A>$? Clearly, we need

$$<x, A> \rightarrow \epsilon \tag{5.9}$$

to derive $A \xrightarrow{+} x$. In addition, one has the rule

$$<x, A> \rightarrow <A, A> \tag{5.10}$$

as $(A \xrightarrow{*} A\xi \rhd \xi) \leftarrow (A \xrightarrow{*} x\xi \rhd \xi)$ because $A \xrightarrow{*} A\xi$ implies $A \xrightarrow{*} x\xi$. For $<A, A>$ one has

$$<A, A> \rightarrow Y \tag{5.11}$$

as $(Y \xrightarrow{*} \xi \rhd \xi) \leftarrow (A \xrightarrow{*} A\xi \rhd \xi)$ because $Y \xrightarrow{*} \xi$ implies $A \xrightarrow{*} A\xi$. Lastly, one has

$$<A, A> \rightarrow Y <A, A>, \tag{5.12}$$

the "proof" of which is left as an exercise. The rules (5.8)-(5.12) constitute a right-recursive grammar segment that may replace the original segment.

[1]The intended interpretation of $<X, A>$ is that it is a sub-bunch of $X \backslash A$; see exercise 5.

The above technique can be employed to transform a possibly left-recursive grammar E_G into a possibly right-recursive one, which is called F_G. The nonterminals of F_G are the start symbol S, the items $<A \rightarrow \alpha \cdot \beta>$ of E_G, and a new set of nonterminals indexed by an element of V and an item. A typical element of this new set is $<X, A \rightarrow \alpha \cdot \beta>$. The grammar F_G is obtained by rewriting the grammar E_G in two steps.

Construction of F_G

In the first step we change E_G by replacing every rule of the form (5.3) by the pair

$$<A \rightarrow \alpha \cdot X\beta> \rightarrow X <X, A \rightarrow \alpha \cdot X\beta>,$$

$$<X, A \rightarrow \alpha \cdot X\beta> \rightarrow <A \rightarrow \alpha X \cdot \beta>, \tag{5.13}$$

which results in a grammar E'_G that is clearly equivalent to E_G. Next we put all rules of E'_G in a big container $\mathcal{C}$.

The second step is a rewrite of the rules in $\mathcal{C}$ that have the form

$$<A \rightarrow \alpha \cdot \beta> \rightarrow B <B, A \rightarrow \alpha \cdot \beta>, \tag{5.14}$$

with $B \in V_N$. First we substitute for B the right-hand sides of its grammar rules in E_G. Hence, we remove a rule (5.14) from $\mathcal{C}$ and construct from it a set of rules that might replace it:

$$\mathcal{D} = \{<A \rightarrow \alpha \cdot \beta> \rightarrow <B \rightarrow \cdot\gamma><B, A \rightarrow \alpha \cdot \beta> \mid B \rightarrow \gamma \text{ in } G\}.$$

Each rule $<A \rightarrow \alpha \cdot \beta> \rightarrow <B \rightarrow \cdot\gamma><B, A \rightarrow \alpha \cdot \beta>$ in $\mathcal{D}$ is rewritten in its turn and the result is to be added to $\mathcal{C}$. The rewriting depends on the value of γ. There are two cases, $\gamma = \epsilon$ and $\gamma = X\delta$:

$\gamma = \epsilon$: One has $<B \rightarrow \cdot\gamma> \rightarrow \epsilon$ in E_G, which can be substituted in the rule $<A \rightarrow \alpha \cdot \beta> \rightarrow <B \rightarrow \cdot\gamma><B, A \rightarrow \alpha \cdot \beta>$ to yield

$$<A \rightarrow \alpha \cdot \beta> \rightarrow <B, A \rightarrow \alpha \cdot \beta> . \tag{5.15}$$

So we add (5.15) to the container $\mathcal{C}$ (if it was not there yet).

$\gamma = X\delta$: one has in E_G the rule $<B \rightarrow \cdot X\delta> \rightarrow X <B \rightarrow X \cdot \delta>$, which can also be substituted in $<A \rightarrow \alpha \cdot \beta> \rightarrow <B \rightarrow \cdot\gamma><B, A \rightarrow \alpha \cdot \beta>$ to yield

$$<A \rightarrow \alpha \cdot \beta> \rightarrow X <B \rightarrow X \cdot \delta><B, A \rightarrow \alpha \cdot \beta>,$$

which in its turn is equivalent to a pair of bilinear rules:

$$<A \rightarrow \alpha \cdot \beta> \rightarrow X <X, A \rightarrow \alpha \cdot \beta> \tag{5.16}$$

and

$$<X, A \rightarrow \alpha \cdot \beta> \rightarrow <B \rightarrow X \cdot \delta><B, A \rightarrow \alpha \cdot \beta> . \tag{5.17}$$

This pair of rules is added to $\mathcal{C}$. That is, (5.16) is not added if it already exists in $\mathcal{C}$ or if it did exist and was already rewritten (which is possible because (5.16) is of the form (5.14) if $X \in V_N$).

This process continues until no rules of the form (5.14) are left in $\mathcal{C}$. The result is a grammar from which all rules for symbols $B \in V_N$ can be discarded, except for the rules for S. What remains is the grammar F_G, which consists of rules of the form (5.13), (5.15), (5.16) with $X \in V_T$, and (5.17).

Characterization of F_G

To characterize F_G compactly, we need the relation $\Rightarrow$, the *leftmost-symbol* rewriting, defined as

$$\alpha \Rightarrow \beta \equiv \exists_{B\gamma\delta}(\alpha = B\gamma \wedge \beta = \delta\gamma \wedge B \rightarrow \delta \wedge \delta \neq \epsilon), \tag{5.18}$$

and its reflexive and transitive closure $\overset{*}{\Rightarrow}$. If $\alpha \overset{*}{\Rightarrow} A\beta$, then A is called a *left corner* of α.

With (5.18), the above construction results in a grammar that can be described as containing the following rules:

$$S \rightarrow <S \rightarrow \cdot\alpha>, \text{ for all } S \rightarrow \alpha \text{ in } G \tag{5.19}$$

$$<X, A \to \alpha \cdot X\beta> \to <A \to \alpha X \cdot \beta>, \text{ for all } A \to \alpha X\beta \text{ in } G \tag{5.20}$$

$$\begin{array}{l} <A \to \alpha \cdot \beta> \to x <x, A \to \alpha \cdot \beta>, \\ \text{for all } x \text{ and } A \to \alpha\beta \text{ in } G \text{ such that } \exists_\gamma(\beta \overset{*}{\Rightarrow} x\gamma) \end{array} \tag{5.21}$$

$$\begin{array}{l} <A \to \alpha \cdot \beta> \to <B, A \to \alpha \cdot \beta>, \\ \text{for all } B \text{ and } A \to \alpha\beta \text{ in } G \text{ such that } \exists_\gamma(\beta \overset{*}{\Rightarrow} B\gamma \wedge B \to \epsilon) \end{array} \tag{5.22}$$

$$\begin{array}{l} <X, A \to \alpha \cdot \beta> \to <B \to X \cdot \delta><B, A \to \alpha \cdot \beta>, \\ \text{for all } B \to X\delta \text{ and } A \to \alpha\beta \text{ in } G \text{ such that } \exists_\gamma(\beta \overset{*}{\Rightarrow} B\gamma) \end{array} \tag{5.23}$$

$$<A \to \alpha\cdot> \to \epsilon, \text{ for all } A \to \alpha \text{ in } G \tag{5.24}$$

Here we do not prove that this is indeed a complete characterization of the result of the above construction. Later on we will encounter convincing indirect evidence.

Examples

Let G again be given by $S \to xy$, $S \to xSy$. Then F_G is as follows:

according to (5.19), $S \to <S \to \cdot xy>$ $S \to <S \to \cdot xSy>$;

according to (5.20),

$$\begin{array}{ll} <x, S \to \cdot xy> \to <S \to x \cdot y> & <y, S \to x \cdot y> \to <S \to xy\cdot> \\ <x, S \to \cdot xSy> \to <S \to x \cdot Sy> & <S, S \to x \cdot Sy> \to <S \to xS \cdot y> \ ; \\ <y, S \to xS \cdot y> \to <S \to xSy\cdot> & \end{array}$$

according to (5.21),

$$\begin{array}{ll} <S \to \cdot xy> \to x <x, S \to \cdot xy> & <S \to xS \cdot y> \to y <y, S \to xS \cdot y> \\ <S \to \cdot xSy> \to x <x, S \to \cdot xSy> & <S \to x \cdot Sy> \to x <x, S \to x \cdot Sy> \ ; \\ <S \to x \cdot y> \to y <y, S \to x \cdot y> & \end{array}$$

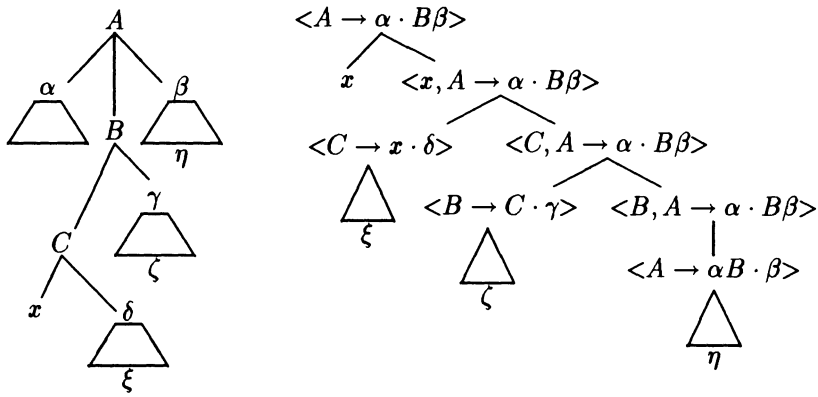

Figure 5.2 Subderivations in G and F_G

according to (5.23), $\begin{array}{l} <x, S \to x \cdot Sy> \to <S \to x \cdot y><S, S \to x \cdot Sy> \\ <x, S \to x \cdot Sy> \to <S \to x \cdot Sy><S, S \to x \cdot Sy> \end{array}$;

and according to (5.24), $<S \to xy\cdot> \to \epsilon \quad <S \to xSy\cdot> \to \epsilon$;

□

The correspondence between derivations in G and in F_G is explained most easily with pictures. If a derivation in G contains a subderivation such as the left one in figure 5.2, then the other tree in figure 5.2 is a subderivation in the corresponding derivation in F_G. A trapezium denotes a subderivation starting from a string of grammar symbols. The variables ξ, ζ, η denote elements of V_T^* generated by these subderivations. Observe that the derivation in F_G is completely different from the one in G. The terminal x is produced directly, at the top rule, whereas it was produced through a series of grammar rules in G. Note also that the terminal strings ξ, ζ, η appear in reversed positions as well.

As a final example, consider the derivation tree of figure 3.1. Its counterpart in F_G is figure 5.3.

Equivalence of G and F_G

From the above construction, it should be clear that for each derivation $S \xrightarrow{*} x_1 ... x_n$ in G and E_G there is a derivation in F_G, although the derivations

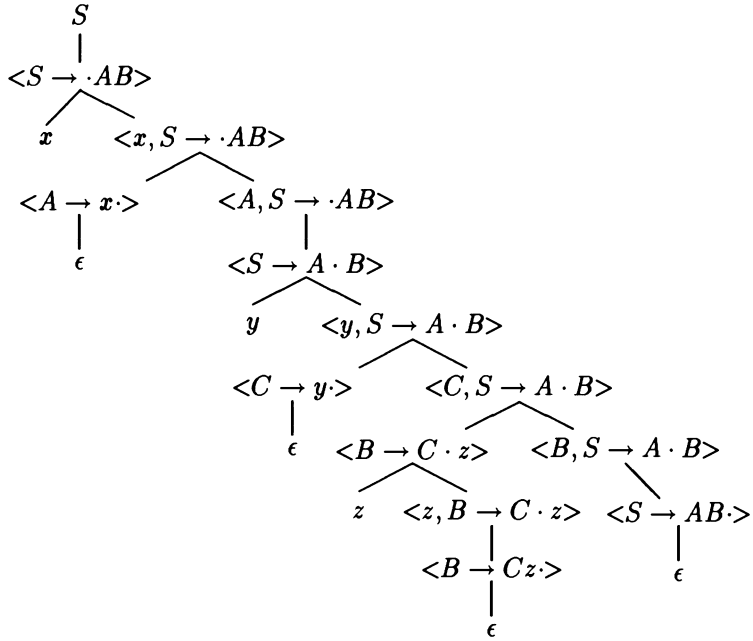

Figure 5.3 Derivation in F_G corresponding to figure 1.2

are very unsimilar. The reverse is also true. Let us prove $L_{F_G}(S) \leftarrow L_G(S)$ using the theorem of section 3.3. The function ϕ doing the job is

$$\phi(<A \rightarrow \alpha \cdot \beta>) = \alpha \backslash A$$
$$\phi(<X, A \rightarrow \alpha \cdot \beta>) = X \backslash (\alpha \backslash A)$$

Then (5.19)-(5.24) map to the following truths:

cf. (5.19) :	$S \geq^G S$	(type tautology)
cf. (5.20) :	$X \backslash (\alpha \backslash A) \geq^G \alpha X \backslash A$	(type tautology)
cf. (5.21) :	$\alpha \backslash A \geq^G x(x \backslash (\alpha \backslash A))$	(type tautology)
cf. (5.22) :	$\alpha \backslash A \geq^G B \backslash (\alpha \backslash A)$	(follows from $B \geq^G \epsilon$)
cf. (5.23) :	$X \backslash (\alpha \backslash A) \geq^G (X \backslash B)(B \backslash (\alpha \backslash A))$	(type tautology)
cf. (5.24) :	$\alpha \backslash A \geq^G \epsilon$	(follows from $A \geq^G \alpha$)

Properties of F_G

What are the properties of F_G? In particular, when is it left recursive? Let us look at the rules starting with a nonterminal, since these are the ones that

cause left recursion.

The only rules for nonterminals $<A \rightarrow \alpha \cdot \beta>$ that start with a nonterminal are of the form (5.22). If G has no ϵ-rules, these do not exist in F_G. So let us assume that G has no ϵ-rules. Then the only grammar rules for $<A \rightarrow \alpha \cdot \beta>$ are of the form (5.21), which start with a terminal symbol, and the ϵ-rules (5.24). Hence, there can be no left-recursive derivation starting from the nonterminal $<A \rightarrow \alpha \cdot \beta>$.

Next we have the rules (5.20), the right-hand side of which starts with a nonterminal of the form $<A \rightarrow \alpha \cdot \beta>$, which we just decided does not lead to left recursion, if G has no ϵ-rules.

Lastly, there are the rules (5.23). If $\delta \neq \epsilon$ the rules for the nonterminal $< B \rightarrow X \cdot \delta >$ all start with a terminal symbol, so that left recursion is impossible. However, if $\delta = \epsilon$, i.e., when a unary rule $B \rightarrow X$ exists in G, the relevant grammar rule is of type (5.24), $<B \rightarrow X \cdot \delta> \rightarrow \epsilon$, and we have

$$<X, A \rightarrow \alpha \cdot \beta> \stackrel{+}{\rightarrow} <B, A \rightarrow \alpha \cdot \beta>$$

if $\exists_\gamma(\beta \stackrel{*}{\Rightarrow} B\gamma \wedge B \rightarrow X)$. Now if $B \rightarrow X \rightarrow Y$, it follows also that

$$<Y, A \rightarrow \alpha \cdot \beta> \stackrel{+}{\rightarrow} <X, A \rightarrow \alpha \cdot \beta> \stackrel{+}{\rightarrow} <B, A \rightarrow \alpha \cdot \beta>,$$

et cetera. Summarizing, one has

$$<Y, A \rightarrow \alpha \cdot \beta> \stackrel{+}{\rightarrow} <B, A \rightarrow \alpha \cdot \beta>$$

if $\exists_\gamma(\beta \stackrel{*}{\Rightarrow} B\gamma \wedge B \stackrel{+}{\rightarrow} Y)$. Hence, if G is cyclic, with a derivation for $B \stackrel{+}{\rightarrow} B$, then also F_G is cyclic, and a fortiori left recursive, with a derivation for

$$<B, A \rightarrow \alpha \cdot \beta> \stackrel{+}{\rightarrow} <B, A \rightarrow \alpha \cdot \beta>$$

if $\exists_\gamma(\beta \stackrel{*}{\rightarrow} B\gamma)$. Conversely, if G is not cyclic and has no ϵ-rules, F_G is not left recursive.

5.4 RECURSIVE DESCENT FOR F_G

Grammar F_G is bilinear, and terminal symbols only appear as the first of two symbols at the right-hand sides of its rules. Therefore, the recursive descent algorithm (4.6) applies, and yields a recognizer in terms of the functions $[< A \rightarrow \alpha \cdot \beta >]$ and $[< X, A \rightarrow \alpha \cdot \beta >]$, which we write as $[A \rightarrow \alpha \cdot \beta]$ and $[X, A \rightarrow \alpha \cdot \beta]$. The grammar rules (5.19)-(5.24) map to inequalities on recognition functions, like (4.2):

$$
\begin{array}{ll}
cf.\ (5.19): & (S \rightarrow \alpha \rhd [S \rightarrow \cdot\alpha](i)) \leftarrow [S](i) \\
cf.\ (5.20): & [A \rightarrow \alpha X \cdot \beta](i) \leftarrow [X, A \rightarrow \alpha \cdot X\beta](i) \\
cf.\ (5.21): & (x = x_{i+1} \wedge \beta \overset{*}{\Rightarrow} x\gamma \rhd [x, A \rightarrow \alpha \cdot \beta](i+1)) \leftarrow [A \rightarrow \alpha \cdot \beta](i) \\
cf.\ (5.22): & (B \rightarrow \epsilon \wedge \beta \overset{*}{\Rightarrow} B\gamma \rhd [B, A \rightarrow \alpha \cdot \beta](i)) \leftarrow [A \rightarrow \alpha \cdot \beta](i) \\
cf.\ (5.23): & (B \rightarrow X\delta \wedge \beta \overset{*}{\Rightarrow} B\gamma \rhd \\
 & \qquad [B, A \rightarrow \alpha \cdot \beta]([B \rightarrow X \cdot \delta](i))) \leftarrow [X, A \rightarrow \alpha \cdot \beta](i) \\
cf.\ (5.24): & i \leftarrow [A \rightarrow \alpha \cdot](i).
\end{array}
$$

The intended minimal solution for the recognition function $[A \rightarrow \alpha \cdot \beta]$ is obtained by combining the inequalities for (5.21), (5.22), and (5.24) and the function for $[X, A \rightarrow \alpha \cdot \beta]$ by combining the inequalities for (5.20) and (5.23), in accordance with (2.1):

$$
\begin{array}{l}
[A \rightarrow \alpha \cdot \beta](i) = \\
\qquad x = x_{i+1} \wedge \beta \overset{*}{\Rightarrow} x\gamma \rhd [x, A \rightarrow \alpha \cdot \beta](i+1) \mid \\
\qquad B \rightarrow \epsilon \wedge \beta \overset{*}{\Rightarrow} B\gamma \rhd [B, A \rightarrow \alpha \cdot \beta](i) \mid \\
\qquad \beta = \epsilon \rhd i, \\
[X, A \rightarrow \alpha \cdot \beta](i) = \\
\qquad \beta = X\gamma \rhd [A \rightarrow \alpha X \cdot \gamma](i) \mid \\
\qquad B \rightarrow X\delta \wedge \beta \overset{*}{\Rightarrow} B\gamma \rhd [B, A \rightarrow \alpha \cdot \beta]([B \rightarrow X \cdot \delta](i)).
\end{array}
\tag{5.25}
$$

The above functions constitute a recognizer for G, which terminates for any input if G is not cyclic and has no ϵ-rules. The correctness depends on the equivalence of F_G, characterized above, and G. There exists a direct proof of the above functions too, though, since they are implementations of the following specifications:

$$
\begin{array}{l}
[A \rightarrow \alpha \cdot \beta](i) = \quad \beta \overset{*}{\rightarrow} x_{i+1} \ldots x_j \rhd j, \\
[X, A \rightarrow \alpha \cdot \beta](i) = \quad \beta \overset{*}{\Rightarrow} X\gamma \wedge \gamma \overset{*}{\rightarrow} x_{i+1} \ldots x_j \rhd j.
\end{array}
\tag{5.26}
$$

We postpone to the next chapter the proof that (5.25) implements (5.26). Its existence is circumstantial evidence for the equivalence of F_G and G.

If we assume a rule $S' \rightarrow S$, then, as above, the relation

$$S \xrightarrow{*} x_1 ... x_n \equiv n \leftarrow [S' \rightarrow .S](0)$$

solves the recognition problem, so that we do not need the function $[S]$.

5.4.1 States

The nonterminals of F_G can be interpreted as *states* a recognition process can be in. The interpretation can be verbalized as follows:

$A \rightarrow \alpha \cdot \beta$: α has been seen; does β follow? If so, A is recognized.

$X, A \rightarrow \alpha \cdot \beta$: α has been seen and after that X, which is a left corner of β. Can this derivation be completed?

With this description, the infix traversal of figure 5.3 provides the story of a plausible recognition process:

$S \rightarrow \cdot AB$: AB is looked for.

x : x is found.

$x, S \rightarrow \cdot AB$: x is a left corner of AB; can AB be completed?

$A \rightarrow x \cdot$: x has been seen, and we recognize A.

$A, S \rightarrow \cdot AB$: A is a left corner of AB; can AB be completed?

$S \rightarrow A \cdot B$: A has been seen; B is looked for.

y : y is found.

$y, S \rightarrow A \cdot B$: y is a left corner of B; can B be completed?

$C \rightarrow y \cdot$: y has been seen, and we recognize C.

$C, S \rightarrow A \cdot B$: C is a left corner of B; can B be completed?

$B \rightarrow C \cdot z$: C has been seen; z is looked for.

z : z is found.

$z, B \rightarrow C \cdot z$: z is the left corner of z; can it be completed?

$B \rightarrow Cz\cdot$: Cz has been seen, and we recognize B.

$B, S \rightarrow A \cdot B$: B is a left corner of B; can it be completed?

$S \rightarrow AB\cdot$: AB has been seen, and we recognize S.

This is no accident: this order of states corresponds exactly to the order of corresponding function calls in the recursive descent recognizer for F_G.

Exercises

1. Use the theorem of section 3.3 to prove that the language defined by E_G is included in the language defined by G, employing a mapping ϕ that is different from (5.5).
2. Prove that (5.25) is an implementation of (5.26) without looking at the next chapter.
3. "Prove" the grammar rule (5.12).
4. Prove that F_G is a cover of G, i.e., that there is a 1-1 mapping from derivations in F_G to derivations in G. Hint: if the rules (5.22) and (5.24) are applied, this corresponds to the application of grammar rules of G.
5. Prove that the language generated by the grammar

   $S \rightarrow x <x, S>$
   $<x, S> \rightarrow \epsilon$
   $<x, S> \rightarrow <S, S>$
   $<S, S> \rightarrow y$
   $<S, S> \rightarrow y <S, S>$

   is included in the language generated by $S \rightarrow Sy$, $S \rightarrow x$, using the theorem of section 3.3.

6

RECURSIVE ASCENT

A recursive descent recognizer for F_G is called a recursive *ascent* recognizer for G. The algorithm is akin to the Earley algorithm, itself discussed in section 7.5 of the next chapter, if the functions are memoized, especially if the algorithm is implemented in the interpretative way (see the discussion following (4.6)). The compiled implementation is potentially more efficient than the standard Earley algorithm. Deterministic recursive descent recognizers with look ahead, as in section 4.4, applied to grammars like F_G, are called PLR(k) recognizers in the literature. Our algorithm could thus be called a "PLR(0)-like recognizer," but it could also be called a nondeterministic left-corner recognizer, following yet another nomenclature from the world of deterministic parsing methods. From all potential names we choose to call the kind of algorithms we are now engaged in "Earley-like" or "recursive ascent Earley" parsers, to emphasize the nondeterministic nature of the algorithms and the fact that they are item-based.

It is fairly evident from the example in the preceding chapter that F_G in general is much larger than G. A recognizer, implemented in the compiled way, therefore consists of a relatively large number of functions. To reduce the size, we can write the function applications $[X, A \rightarrow \alpha \cdot \beta](i)$ as $\overline{[A \rightarrow \alpha \cdot \beta]}(X, i)$ and treat X as a function parameter rather than as part of the function name. Then the recognizer (5.25) consists, in the compiled implementation, of two functions for each item of the grammar.

The role of grammar transformations stops here. The readers who did not like the preceding chapter, or did not understand it, get a second chance at this point. In the remainder of this book, we will come up with algorithms that are less directly related to grammar transformations. All there is, from

now on, is specifications and implementations of the same kind as (5.26) and (5.25).

6.1 THE ALGORITHM

On the basis of the preceding chapter, let us see whether the recognition functions ("$\Rightarrow$" was defined in (5.18)) specified as

$$\begin{aligned} [A \to \alpha \cdot \beta](i) &= \beta \xrightarrow{*} x_{i+1} \ldots x_j \ \triangleright j, \\ \overline{[A \to \alpha \cdot \beta]}(X, i) &= \beta \overset{*}{\Rightarrow} X\gamma \wedge \gamma \xrightarrow{*} x_{i+1} \ldots x_j \ \triangleright j, \end{aligned} \tag{6.1}$$

which is (5.26) reformulated, are implemented by the corresponding reformulation of (5.25):

$$\begin{aligned} &[A \to \alpha \cdot \beta](i) = \\ &\qquad \beta \overset{*}{\Rightarrow} x_{i+1}\gamma \ \triangleright \overline{[A \to \alpha \cdot \beta]}(x_{i+1}, i+1) \ | \\ &\qquad \beta \overset{*}{\Rightarrow} B\gamma \wedge B \to \epsilon \ \triangleright \overline{[A \to \alpha \cdot \beta]}(B, i) \ | \\ &\qquad \beta = \epsilon \ \triangleright i \end{aligned} \tag{6.2}$$

$$\begin{aligned} &\overline{[A \to \alpha \cdot \beta]}(X, i) = \\ &\qquad \beta = X\gamma \ \triangleright [A \to \alpha X \cdot \gamma](i) \ | \\ &\qquad \beta \overset{*}{\Rightarrow} C\gamma \wedge C \to X\delta \ \triangleright \overline{[A \to \alpha \cdot \beta]}(C, [C \to X \cdot \delta](i)). \end{aligned} \tag{6.3}$$

In the last line, the function $\overline{[A \to \alpha \cdot \beta]}$ distributes over the bunch that is its second argument. The correctness of (6.2) is shown by the following proof. First we notice that

$$\begin{aligned} \beta \xrightarrow{*} x_{i+1} \ldots x_j \equiv\ & \exists_\gamma(\beta \overset{*}{\Rightarrow} x_{i+1}\gamma \wedge \gamma \xrightarrow{*} x_{i+2} \ldots x_j) \vee \\ & \exists_{B\gamma}(\beta \overset{*}{\Rightarrow} B\gamma \wedge B \to \epsilon \wedge \gamma \xrightarrow{*} x_{i+1} \ldots x_j) \vee \\ & (\beta = \epsilon \wedge i = j). \end{aligned} \tag{6.4}$$

Substituting this in the specification (6.1) of $[A \to \alpha \cdot \beta]$ and replacing existential quantifiers by implicit **let**'s (see chapter 2), one gets

$$
\begin{aligned}
[A \to \alpha \cdot \beta](i) = \\
& \beta \overset{*}{\Rightarrow} x_{i+1}\gamma \wedge \gamma \overset{*}{\to} x_{i+2}...x_j \ \rhd j \mid \\
& \beta \overset{*}{\Rightarrow} B\gamma \wedge B \to \epsilon \wedge \gamma \overset{*}{\to} x_{i+1}...x_j \ \rhd j \mid \\
& \beta = \epsilon \wedge i = j \ \rhd j.
\end{aligned}
$$

This directly leads to the implementation given above, using the specifications of $\overline{[A \to \alpha \cdot \beta]}(x_{i+1}, i+1)$ and $\overline{[A \to \alpha \cdot \beta]}(B, i)$.

For establishing the correctness of (6.3), notice that $\beta \overset{*}{\Rightarrow} X\gamma$ either consists of zero steps, in which case $\beta = X\gamma$, or contains at least one step:

$$
\begin{aligned}
\exists_\gamma(\beta \overset{*}{\Rightarrow} X\gamma \wedge \gamma \overset{*}{\to} x_{i+1}...x_j) \equiv \exists_\gamma(\beta = X\gamma \wedge \gamma \overset{*}{\to} x_{i+1}...x_j) \vee \\
\exists_{C\delta\gamma k}(\beta \overset{*}{\Rightarrow} C\gamma \wedge C \to X\delta \wedge \delta \overset{*}{\to} x_{i+1}...x_k \wedge \gamma \overset{*}{\to} x_{k+1}...x_j)
\end{aligned} \tag{6.5}
$$

Hence, $\overline{[A \to \alpha \cdot \beta]}(X, i)$ may be written as the union of

$$\beta = X\gamma \wedge \gamma \overset{*}{\to} x_{i+1}...x_j \ \rhd j \tag{6.6}$$

and

$$\beta \overset{*}{\Rightarrow} C\gamma \wedge C \to X\delta \wedge \delta \overset{*}{\to} x_{i+1}...x_k \wedge \gamma \overset{*}{\to} x_{k+1}...x_j \ \rhd j. \tag{6.7}$$

With the specification of $[A \to \alpha X \cdot \gamma]$, (6.6) may be rewritten as

$$\beta = X\gamma \ \rhd [A \to \alpha X \cdot \gamma](i).$$

Using the specification of $\overline{[A \to \alpha \cdot \beta]}(C, k)$, the bunch (6.7) becomes

$$\beta \overset{*}{\Rightarrow} C\gamma \wedge C \to X\delta \wedge \delta \overset{*}{\to} x_{i+1}...x_k \ \rhd \overline{[A \to \alpha \cdot \beta]}(C, k),$$

and finally, with the specification of $[C \to X \cdot \delta]$,

$$\beta \overset{*}{\Rightarrow} C\gamma \wedge C \to X\delta \ \rhd \overline{[A \to \alpha \cdot \beta]}(C, [C \to X \cdot \delta](i)).$$

□

Note that in (6.2) and (6.3) no functions $[A \to \cdot\alpha]$ are invoked, except for $[S' \to \cdot S](0)$. This will create a problem for the construction of parse trees, as we will see in the next chapter.

Note that one may also formulate recursive ascent recognition functions with string arguments, as in chapter 3. The specification (6.1) for string arguments is

$$\begin{aligned} [A \to \alpha \cdot \beta](\xi) = &\quad \beta \xrightarrow{*} \eta \wedge \xi = \eta\rho \rhd \rho, \\ \overline{[A \to \alpha \cdot \beta]}(X, \xi) = &\quad \beta \overset{*}{\Rightarrow} X\gamma \wedge \gamma \xrightarrow{*} \eta \wedge \xi = \eta\rho \rhd \rho. \end{aligned} \tag{6.8}$$

Below, we will work out a variant of this specification.

A function $[A \to \alpha \cdot \beta]$ calls $\overline{[A \to \alpha \cdot \beta]}$ with the left-corner *bottoms* of β as the first argument. These are the left corners that either are terminal symbols or rewrite directly to ϵ. $\overline{[A \to \alpha \cdot \beta]}(X, i)$ subsequently calls itself for all left corners that are rewritten in leftmost symbol derivations reaching the given left-corner X. The left corners of β are thus recognized in a bottom-up way: if $\beta \overset{*}{\Rightarrow} B_1\gamma \overset{*}{\Rightarrow} B_2\gamma$ then B_2 is recognized before B_1. The name recursive *ascent* reflects this bottom-up character of the recognition process.

The function (6.2) applied to n calls the function $\overline{[A \to \alpha \cdot \beta]}(x_{n+1}, n+1)$, which is in principle undefined. If we assume that x_{n+1} is some symbol that is not in V, then the clause $\beta \overset{*}{\Rightarrow} x_{n+1}\gamma$ is false, and $\overline{[A \to \alpha \cdot \beta]}(x_{n+1}, n+1)$ always produces *null*. Note that the clauses $\beta \overset{*}{\Rightarrow} x_{i+1}\gamma$ can be removed from (6.2) without invalidating the implementation. It was kept to indicate that in actual implementations a test at this point on the value of x_{i+1} might improve efficiency, and to stress the relation with the recognizer (5.25), in which a similar clause naturally came from the grammar F_G. Similar things are true for the clause $\beta \overset{*}{\Rightarrow} B\gamma$ in (6.2). In later implementations of related algorithms, we will not always worry about the problem of calling a function for x_{n+1}, and also the extra guards of the kind $\beta \overset{*}{\Rightarrow} x_{i+1}\gamma$ will no longer appear.

Example

Consider the grammar $S \rightarrow xSy$, $S \rightarrow \epsilon$. Then, in imperative style, the following are the recursive ascent functions for $S \rightarrow x \cdot Sy$:

```
[S → x · Sy](i) =   result:=null;
                    result:=result | [S → x · Sy]̄(S, i);
                    if x_{i+1} = x then
                        result:=result | [S → x · Sy]̄(x, i + 1);
                    fi;
                    return result;
```

```
[S → x · Sy]̄(X, i) =   result:=null;
                       if X = x then
                          for all k ← [S → x · Sy](i) do
                             result:=result | [S → x · Sy]̄(S, k);
                          od;
                       else if X = S then
                          result:=result | [S → xS · y](i)
                       fi;
                       return result;
```

6.2 TERMINATION

The above is an algorithm only if there are no termination problems. If the grammar has a cyclic derivation $B \xrightarrow{+} B$, the execution of $\overline{[A \rightarrow \alpha \cdot \beta]}(B, i)$ leads to a call of itself, and the algorithm does not terminate. Also, there may be a cycle of transitions labeled by nonterminals that derive ϵ. This occurs if for some k, one has that for $i = 1...k$,

$$A_i \rightarrow \alpha_i \beta_i \wedge \exists_\gamma (\beta_i \overset{*}{\Rightarrow} A_{i+1}\gamma) \wedge \alpha_i \xrightarrow{+} \epsilon$$

while $A_1 = A_{k+1} \wedge \alpha_1 = \alpha_{k+1} \wedge \beta_1 = \beta_{k+1}$. In this case, the execution of $[A_1 \rightarrow \alpha_1 \cdot \beta_1](i)$ leads to a call of itself, and the algorithm does not terminate. A cycle of this form occurs *iff* there is a derivation

$$A \xrightarrow{+} \alpha A \beta \text{ such that } \alpha \xrightarrow{+} \epsilon. \tag{6.9}$$

This refines the findings in the previous chapter. There we concluded that the algorithm always terminates if G is not cyclic and has no ϵ-rules. Here we find that there is only a problem with ϵ-rules if they indirectly make the grammar left recursive, as displayed in (6.9).

6.3 A VARIANT THAT WORKS WITH STRINGS

It is easy to define a variant of the recognizer that has no problems with derivations (6.9). It is obtained from dropping the restriction that the leftmost symbol derivation may not use ϵ-rules. Hence, we define (cf. (5.18))

$$\alpha \Rightarrow' \beta \equiv \exists_{B\gamma\delta}(\alpha = B\gamma \wedge \beta = \delta\gamma \wedge B \rightarrow \delta). \tag{6.10}$$

Then the above analysis can be repeated. The most elegant way is to use strings instead of numbers. Instead of (6.8) we now have

$$\begin{aligned} [A \rightarrow \alpha \cdot \beta](\xi) &= \beta \xrightarrow{*} \eta \wedge \xi = \eta\rho \rhd \rho, \\ \overline{[A \rightarrow \alpha \cdot \beta]}(X\xi) &= \beta \xrightarrow{*} X\eta \wedge \xi = \eta\rho \rhd \rho. \end{aligned} \tag{6.11}$$

$\overline{[A \rightarrow \alpha \cdot \beta]}$ looks more like it does in (6.8) if one uses the following equivalence to express $\beta \xrightarrow{*} X\xi$ in terms of $\Rightarrow'$:

$$\beta \xrightarrow{*} X\xi \equiv \exists_\gamma(\beta \overset{*}{\Rightarrow}{}' X\gamma \wedge \gamma \xrightarrow{*} \xi), \tag{6.12}$$

and if the argument $X\xi$ is split into X, ξ. As we wrote them in (6.11), the two recognition functions are similar but not identical: the argument of $[A \rightarrow \alpha \cdot \beta]$ consists of terminals and may be empty (it is in V_T^*), and the string β may derive ϵ. By contrast, the argument of $\overline{[A \rightarrow \alpha \cdot \beta]}$ may not be empty: its first argument is an arbitrary grammar symbol (the argument is in VV_T^*). Also, β may not derive ϵ here. Let us merge the two functions into one (δ is in $V_T^* \cup V_N V_T^*$):

$$
\begin{aligned}
&[A \to \alpha \cdot \beta](\delta) = \\
&\qquad \delta \in V_T^* \wedge \beta \xrightarrow{*} \epsilon \rhd A\delta \mid \\
&\qquad \delta = X\zeta \wedge \beta \xrightarrow{*} X\eta \wedge \zeta = \eta\rho \rhd A\rho.
\end{aligned} \tag{6.13}
$$

We "prefixed" the output of the functions with A, so that now

$$S' \leftarrow [S' \to \cdot S](\xi)$$

if and only if ξ is a correct sentence. The reason for this sneaky change will be clear from the result.

In (6.13), $\beta \xrightarrow{*} X\eta$ means that either X is introduced by a grammar rule $C \to \mu X \nu$, with $\mu \xrightarrow{*} \epsilon$, or X is already in β: $\beta = \mu X \gamma$, with $\mu \xrightarrow{*} \epsilon$:

$$
\begin{aligned}
&[A \to \alpha \cdot \beta](\delta) = \\
&\qquad \delta \in V_T^* \wedge \beta \xrightarrow{*} \epsilon \rhd A\delta \mid \\
&\qquad \delta = X\zeta \wedge \beta = \mu X\gamma \wedge \mu \xrightarrow{*} \epsilon \wedge \gamma \xrightarrow{*} \eta \wedge \zeta = \eta\rho \rhd A\rho \mid \\
&\qquad \delta = X\zeta \wedge \beta \xrightarrow{*} C\eta_1 \wedge C \to \mu X\nu \wedge \mu \xrightarrow{*} \epsilon \wedge \\
&\qquad\qquad\qquad \nu \xrightarrow{*} \eta_2 \wedge \zeta = \eta_2\eta_1\rho \rhd A\rho.
\end{aligned}
$$

With (6.13) one finally gets

$$
\begin{aligned}
&[A \to \alpha \cdot \beta](\delta) = \\
&\qquad \delta \in V_T^* \wedge \beta \xrightarrow{*} \epsilon \rhd A\delta \mid \\
&\qquad \delta = X\zeta \wedge \beta = \mu X\gamma \wedge \mu \xrightarrow{*} \epsilon \rhd [A \to \alpha\mu X \cdot \gamma](\zeta) \mid \\
&\qquad \delta = X\zeta \wedge \beta \xrightarrow{*} C\gamma \wedge C \to \mu X\nu \wedge \mu \xrightarrow{*} \epsilon \rhd \\
&\qquad\qquad\qquad [A \to \alpha \cdot \beta]([C \to \mu X \cdot \nu](\zeta)).
\end{aligned} \tag{6.14}
$$

Here we could use the distributivity of functions over bunches, in writing $[A \to \alpha \cdot \beta]([C \to \mu X \cdot \delta](\zeta))$, only because of the above sneaky change of the output of the recognition functions. In deriving (6.14) it is critical that $C \notin V_T$, or in other words, that terminals and nonterminals are disjoint: $V_N \cap V_T = \emptyset$. For earlier algorithms, such as (6.2),(6.3) this is not required.

If one has a grammar without ϵ-rules, (6.14) becomes even simpler

$$
\begin{aligned}
&[A \to \alpha \cdot \beta](\delta) = \\
&\qquad \delta \in V_T^* \wedge \beta = \epsilon \rhd A\delta \mid \\
&\qquad \delta = X\zeta \wedge \beta = X\gamma \rhd [A \to \alpha X \cdot \gamma](\zeta) \mid \\
&\qquad \delta = X\zeta \wedge \beta \xrightarrow{*} C\gamma \wedge C \to X\nu \rhd [A \to \alpha \cdot \beta]([C \to X \cdot \nu](\zeta)).
\end{aligned}
$$

6.4 COMPLEXITY

In the standard Earley algorithm (see section 7.5), a so-called parse matrix is used to store recognition results. Such a device is not needed if the above recognizer is implemented with memo-functions. The worst-case complexity analysis of the memoized recognizer is quite simple. Let n be the sentence length, $|G|$ the number of items of the grammar, p the number of nonterminals, and q the maximum number of items $C \to X \cdot \delta$ with the same X. Then, there are $O(|G|pn)$ different invocations of recognizer functions. Each invocation of a function $\overline{[A \to \alpha \cdot \beta]}$ invokes $O(qn)$ other functions that all result in a bunch with $O(n)$ elements. The merge of these bunches into one bunch with no duplicates can be accomplished in $O(qn^2)$ time on a random access machine. Hence, the total time-complexity is $O(|G|pqn^3)$. The space needed for storing function results is $O(n)$ per invocation, i.e., $O(|G|pn^2)$ for the whole recognizer. These complexity results are almost identical to the usual ones for Earley parsing. Only the dependence on the grammar variables $|G|, p$ and q slightly differs. Worst-case performance need not be relevant in practice. We estimate that for many practical grammars, the present algorithm is more efficient than existing implementations, if the functions are implemented in a smartly compiled way. Note, for instance, that the conditions

$$
\begin{aligned}
&\exists_\gamma(\beta \overset{*}{\Rightarrow} x_{i+1}\gamma) \\
&\exists_\gamma(\beta \overset{*}{\Rightarrow} B\gamma) \wedge B \to \epsilon \\
&\beta = X\gamma \\
&\exists_\gamma(\beta \overset{*}{\Rightarrow} C\gamma) \wedge C \to X\delta
\end{aligned}
$$

in (6.2,6.3) are independent of the input string in the sense that for every β, x_{i+1}, X the values of the free variables B, γ, C, δ that make them true can be computed before parsing.

The algorithms we are now engaged in are attractive alternatives to the standard Earley algorithm. This is especially true of the variant developed in the next chapter. For a discussion on the relation between recursive

ascent Earley parsers and the standard version of the Earley parser (with the parse matrix), see section 7.6. Here we point to one distinction between the two: recursive ascent parsers have the above-mentioned problem with cyclic grammars, whereas the standard Earley parser has no such problem. It is possible to make the above recognizer handle cyclic grammars as well by adding some form of loop detection to it. A convenient way of doing so is by extending the memoization technique. When a function $f(x)$ (f is $[q]$ or $\overline{[q]}$, x is i or X, i) is called, one of three things is true. Firstly, f may have been called before with the same parameter x values and has finished. Then the stored result may be reused. Secondly, it may be that f has not yet been called with x. Then the result must be computed. Thirdly, it may be that f has been called with x but has not yet been completed. Then a loop has been detected, and one may use the value *null* as a temporary function result. There is a caveat here: from the time that the temporary value for $f(x)$ is assumed until the time that $f(x)$ returns its definitive value, all function invocations may have used the temporary value of $f(x)$ in a critical way. This means that in the memoization table they should still be registered as "not evaluated." Invocation $f(x)$ itself, however, although it also used its own temporary value, can safely be registered as having been performed. This means that if a loop occurs, some invocations have to be computed (at most) twice. Because the number of loops is bounded, the algorithm with loop detection still has $O(n^3)$ time complexity.

The main practical virtue of the above implementation comes from the efficiency gain that can be achieved by implementing the functions in the compiled way. Concerning efficiency, however, the above algorithms are not the optimal ones. For instance, there is an additional improvement following from the observation that $[A \rightarrow \alpha \cdot \beta]$ and $\overline{[A \rightarrow \alpha \cdot \beta]}$ only depend on β. This means that functions for different items may be equivalent and can be identified. For an example of such functions, see (6.15) in the exercises.

6.5 EBNF GRAMMARS

If one wants to give a compact specification of the syntax of some language, one often uses an extension of context-free grammars, called Extended Backus Naur Form (EBNF) grammars. The right-hand sides of rules in these grammars are arbitrary regular expressions over grammar symbols, instead of only strings of grammar symbols. EBNF grammars are also called extended context-free grammars, or regular right-hand-side grammars. In com-

putational linguistics, they are called Recursive Transition Networks (RTN). With EBNF grammars one can describe exactly the same set of languages as with context-free grammars, since every EBNF grammar can be translated into a normal CF grammar by replacing each right-hand side by a regular (sub)grammar. The difference is what is called the strong generative power: the degree of the nodes in a parse tree (the number of sons a node can have) can be unbounded for trees according to an EBNF grammar. In theoretical settings, the treatment of rules with regular expressions is often a bit cumbersome. Hence, it is tempting to dismiss regular expressions with the excuse that they do not add to the weak generative power anyway. To make up for our succumbing to this temptation, we show in this section how to generalize results for normal context-free grammars to EBNF grammars.

A regular expression over V is defined as follows:

- $X \in V$ are regular expressions.
- If a, b are regular expressions, then $[a]$, $\{a\}$, $a|b$, and ab are regular expressions.

We use brackets to disambiguate regular expressions that would be ambiguous otherwise: $(A|B)C$ is different from $A|(BC)$. A regular expression a defines a bunch $C(a)$ of strings of grammar symbols, defined as follows (we use variables a, b to denote regular expressions):

- if $a \in V$, then $C(a) = a$.
- $C([a]) = \epsilon \,|\, C(a)$, $C(\{a\}) = \epsilon \,|\, C(a\{a\})$, $C(a|b) = C(a) \,|\, C(b)$, and $C(ab) = C(a)C(b)$.

If the regular expression contains a subexpression $\{a\}$, then the bunch of strings defined by it is infinite. An example is $A\{B|(C[D])\}$, which defines the bunch

$$A|AB|AC|ACD|ABC|ABB|ACC|ACDB|ACDC|\ldots\ .$$

Note that as long as $C(a)$ is finite, it can be viewed as a regular expression that is equivalent to, if longer than, a. The natural semantics of a grammar rule $A \rightarrow a$ is (L_G distributes over bunches)

$$L_G(C(a)) \leftarrow L_G(A).$$

As a rewriting rule, $A \rightarrow a$ can be used to rewrite an A into any α such that $\alpha \leftarrow C(a)$.

In a string of grammar symbols, each symbol has at most one symbol to its right, its successor. In fact, each string can be characterized by giving the first symbol, the last symbol, and the successor of every symbol except the last one. A general regular expression a with N_a symbols (elements of V) can likewise be characterized by (defining $T_a^+ = 1...N_a$ and $T_a = 0...N_a$)

1. a mapping $\phi_a : T_a^+ \mapsto V$ associating a grammar symbol with each number.
2. a function $succ_a : T_a \mapsto 2^{T_a^+}$ mapping each number to its *set* of successors. The regular expression can start with the symbols corresponding to the numbers in $succ_a(0)$.
3. a set $\sigma_a \in 2^{T_a}$ of numbers of symbols with which the regular expression can end.

We refer to T_a as the set of positions in a. Note that position 0 is not associated with a symbol in V and is not a possible element of $succ_a(k)$. It can be an element of σ_a, though, in which case there is an empty path through the regular expression, i.e., $\epsilon \leftarrow C(a)$.

If $a = A\{B|(C[D])\}$ then a is characterized by

$T_a = 0,1,2,3,4$	$\phi_a(1) = A$	$\phi_a(2) = B$
$\phi_a(3) = C$	$\phi_a(4) = D$	$succ_a(1) = \{2,3\}$
$succ_a(2) = \{2,3\}$	$succ_a(3) = \{2,3,4\}$	$succ_a(4) = \{2,3\}$
$succ_a(0) = \{1\}$	$\sigma_a = \{1,2,3,4\}$	

We define an item as a pair $(A \rightarrow a, k)$, with the interpretation that position k, with grammar symbol $\phi_a(k)$, is "just before the dot." The correspondence with dotted rules is the following. Let $\alpha = B_1...B_l$; then α is a simple regular expression characterized by $\phi_\alpha(k) = B_k$, $succ_\alpha(k) = \{k+1\}$ if $0 \leq k < l$, $succ_\alpha(l) = \emptyset$, and $\sigma_\alpha = \{l\}$. Item $(A \rightarrow \alpha, 0)$ corresponds to the initial item $A \rightarrow \cdot\alpha$ and $(A \rightarrow \alpha, k)$ to the dotted-rule item with the dot just after B_k.

Items $(A \rightarrow a, k)$ can be used to rewrite the EBNF rule $A \rightarrow a$, just as dotted rules were used to rewrite a normal context-free grammar rule:

$$
\begin{array}{l}
A \rightarrow <(A \rightarrow a, 0)> \\
<(A \rightarrow a, k)> \rightarrow \phi_a(l) <(A \rightarrow a, l)> \text{ if } l \in succ_a(k) \\
<(A \rightarrow a, k)> \rightarrow \epsilon \text{ if } k \in \sigma_a
\end{array}
$$

If EBNF grammar G is rewritten in this way, the result is again called E_G. The analysis of the preceding chapter and this chapter can be repeated, leading, via a grammar F_G and generalization

$$\alpha \Rightarrow \beta \equiv \exists_{B\gamma\delta\mu}(\alpha = B\gamma \wedge \beta = \delta\gamma \wedge B \rightarrow a \wedge \delta \leftarrow C(a) \wedge \delta \neq \epsilon)$$

of (5.18), to the following extension of (6.2),(6.3):

$$
\begin{array}{l}
[A \rightarrow a, k](i) = \\
\quad l \in succ_a(k) \wedge \phi_a(l) \overset{*}{\Rightarrow} x_{i+1}\gamma \ \rhd \overline{[A \rightarrow a, k]}(x_{i+1}, i+1) \ | \\
\quad B \rightarrow b \wedge 0 \in \sigma_b \wedge l \in succ_a(k) \wedge \phi_a(l) \overset{*}{\Rightarrow} B\gamma \ \rhd \overline{[A \rightarrow a, k]}(B, i) \ | \\
\quad k \in \sigma_a \ \rhd i
\end{array}
$$

$$
\begin{array}{l}
\overline{[A \rightarrow a, k]}(X, i) = \\
\quad X = \phi_a(l) \wedge l \in succ_a(k) \ \rhd [A \rightarrow a, l](i) \ | \\
\quad l \in succ_a(k) \wedge \phi_a(l) \overset{*}{\Rightarrow} C\gamma \wedge C \rightarrow b \wedge m \in succ_b(0) \ \wedge \\
\qquad\qquad X = \phi_b(m) \ \rhd \overline{[A \rightarrow a, k]}(C, [C \rightarrow b, m](i))
\end{array}
$$

Algorithms that will be developed in subsequent chapters can be extended to EBNF grammars in the same way. The strategy is to remove dotted rules and to use the functions $\phi, succ, \sigma$ instead.

Exercises

1. Implement the following variant of (6.1):

$$[A \rightarrow \alpha \cdot \beta](i) = \quad \beta \overset{*}{\rightarrow} x_{i+1} \ldots x_j \ \rhd (A, j)$$

$$\overline{[A \rightarrow \alpha \cdot \beta]}(X, i) = \quad \beta \overset{*}{\Rightarrow} X\gamma \wedge \gamma \overset{*}{\rightarrow} x_{i+1} \ldots x_j \ \rhd (A, j)$$

2. Show that $n \leftarrow [S](0)$ if and only if $S \xrightarrow{*} x_1 \ldots x_n$, if

$$
\begin{aligned}
&[\beta](i) = \\
&\qquad \beta \overset{*}{\Rightarrow} x_{i+1}\gamma \ \triangleright\ \overline{[\beta]}(x_{i+1}, i+1) \ | \\
&\qquad B \rightarrow \epsilon \wedge \beta \overset{*}{\Rightarrow} B\gamma \ \triangleright\ \overline{[\beta]}(B, i) \ | \\
&\qquad \beta = \epsilon \ \triangleright\ i \\
&\overline{[\beta]}(X, i) = \\
&\qquad \beta = X\gamma \ \triangleright\ [\gamma](i) \ | \\
&\qquad \beta \overset{*}{\Rightarrow} C\gamma \wedge C \rightarrow X\delta \ \triangleright\ \overline{[\beta]}(C, [\delta](i))
\end{aligned}
\tag{6.15}
$$

3. Verify that the above recognizer for EBNF grammars is indeed a generalization of (6.2),(6.3).

7

PARSE FOREST

The difference between a recognizer and a parser is that the latter identifies all possible derivations for given input. Usually, a set of parse trees is produced. If there are many parse trees for average sentences, it becomes an issue how to represent them as compactly as possible. This is done by constructing a parse forest. Parse forest construction will be formulated with the presupposition that a bottom-up recognition algorithm provides the necessary information. We find that the above recursive ascent recognizers do not give enough information. A revised recursive ascent recognizer does the job, however. We also discuss a conventional formulation of the Earley recognizer, and its relation to parse forests.

7.1 INFORMAL INTRODUCTION

The notion of parse forest is most easily introduced by way of pictures. In figure 3.1 we introduced a way to draw a syntax tree. The tree on the left in figure 7.1 is of this kind. The tree on the right of this figure is a picture of the same derivation. It uses auxiliary nodes labeled by strings of grammar symbols ($<ABC>$ and $<AB>$) to make the tree binary: each node has at most two sons. The second way of representing a syntax tree is used in a parse forest. For instance, if we have two derivations - the one of figure 7.1 and another that is almost the same, but with B deriving ϵ and C deriving yz - then figure 7.2 represents both trees together. Two phenomena in this picture should be noted. Firstly, instead of a line from $<ABC>$ to $<BC>$ there is now a line to a set of trees with top $<BC>$. The two elements of the set indicate that there are two alternatives. Both alternatives share the

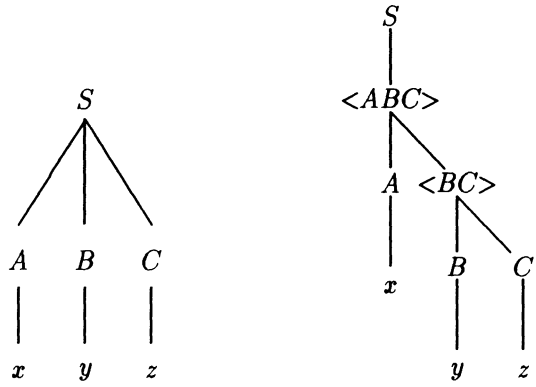

Figure 7.1 Two ways to display the same derivation

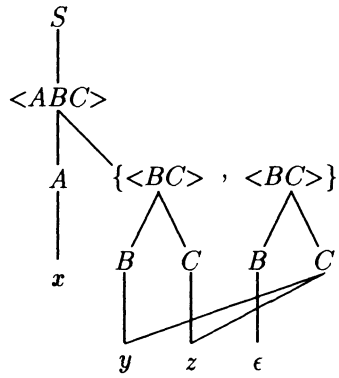

Figure 7.2 Parse forest

same context. For each syntax tree, one of the alternatives must be chosen. Secondly, the terminal nodes y and z are shared by the subtrees emanating from the two $<BC>$'s.

In a parse forest, these two kinds of convergence, sharing of context and sharing of nodes, are exploited in a general way. Any number of trees may be put in a set, and not just trees that are labeled by auxiliary symbols. Any node, not just the terminal ones, but also the nodes labeled with nonterminals and the auxiliary nodes, can be shared arbitrarily many times. This enables the representation of all derivations for noncyclic grammars and arbitrary sentences in cubic space. This is explained below in a mathematical way.

7.2 THE GRAMMAR E'_G

Formally, the right-hand tree of figure 7.1 is a tree for a transformed grammar that is akin to the grammar E_G that we encountered in chapter 5. The transformation is given by changing each non-ϵ grammar rule $A \rightarrow X_1 ... X_n$ into:

$$
\begin{array}{l}
A \rightarrow <X_1 ... X_n> \\
<X_1 ... X_n> \rightarrow X_1 <X_2 ... X_n> \\
<X_2 ... X_n> \rightarrow X_2 <X_3 ... X_n> \\
\qquad\qquad . \\
\qquad\qquad . \\
\qquad\qquad . \\
<X_n> \rightarrow X_n
\end{array}
$$

We call the resulting grammar E'_G.

If a forest of trees for E'_G is constructed, it is important to know what is involved in translating such trees into trees for G. The middle tree of figure 7.3 is (part of) a syntax tree according to E'_G, the corresponding tree according to G being given by the left tree. The mapping seems to involve the deletion of the nodes that are labeled with auxiliary nonterminals. However, in actual implementations, trees are not represented as in the left tree. Rather, a tree is encoded as a top node plus a list of son trees. If one uses programming languages like Pascal or C, without built-in list types, a node's list of son trees is most conveniently implemented as a linked-list structure involving auxiliary nodes that connect the elements of the list.

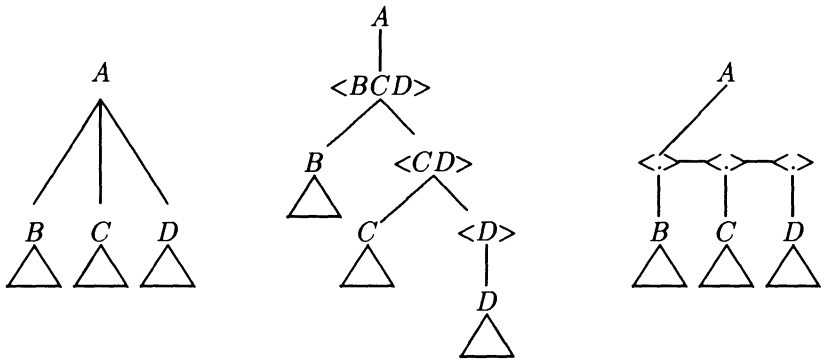

Figure 7.3 The tree on the left is a syntax tree of G. The middle is the corresponding tree of E'_G. The tree on the right is the same as the middle one, but it abstracts from the names of auxiliary nodes, which are displayed horizontally.

Now, a different interpretation of the middle tree of figure 7.3 is given by the figure's right tree. The labels of auxiliary nodes are simply discarded, but the nodes, and the binary topology of the tree, remain. However, the nodes for the auxiliary nonterminals, displayed horizontally now, are viewed as the auxiliary nodes of a linked list. In this case the list contains the trees with tops B, C, D. Summarizing, the transformation from a binary tree of E'_G to the corresponding tree of G only involves a reinterpretation of the auxiliary nodes. No structural change of the tree is involved and thus *no* computation.

7.3 FOREST FOR BILINEAR GRAMMARS

The grammar E'_G has three types of rules, namely, binary rules, unary rules, and ϵ-rules (in case the original grammar has ϵ-rules):

$$A \rightarrow XY,$$
$$A \rightarrow X,$$
$$A \rightarrow \epsilon.$$

In this section, we study grammars G that have only such rules. The possible rule applications (see section 3.2) are thus

$$\begin{aligned}&(A,\eta\rho) \rightarrow (X,\eta)(Y,\rho),\\&(A,\eta) \rightarrow (X,\eta),\\&(A,\epsilon) \rightarrow (\epsilon,\epsilon).\end{aligned} \tag{7.1}$$

A pair (X,η) is called a dressed grammar symbol. The assumption in a rule application in which this pair occurs on the right-hand side is that η is in $L_G(X)$. For parsing purposes, one is interested in the dressed grammar symbols for which this assumption is true for a substring η of the input sentence ξ. The set of such dressed grammar symbols is called $\mathcal{T}(\xi)$. It is defined as the smallest set that satisfies the following:

1. $(x,x) \in \mathcal{T}(\xi)$ for all terminals x that occur in ξ.
2. $(A,\epsilon) \in \mathcal{T}(\xi)$ if there is a grammar rule $A \rightarrow \epsilon$, i.e., if $(A,\epsilon) \rightarrow (\epsilon,\epsilon)$.
3. $(A,\delta) \in \mathcal{T}(\xi)$ if $(A,\delta) \rightarrow (X,\delta)$ is a rule application, with $(X,\delta) \in \mathcal{T}(\xi)$.
4. $(A,\eta\rho) \in \mathcal{T}(\xi)$ if there is a rule application $(A,\eta\rho) \rightarrow (X,\eta)(Y,\rho)$, with $(X,\eta) \in \mathcal{T}(\xi)$ and $(Y,\rho) \in \mathcal{T}(\xi)$ and $\eta\rho$ is a substring of ξ.

This is a constructive definition: it describes a simple bottom-up algorithm to calculate $\mathcal{T}(\xi)$. The algorithm initializes $\mathcal{T}(\xi)$ with the elements implied by the nonrecursive clauses 1,2 and subsequently adds new elements according to the recursive clauses 3,4 until no more new elements can be added. For historical reasons, the set $\mathcal{T}(\xi)$ is called a *parse matrix*. The algorithm that constructs the parse matrix is a bottom-up recognizer. From the parse matrix it can be deduced whether the sentence ξ is grammatically correct: (S,ξ) must be in $\mathcal{T}(\xi)$.

Whether or not a given string of terminal symbols is correct is only one interesting question. This chapter is about proofs that show such correctness, i.e., about (collections of) derivations. One representation of such a proof is a set of rule applications (see section 3.2). A set P of rule applications is a valid proof for (S,ξ) if with the above four clauses it can be proven that $(S,\xi) \in \mathcal{T}(\xi)$, using only rule applications of P. Consider the grammar

$$\begin{aligned}&S \rightarrow SS,\\&S \rightarrow x.\end{aligned} \tag{7.2}$$

The string xxx is correct according to this grammar. One proof is formed by the following set of rule applications:

$$\{(S, xxx) \rightarrow (S, xx)(S, x),\ (S, xx) \rightarrow (S, x)(S, x),\ (S, x) \rightarrow (x, x)\}.$$

A set of rule applications P may "hide" any number of derivations. If this number is greater than zero, then P is a valid proof, and it is possible to construct parse trees from it, one for each derivation. Given a set P of rule applications, supposed to be a proof, function *arb* creates the set of parse trees encoded by P. The function is defined for dressed grammar symbols (X, δ) that occur in rule applications in P, as follows:

$$
\begin{aligned}
arb(X, \delta) \stackrel{def}{=}\ & \\
& \{\ s \mid X \text{ is a terminal and } s = \delta = X\} \cup \\
& \{\ s \mid \delta = \epsilon \text{ and } (X, \epsilon) \rightarrow (\epsilon, \epsilon) \in P \text{ and } s = \begin{array}{c} X \\ | \\ \epsilon \end{array}\ \} \cup \\
& \{\ s \mid (X, \delta) \rightarrow (Y, \delta) \in P \text{ and } t \in arb(Y, \delta) \text{ and } s = \begin{array}{c} X \\ | \\ t \end{array}\ \} \cup \\
& \{\ s \mid (X, \delta) \rightarrow (Y, \eta)(Z, \rho) \in P \text{ and} \\
& \qquad t_1 \in arb(Y, \eta) \text{ and } t_2 \in arb(Z, \rho) \text{ and } s = \begin{array}{c} X \\ \diagup\ \diagdown \\ t_1 \qquad t_2 \end{array}\ \}.
\end{aligned}
$$

If $arb(X, \delta)$ terminates and does not produce the empty set, then δ is in $L_G(X)$, and P is a valid proof for this. If the result is empty, P does not prove that δ is in the language associated with X. The function *arb* need not terminate if the grammar is cyclic. For this reason we (once more) exclude cyclic grammars from our considerations.

For very ambiguous sentences, the number of parse trees is an exponential function of the sentence length. Take, for instance, the grammar (7.2), which produces the language that consists of all finite strings of x's. According to this grammar, there are as many parse trees for a given string of n x's as there are binary trees with n leaves, which number is exponential in n. If there are exponentially many parse trees, their storage takes up an exponential amount of space if they are all stored separately. Fortunately, the set of all parse trees can be represented more compactly in polynomial space, with the sharing techniques that were discussed above. In particular, different parse trees may

have subtrees in common, and, instead of letting each parse tree have its own copy of such a common subtree, one can create only one instance and have various parse trees refer to it with a pointer. This is accomplished by the function *for*, which turns the above parse matrix into a *parse forest*:

$$
\begin{aligned}
for(X,\delta) \stackrel{def}{=} & \text{ a } \underline{\text{pointer}} \text{ to the set} \\
& \{\ s \mid X \text{ is a terminal and } s = \delta = X\ \} \cup \\
& \{\ s \mid \delta = \epsilon \text{ and } X \rightarrow \epsilon \text{ and } s = X(\epsilon)\ \} \cup \\
& \{\ s \mid (X,\delta) \rightarrow (Y,\delta) \text{ is a rule application and } (Y,\delta) \in \mathcal{T}(\xi) \text{ and} \\
& \qquad s = X(for(Y,\delta))\ \} \cup \\
& \{\ s \mid (X,\delta) \rightarrow (Y,\eta)(Z,\rho) \text{ is a rule application and } (Y,\eta),(Z,\rho) \in \mathcal{T}(\xi) \\
& \qquad \text{and } s = X(for(Y,\eta), for(Z,\rho))\ \}
\end{aligned}
$$

Provided that $(S,\xi) \in \mathcal{T}(\xi)$, the pointer $for(S,\xi)$ refers to the parse forest for the entire sentence ξ. Function *for* is similar to *arb*, but the differences are important. Whereas in *arb*, tree segments are created that bind new nodes X to individual trees t, t_1, t_2, in *for* the tree segments s are connected to forests of trees via pointers. Due to the use of pointers, the size of each tree segment s is independent of properties of the sentence. A direct consequence of this is that the size of what is created in the body of $for(X,\delta)$ is proportional to the number of rule applications having left-hand side (X,δ). In the computation of $for(S,\xi)$, a function invocation $for(X,\eta)$ may appear more than once. This means that there is a danger that multiple copies of the same subforest are created, which is not the intention. This means that the function for should be a *memo*-function. Then it follows that the size of the whole parse forest is at most proportional to the total number of rule applications allowed by the grammar.

It may seem strange that a notion like pointer enters our mathematical discussion, and indeed it can be avoided. There always exists a minimal proof P, a set of rule applications, from which all parse trees can be obtained with the function *arb*. In an abstract sense, then, this P can also be viewed as a representation of the parse forest. The relevant collection of rule applications can be obtained from the parse matrix with an algorithm that has the same structure as *for*.

Whereas the parse forest may encode an exponential number of trees, its size is at most proportional to the number of rule applications. Let us investigate this number. Note that there are $O(|\xi|^2)$ different substrings of ξ. It follows that there are also $O(|\xi|^2)$ left-hand sides of rule applications. For each left-hand side (A, δ) the number of binary rule applications in (7.1) is proportional to the number of ways that δ can be split into two parts, which is $|\delta + 1|$. Since $|\delta| = O(|\xi|)$, we have that there are $O(|\xi|^2) \times O(|\xi|) = O(|\xi|^3)$ binary rule applications. For each left-hand side, the number of nonbinary applications in (7.1) is bounded by a constant, so these can be neglected in this worst-case analysis. It follows that the size of the parse forest is at most a cubic function of the length of the sentence.

Note how critically this result depends on our assumption that there are no grammar rules with left-hand sides longer than two grammar symbols. If a parse forest is constructed according to the above schema for arbitrary grammar rules, its maximum size will be $O(n^k)$, with k the maximum number of nonterminals on the right-hand sides of grammar rules.

In practice, substrings of ξ are always encoded by a pair of integers. If $\xi = x_1 ... x_n$, then the pair (i, j) stands for $x_{i+1} ... x_j$. So, if $\delta = x_{i+1} ... x_j$, we write (A, δ) as (A, i, j). All of the above can be repeated for numbers instead of strings. This is left as an exercise.

7.4 THE SET Q

Let us apply the preceding section to E'_G. The bottom-up recognizer computes a parse matrix that consists of triples (I, i, j), where I is a grammar symbol of E'_G, i.e., an auxiliary grammar symbol $<\alpha>$ or a grammar symbol of G. Triple (I, i, j) is added without inspection of the input symbols before $i + 1$ and after j. Therefore, (I, i, j) need not be useful for the parse forest: it may happen that, in complete derivations of the whole input, $x_{i+1} ... x_j$ is never analyzed as being derived from I. So, only some subset of the parse matrix will be actually useful. Such a subset will be called a filtered parse matrix. In fact, triples $(<\beta>, i, j)$ (which we will write as (β, i, j) from now on) and (A, k, j) are needed only if there is a derivation

$$S \xrightarrow{*} x_1 ... x_k A x_{j+1} ... x_n \rightarrow x_1 ... x_k \alpha \beta x_{j+1} ... x_n \xrightarrow{*} x_1 ... x_i \beta x_{j+1} ... x_n \xrightarrow{*} x_1 ... x_n.$$

To compute a proper subset of the parse matrix that contains sufficient triples,

one needs an algorithm that is more sophisticated than the pure bottom-up algorithm that computes the full parse matrix. For us, the obvious approach is to try to obtain a filtered parse matrix from the result of recursive ascent recognition. The simplest way to extract the appropriate triples from the recognizers as we have defined them, is to assume an implementation of memo-functions that enables access to the memoized function results, after execution of the recognizer. For instance, from the memoized results of (6.2) all (β, i, j) can be gathered such that a function $[A \rightarrow \alpha \cdot \beta](i)$ was invoked and produced a bunch that includes j. Note, however, that in this way one does not get the triples (β, i, j) in which β is the full right-hand side of a grammar rule, since no functions $[A \rightarrow \cdot\beta]$ are invoked. To obtain these triples in a neat way as well, it is most convenient to revise the specifications in the following way:

$$
\begin{aligned}
&[A \rightarrow \alpha \cdot \beta](i) = \\
&\qquad \beta \xrightarrow{*} x_{i+1} \ldots x_j \ \triangleright (A \rightarrow \alpha \cdot \beta, j) \mid \\
&\qquad \beta \overset{*}{\Rightarrow} B\mu \wedge B \rightarrow \gamma \wedge \gamma \xrightarrow{*} x_{i+1} \ldots x_j \ \triangleright (B \rightarrow \cdot\gamma, j) \\
&\overline{[A \rightarrow \alpha \cdot \beta]}(X, i) = \\
&\qquad \beta \overset{*}{\Rightarrow} X\gamma \wedge \gamma \xrightarrow{*} x_{i+1} \ldots x_j \ \triangleright (A \rightarrow \alpha \cdot \beta, j) \mid \\
&\qquad \beta \overset{*}{\Rightarrow} B\mu \wedge B \rightarrow \gamma \wedge \gamma \overset{*}{\Rightarrow} X\nu \wedge \nu \xrightarrow{*} x_{i+1} \ldots x_j \\
&\qquad\qquad \triangleright (B \rightarrow \cdot\gamma, j)
\end{aligned}
\tag{7.3}
$$

The corresponding implementation will be derived below. If we define, for the revised recognizer,

$$
\begin{aligned}
Q = \{&(\beta, i, j) | [I](i) \text{ was invoked and } (A \rightarrow \alpha \cdot \beta, j) \leftarrow [I](i)\} \cup \\
\{&(A, i, j) | [I](i) \text{ was invoked and } (A \rightarrow \cdot\beta, j) \leftarrow [I](i)\} \cup \\
\{&(x_{i+1}, i, i+1) | 0 \leq i < n\},
\end{aligned}
\tag{7.4}
$$

then Q is a filtered parse matrix for E'_G, as it was defined above by the bottom-up procedure, with numbers instead of strings. It contains sufficient triples to compute the parse forest. The memo-table of (7.3) also yields a filtered parse matrix for E_G, with triples (J, i, j) such that $(J, j) \leftarrow [I](i)$, for some I. In general, however, the filtered parse matrix for E'_G will be smaller and will lead therefore to a smaller parse forest.

One might wonder why the recognizer had to be revised. The reason is that (6.2) and (6.3) are what they were constructed to be: a natural recognizer for

F_G, and only indirectly one for G. For this reason, the parse trees that are most naturally produced from (6.2) and (6.3) are according to F_G. Although it is possible to translate individual parse trees according to F_G into trees according to G, it is difficult to produce a compact parse forest in this way.

Implementation of the revised recognizer

Let us derive an implementation of functions specified by (7.3). To simplify notation, we define $\overline{A \to \alpha \cdot \beta} = \{A \to \alpha \cdot \beta\} \cup pred(A \to \alpha \cdot \beta)$, where

$$pred(A \to \alpha \cdot \beta) = \{B \to \cdot\gamma | \beta \overset{*}{\Rightarrow} B\mu \wedge B \to \gamma\} \tag{7.5}$$

is the set of *predict* items for $A \to \alpha \cdot \beta$. With this definition, (7.3) can be reformulated as

$$\begin{array}{l} [A \to \alpha \cdot \beta](i) = \\ \qquad B \to \gamma \cdot \delta \in \overline{A \to \alpha \cdot \beta} \wedge \delta \overset{*}{\to} x_{i+1} \ldots x_j \ \triangleright (B \to \gamma \cdot \delta, j) \\ \overline{[A \to \alpha \cdot \beta]}(X, i) = \\ \qquad B \to \gamma \cdot \delta \in \overline{A \to \alpha \cdot \beta} \wedge \delta \overset{*}{\Rightarrow} X\mu \wedge \mu \overset{*}{\to} x_{i+1} \ldots x_j \\ \qquad\quad \triangleright (B \to \gamma \cdot \delta, j) \end{array} \tag{7.6}$$

Using (6.4) one gets

$$\begin{array}{l} [A \to \alpha \cdot \beta](i) = \\ \qquad B \to \gamma \cdot \delta \in \overline{A \to \alpha \cdot \beta} \wedge \delta \overset{*}{\Rightarrow} x_{i+1}\mu \wedge \mu \overset{*}{\to} x_{i+2} \ldots x_j \\ \qquad\quad \triangleright (B \to \gamma \cdot \delta, j) \mid \\ \qquad B \to \gamma \cdot \delta \in \overline{A \to \alpha \cdot \beta} \wedge \delta \overset{*}{\Rightarrow} C\mu \wedge C \to \epsilon \wedge \mu \overset{*}{\to} x_{i+1} \ldots x_j \\ \qquad\quad \triangleright (B \to \gamma \cdot \delta, j) \mid \\ \qquad B \to \gamma \cdot \in \overline{A \to \alpha \cdot \beta} \ \triangleright (B \to \gamma \cdot, i) \end{array}$$

which may be rewritten as

$$\begin{array}{l} [A \to \alpha \cdot \beta](i) = \\ \qquad \overline{[A \to \alpha \cdot \beta]}(x_{i+1}, i+1) \mid \\ \qquad C \to \cdot \in \overline{A \to \alpha \cdot \beta} \ \triangleright \overline{[A \to \alpha \cdot \beta]}(C, i) \mid \\ \qquad B \to \gamma \cdot \in \overline{A \to \alpha \cdot \beta} \ \triangleright (B \to \gamma \cdot, i) \end{array} \tag{7.7}$$

With (6.5) we get

$$
\begin{aligned}
&\overline{[A \to \alpha \cdot \beta]}(X, i) = \\
&\quad B \to \gamma \cdot \delta \in \overline{A \to \alpha \cdot \beta} \wedge \delta = X\mu \wedge \mu \xrightarrow{*} x_{i+1} \ldots x_j \\
&\qquad \triangleright (B \to \gamma \cdot \delta, j) \mid \\
&\quad B \to \gamma \cdot \delta \in \overline{A \to \alpha \cdot \beta} \ \wedge \delta \overset{*}{\Rightarrow} C\mu \wedge C \to X\nu \ \wedge \\
&\qquad \nu \xrightarrow{*} x_{i+1} \ldots x_k \wedge \mu \xrightarrow{*} x_{k+1} \ldots x_j \ \triangleright (B \to \gamma \cdot \delta, j)
\end{aligned}
$$

which in turn leads to

$$
\begin{aligned}
&\overline{[A \to \alpha \cdot \beta]}(X, i) = \\
&\quad B \to \gamma \cdot X\mu \in \overline{A \to \alpha \cdot \beta} \ \wedge \\
&\qquad (B \to \gamma X \cdot \mu, j) \leftarrow [B \to \gamma X \cdot \mu](i) \ \triangleright (B \to \gamma \cdot X\mu, j) \mid \\
&\quad C \to \cdot X\nu \in pred(A \to \alpha \cdot \beta) \wedge (C \to X \cdot \nu, k) \leftarrow [C \to X \cdot \nu](i) \ \triangleright \\
&\qquad \overline{[A \to \alpha \cdot \beta]}(C, k)
\end{aligned}
\tag{7.8}
$$

Equations (7.7) and (7.8) constitute the desired implementation. It is left as an exercise to make a variant that terminates for all non-cyclic grammars, by treating ϵ-rules as in section 6.3. The above recognizer and its variants are even more closely related to the Earley parser than foregoing ones. In practical implementations, the recognition and forest-building phases can be interleaved, using side effects in the recognition functions. We leave the details of this to the interested reader.

7.5 STANDARD EARLEY PARSER

In recursive ascent Earley parsers, the memo-table is interpreted as a parse matrix, the creation of which may thus be seen as a side effect. By contrast, conventional presentations of Earley parsing are explicitly centered around the creation of a parse matrix. Therefore, this is an appropriate place to mention such a presentation. It is based on the grammar transformation $G \mapsto \overline{E}_G$, given by the following rewriting of $A \to X_1 \ldots X_n$:

$$
\begin{array}{l}
A \rightarrow <A \rightarrow X_1 ... X_n \cdot>, \\
<A \rightarrow X_1 ... X_n \cdot> \rightarrow <A \rightarrow X_1 ... X_{n-1} \cdot X_n> X_n, \\
<A \rightarrow X_1 ... X_{n-1} \cdot X_n> \rightarrow <A \rightarrow X_1 ... X_{n-2} \cdot X_{n-1} X_n> X_{n-1}, \\
\qquad\qquad\qquad\qquad \vdots \\
<A \rightarrow X_1 \cdot X_2 ... X_n> \rightarrow <A \rightarrow \cdot X_1 ... X_n> X_1, \\
<A \rightarrow \cdot X_1 ... X_n> \rightarrow \epsilon.
\end{array}
$$

Note the duality between the grammars $\overline{E}_G$ and E_G (see chapter 5).

The standard Earley algorithm constructs a filtered parse matrix for $\overline{E}_G$. Elements of the parse matrix are triples (I, i, j), where I is a symbol of $\overline{E}_G$, i.e., an item or a grammar symbol of G. The above bottom-up procedure for constructing the parse matrix, applied to $\overline{E}_G$, with integers instead of strings, reads:

1. $(x_{i+1}, i, i+1) \in \mathcal{T}(\xi)$ for $i = 0 ... n-1$.
2. $(< A \rightarrow \cdot\alpha >, i, i) \in \mathcal{T}(\xi)$ for all rules $A \rightarrow \alpha$ of G.
3. $(< A \rightarrow \alpha X \cdot \beta >, i, j) \in \mathcal{T}(\xi)$ if, for some k, $(< A \rightarrow \alpha \cdot X\beta >, i, k) \in \mathcal{T}(\xi)$ and $(X, k, j) \in \mathcal{T}(\xi)$.
4. $(A, i, j) \in \mathcal{T}(\xi)$ if, for some α, $(< A \rightarrow \alpha\cdot >, i, j) \in \mathcal{T}(\xi)$.

A sufficient subset (like Q) is obtained by replacing the second clause in the last algorithm by

2a. $(< S \rightarrow \cdot\alpha >, 0, 0) \in \mathcal{T}(\xi)$.

2b. $(< A \rightarrow \cdot\alpha >, i, i) \in \mathcal{T}(\xi)$ if $A \rightarrow \alpha$ is a grammar rule and there is some $(< B \rightarrow \beta \cdot A\gamma >, k, i) \in \mathcal{T}(\xi)$.

This is called *prediction* in the Earley jargon. Clauses 1 and 2a are the initialization of the algorithm now. Clause 2b is recursive. The effect of this replacement is that it makes the recognition process scan the input from left to right, adding only elements (I, i, j) to the parse matrix that are useful for at least one completion of the string $x_1 ... x_j$ into a correct sentence.

The standard Earley algorithm leads to a parse forest for $\overline{E}_G$. Parse trees for $\overline{E}_G$ are related to parse trees for G in the same trivial way as parse trees for E'_G. In both cases, on the basis of the forest one can enumerate all trees according to the original grammar, without substantial computation.

7.6 EARLEY VERSUS EARLEY

Above, two superficially different algorithms are called "Earley parser." The first is a modified recursive descent parser for F_G. The modification was necessary because the original recognizer did not produce enough triples. The other is a modified bottom-up recognizer for $\overline{E}_G$. Here the modification was made because the original recognizer produces too many triples. The first parser computes a global data structure through memoization; the second computes an explicit parse matrix. The first parser is recursive and the second is iterative. Technically, both parsers are totally different. In some respects they are each other's dual:

Both have bottom-up and top-down properties. The first parser is top-down, but grammar F_G is a transformation of G in such a way that derivations are turned upside down (see chapter 5). The second parser is bottom-up, but the predict action gives it a top-down character as well. Hence, in one the bottom-up flavor is added on a top-down background; in the other the top-down flavor is added on a bottom-up background.

There is a duality between the binary representations of the parse trees in the forests that can be constructed with them. This is caused by the fact that the memo-table for the recursive ascent Earley parser yields a filtered parse matrix for E_G, as explained in section 7.4, whereas the standard Earley parser creates a filtered parse matrix for $\overline{E}_G$.

The parse matrix for E_G that can be deduced from the memo-table of the recursive ascent Earley parser of this chapter can also be characterized along the lines of section 7.5. However, we need an auxiliary set $\mathcal{C}(\xi)$, in addition to a parse matrix $\mathcal{T}(\xi)$. The latter is defined in terms of the former and itself:

1. $(x_{i+1}, i, i+1) \in \mathcal{T}(\xi)$ for $i = 0...n-1$.
2. $(< A \rightarrow \alpha\cdot >, i, i) \in \mathcal{T}(\xi)$ if $(< A \rightarrow \alpha\cdot >, i) \in \mathcal{C}(\xi)$.

3. $(< A \rightarrow \alpha \cdot X\beta >, i, j) \in \mathcal{T}(\xi)$ if $(< A \rightarrow \alpha \cdot X\beta >, i) \in \mathcal{C}(\xi)$ and, for some k, $(< A \rightarrow \alpha X \cdot \beta >, k, j) \in \mathcal{T}(\xi)$ and $(X, i, k) \in \mathcal{T}(\xi)$.

4. $(A, i, j) \in \mathcal{T}(\xi)$ if, for some α, $(< A \rightarrow \cdot\alpha >, i, j) \in \mathcal{T}(\xi)$.

Again, the parse matrix contains the recognized items. The four clauses here are clearly reminiscent of the four clauses of the above Earley parser without prediction. If the checks involving $\mathcal{C}(\xi)$ are removed, one gets exactly the definition of the parse matrix for E_G. Prediction, or filtering, is in the present case accomplished via the auxiliary set $\mathcal{C}(\xi)$:

1. $(S \rightarrow \cdot\alpha, 0) \in \mathcal{C}(\xi)$ for all rules $S \rightarrow \alpha$.

2. $(< A \rightarrow \cdot\alpha >, i) \in \mathcal{C}(\xi)$ if $A \rightarrow \alpha$ is a grammar rule and there is some $(< B \rightarrow \beta \cdot A\gamma >, i) \in \mathcal{C}(\xi)$.

3. $(A \rightarrow \alpha X \cdot \beta, j) \in \mathcal{C}(\xi)$ if, for some i, $(A \rightarrow \alpha \cdot X\beta, i) \in \mathcal{C}(\xi)$ and $(X, i, j) \in \mathcal{T}(\xi)$.

The first two clauses are like the prediction clauses 2a and 2b. The third is peculiar to the recursive ascent solution.

The recursive ascent Earley parser can be made more reminiscent of the standard Earley parser by applying Tomita's simulation method (section 9.9) to it. The result is an Earley parser that uses a graph-structured stack as its central data-structure, much like the standard implementation uses its parse matrix.

To summarize, the two parsers explore similar search spaces. Both are item-based algorithms for general context-free grammars. The sizes of the parse matrices that they produce are about the same. They accomplish the same task, producing a filtered parse matrix that is sufficient for the construction of a cubic parse forest. The major difference between the two parsers is superficial: it lies in the method used to explore the search space.

Exercises

1. Repeat section 7.3 with numbers instead of strings, i.e., with $\xi = x_1 ... x_n$ and with (i, j) denoting $x_{i+1} ... x_j$.

2. There is a small difference between a parse matrix with tuples (A, δ) and a parse matrix with triples (A, i, j). In general, one can be smaller than the other. Which one?

3. Revise algorithm (6.15) in the sense of (7.7) and (7.8) and find a matching parse forest construction.

4. Revise (6.14) in the same spirit. Write a commercial for the result.

5. As (7.8) is written, it would seem that in $\overline{[A \rightarrow \alpha \cdot \beta]}(X, i)$, for every $B \rightarrow \cdot X\mu \in pred(A \rightarrow \alpha \cdot \beta)$, the function $[B \rightarrow X \cdot \mu]$ is to be invoked twice. Rewrite (7.8) to show that one invocation is sufficient.

8

ATTRIBUTE GRAMMARS

Context-free grammars have limited power. Not all languages can be described by them completely. Programming languages, for instance, typically require that identifiers may be used only if they are properly declared. It is in general not possible to write a context-free grammar that describes only the programs without declaration errors.

Many extensions of context-free grammars, to make them more powerful, have been suggested. As we saw in chapter 3, context-free grammars may be generalized to general string rewriting systems by allowing rewriting rules with more than just one symbol on the left-hand side. The problem with such systems is that they lack a declarative interpretation: it is impossible to characterize the meaning of an individual rule independently from the other rules.

A very useful extension, in which declarativeness is preserved (in principle), is the attribute grammar formalism. This is a generalization of context-free grammars seen as abstract programs. In the abstract program interpretation, a context-free grammar defines a function $[X]$ for each symbol X. The generalization adds a function f_X to this, for each derivation starting from X. In some cases, $[X]$ and f_X can be combined into one function. The attribute grammar formalism allows one to define any language for which a recognition procedure exists.

Attribute grammars are being used in two rather different ways. In compiler technology, attributes are used mainly for semantic purposes: to compute the translation of a program to object code. In natural language processing, attributes typically have a purely syntactic function. These two kinds of

applications call for rather different attribute evaluation strategies. In a compiler, attributes are typically calculated after syntactically checking whether the program is correct according to a context-free grammar. In the linguistic application, attribute evaluation cannot be postponed so long. We present a special attribute grammar formalism that allows attribute evaluation at recognition time.

8.1 NOTATIONAL CONVENTIONS

An attribute grammar is based on a context-free grammar, called the *backbone* grammar. Each grammar symbol X of this grammar has a number of attributes. Its attributes come in two kinds: inherited ones, the type of which is denoted collectively as $inh(X)$, and synthesized ones, with type $syn(X)$.

We extend the mappings inh and syn to functions over the Lambek types T_V (see section 3.3), as follows:

$inh(\epsilon)$ is void: $inh(\epsilon) \times inh(a) = inh(a) \times inh(\epsilon) = inh(a)$
$inh(ab) = inh(a) \times inh(b)$
$inh(a\backslash b) = inh(b) \times syn(a)$
$inh(b/a) = syn(a) \times inh(b)$

and the same for syn:

$syn(\epsilon)$ is void
$syn(ab) = syn(a) \times syn(b)$
$syn(a\backslash b) = syn(b) \times inh(a)$
$syn(b/a) = inh(a) \times syn(b)$

It follows that inh and syn respect type identities, as they should, e.g.,

$inh_{\epsilon\backslash a} = inh_{a/\epsilon} = inh_a$
$inh(a/bc) = inh((a/c)/b)$
$inh(bc\backslash a) = inh(c\backslash(b\backslash a))$
$inh(b\backslash(a/c)) = inh((b\backslash a)/c)$

and similarly for syn.

Below we will use functions f_a that have domain $inh(a)$ and range $syn(a)$:

$$f_a : F_a, \text{ where } F_a = inh(a) \mapsto syn(a).$$

If we have a function f_a and $a \equiv b$ is a type identity, then f_b denotes the same function as f_a. That is, the subscript is defined modulo type identities.

Functions f_a are called *attribute functions*. We will assume that they are bunch-valued. Bunch-valued attribute functions accommodate ambiguous attribute evaluation, a requisite for natural-language processing.

A function of type F_a can be concatenated with a function of type F_b. The result is a function of type F_{ab} (inh_a,syn_a are variables of the types $inh(a)$,$syn(a)$, respectively):

$$\begin{aligned} &concat(f_a, f_b) = f_{ab}, \text{ where} \\ &f_{ab}(inh_a, inh_b) = \quad syn_a \leftarrow f_a(inh_a) \wedge syn_b \leftarrow f_b(inh_b) \ \rhd (syn_a, syn_b). \end{aligned}$$

It is often convenient to write $concat(f_a, f_b)$ with infix operator #:

$$concat(f_a, f_b) = f_a \# f_b.$$

A function of type F_a can be combined with a function of type $F_{a\backslash b}$ with the composition function $compose_{left}$, into a function of type F_b :

$$\begin{aligned} &compose_{left}(f_a, f_{a\backslash b}) = f_b, \text{ where} \\ &f_b(inh_b) = \quad (syn_b, inh_a) \leftarrow f_{a\backslash b}(inh_b, syn_a) \wedge syn_a \leftarrow f_a(inh_a) \ \rhd syn_b. \end{aligned}$$

A convenient infix notation for $compose_{left}$ is

$$compose_{left}(f_a, f_{a\backslash b}) = f_a \circ f_{a\backslash b}.$$

The definition of $compose_{left}$ is formally circular: syn_a depends on inh_a, which in turn depends on syn_a. If $inh(a), syn(a)$ are structured domains, however, not every aspect of syn_a necessarily depends on every aspect of

inh_a, so that the function produced by a $compose_{left}$ operation may well be effective. Below we will see examples of this.

Also a function $compose_{right}$ can be defined that combines a function of type $F_{b/a}$ and a function of type F_a into a function of type F_b. This is left as an exercise, since we will not need this composition function below.

The concatenation and composition functions obey a number of identities. We will use (remember: $f_{c\backslash(b\backslash a)} \equiv f_{bc\backslash a}$)

$$f_b \circ (f_c \circ f_{c\backslash(b\backslash a)}) = (f_b \# f_c) \circ f_{bc\backslash a}, \tag{8.1}$$

which says that the same function of type F_a may be obtained from functions f_b, f_c, and $f_{bc\backslash a}$ in two different but equivalent ways. One may either concatenate f_b and f_c and compose the result with $f_{bc\backslash a}$, or one may compose f_b with the composition of f_c and $f_{c\backslash(b\backslash a)}$.

Because the function types F_a, $F_{\epsilon a}$ and $F_{\epsilon\backslash a}$ are the same, the function f_ϵ, the unique function of type F_ϵ, is the identity for both $\circ$ and $\#$:

$$f_\epsilon \# f_a = f_a,$$
$$f_\epsilon \circ f_{\epsilon\backslash a} = f_{\epsilon\backslash a}.$$

Grammar symbols need not have attributes. If a has no attributes, then $f_a = f_\epsilon$.

8.2 ATTRIBUTE FUNCTIONS

The goal of an attribute grammar is to associate, for each derivation of a sentence, with each occurrence of X in that derivation, an attribute function f_X, of type F_X. In particular, with the whole derivation of the sentence, starting from start symbol S, a function f_S is associated. Usually, the start symbol has only synthesized attributes, in which case the invocation $f_S()$ gives the ultimate result of the attribute evaluation. (Although f_S has no arguments, we write brackets to indicate that we here refer, not to the function f_S, but to its invocation.) In the case of compilers, it may contain machine code and error messages. In other contexts, attributes may contain the answer

to a database query, or a representation of the logical structure of a natural language sentence. It is often useful to see f_X as a collection of functions: one for each attribute in its range $syn(X)$.

The functions f_X depend only on the part of the derivation that is dominated by X. In other words, in a parse tree each node labeled X gets a function f_X that depends only on the subtree of which this node is the root. Technically, f_X is some composition of atomic functions that are to be provided in the specification of the attribute grammar. If x is a terminal symbol, there cannot be a parse tree below it, and therefore f_x must be an atomic attribute function. If $X \in V_N$ dominates a derivation $X \xrightarrow{*} x_i ... x_j$, then f_X is composed of the atomic functions $f_{x_i} ... f_{x_j}$ and of atomic functions associated with the grammar rules that occur in the derivation. The atomic function associated with grammar rule $X \rightarrow \alpha$ is of type $F_{\alpha \backslash X}$. That is, it can be combined with a function of type F_α to give a function of type F_X. The function of type F_α can in its turn be obtained as the concatenation of functions f_Y, one for every Y in α.

So, the specification of an attribute grammar consists of backbone rules, with for each rule $X \rightarrow \alpha$ a function $f_{\alpha \backslash X}^{X \rightarrow \alpha}$ (if no confusion can arise, we write $f_{\alpha \backslash X}$) and for each terminal x a function f_x. The declarative nature of an attribute grammar depends on whether or not these atomic attribute functions can be understood declaratively. This in turn depends on a declarative understanding of the attributes: it must be clear what attributes "mean."

To define formally how attribute functions are associated with grammar symbols, we define *dressed derivations*. A dressed direct derivation is defined by

$$\begin{aligned} \Pi_1 \ (X, f_X) \ \Pi_2 \ \rightarrow \ \Pi_1 \ (Y_1, f_{Y_1}) ... (Y_k, f_{Y_k}) \ \Pi_2 \ \equiv \\ X \rightarrow Y_1 ... Y_k \wedge f_X = (f_{Y_1} \# ... \# f_{Y_k}) \circ f_{Y_1 ... Y_k \backslash X}^{X \rightarrow Y_1 ... Y_k}, \end{aligned} \tag{8.2}$$

where Π_1, Π_2 are arbitrary sequences of pairs (Y, f_Y). The transitive and reflexive closure is defined for dressed derivations, just as for normal derivations. Then, to each derivation $S \xrightarrow{*} x_1 ... x_n$ corresponds a dressed derivation

$$(S, f_S) \xrightarrow{*} (x_1, f_{x_1}) ... (x_n, f_{x_n}).$$

The attribute functions are constructed bottom-up: starting with the termi-

nals, which each have an associated atomic function, an attribute function is associated with each nonterminal by (8.2). As in (8.1), the concatenation operator # can be eliminated, in $f_X = (f_{Y_1}\#...\#f_{Y_k}) \circ f^{X \to Y_1...Y_k}_{Y_1...Y_k \backslash X}$. The function f_X may be computed alternatively as a sequence of compositions. First, with $f_{Y_k \backslash (Y_1...Y_{k-1} \backslash X)} = f^{X \to Y_1...Y_k}_{Y_1...Y_k \backslash X}$,

$$f_{Y_1...Y_{k-1} \backslash X} = f_{Y_k} \circ f_{Y_k \backslash (Y_1...Y_{k-1} \backslash X)}$$

is computed, followed by $(f_{Y_{k-1} \backslash (Y_1...Y_{k-2} \backslash X)} \equiv f_{Y_1...Y_{k-1} \backslash X})$

$$f_{Y_1...Y_{k-2} \backslash X} = f_{Y_{k-1}} \circ f_{Y_{k-1} \backslash (Y_1...Y_{k-2} \backslash X)},$$

and so on, until one arrives at $f_{\epsilon \backslash X} \equiv f_X$. Note that for ϵ-rules, this procedure is empty. The function for a nonterminal A that directly derives ϵ in a derivation is $f_A \equiv f_{\epsilon \backslash A}$.

8.3 EXAMPLE

Let us consider an example of an attribute grammar. Its backbone is

$$\begin{aligned} S &\to N \\ N &\to NN \\ N &\to 0 \\ N &\to 1 \end{aligned}$$

and generates all strings of zeros and ones. We wish to interpret such strings as binary numbers and to compute their (decimal) value. To that end, we assume that N has a pair of synthesized attributes of type integer. The first contains the *value* of the binary number generated from it. The second contains the *length* of the binary number. N has one inherited attribute, also an integer, that denotes the *position* of the string derived from N. This position is the number of bits that come after it in the string derived from S. Start symbol S has one synthesized attribute: the ultimate value. Terminals have no attributes in our grammar.

To each rule we associate an attribute function, and we use bunch notation to specify it. For $N \to 0$ and $N \to 1$, we write the following functions, which

produce N's synthesized attributes in terms of its inherited attribute:[1]

$$\begin{aligned} f_{0\backslash N}(pos) = &\quad val = 0 \wedge len = 1 \rhd (val, len), \\ f_{1\backslash N}(pos) = &\quad val = 2^{pos} \wedge len = 1 \rhd (val, len). \end{aligned}$$

These functions are quite self-evident: each "1" in the binary number at position *pos* stands for the value 2^{pos}. Both 0 and 1 have length 1.
The attribute function for $S \rightarrow N$ produces the synthesized attribute of S and the inherited attribute of N in terms of the synthesized attributes of N:

$$f_{N\backslash S}(val_1, len_1) = \quad pos = 0 \wedge val = val_1 \rhd (val, pos).$$

The binary string derived by this N is the one that is derived by S as well. The position of this string is thus zero. Moreover, the value attribute *val* of the binary string under S is the same as the value attribute val_1 of the same string under N.
The most complicated attribute function is the one for $N \rightarrow NN$:

$$\begin{aligned} &f_{NN\backslash N}(pos, val_1, len_1, val_2, len_2) = \\ &\quad val = val_1 + val_2 \wedge len = len_1 + len_2 \wedge pos_1 = pos + len_2 \wedge \\ &\qquad pos_2 = pos \rhd (val, len, pos_1, pos_2). \end{aligned}$$

The function is of type $inh(N) \times syn(N) \times syn(N) \mapsto syn(N) \times inh(N) \times inh(N)$, and the names of the variables should make it easy to interpret. The variables with indices 1,2 correspond to the first and second N on the right-hand side of $N \rightarrow NN$, respectively. The variables without indices stand for attributes of the left-hand side. The attribute function says that the position of the second N on the right-hand side of $N \rightarrow NN$ equals the position of the left-hand side N and that the position of the first N equals the position of the left-hand side N plus the length of the second N. Furthermore it adds the values and lengths of both right-hand side N's to become the value and length of the left-hand side N. This completes the attribute grammar. □

If the yield of an attribute function $f_{\alpha\backslash X}$ contains more than one attribute, we need a notational convention to indicate the subfunctions corresponding to individual attributes. If we need the attribute number i, this is computed

[1]In this example, we use an output variable for each output field, to enhance readability. Mathematically, output variables are superfluous and could be eliminated.

by $f_{\alpha\backslash X}.i$, which is assumed to "inherit" only those parameters of $f_{\alpha\backslash X}$ that it really uses. An example will illustrate this. The above function $f_{N\backslash S}$ produces two attributes and may be split into

$$f_{N\backslash S}.1(val_1) = \quad val = val_1 \triangleright val,$$

which calculates the first attribute and

$$f_{N\backslash S}.2() = \quad pos = 0 \triangleright pos,$$

which calculates the second. Note that the condition on the usefulness of parameters made the formal parameter *len* of $f_{N\backslash S}$ disappear, and that $f_{N\backslash S}.2$ has no parameters at all.

The above-mentioned circularity in the definition of $\circ$ can be avoided if attribute functions are such that not every output parameter depends on every input parameter. In the attribute functions of the above example grammar, for instance, most output values do not depend on all input parameters. Consider the computation of the composition of two attribute functions of our example grammar, $f_S = f_{1\backslash N} \circ f_{N\backslash S}$ ($f_{1\backslash N}$ has type F_N, because terminal "1" has no attributes):

$$\begin{aligned} &f_S() = val \\ &(val, pos) = f_{N\backslash S}(val_1, len) \\ &(val_1, len) = f_{1\backslash N}(pos) \end{aligned}$$

(Because the attribute functions of the example are single-valued, we use "=" instead of "$\leftarrow$" here.) This definition seems circular, but it is not. This becomes apparent if we split $f_{N\backslash S}$ into its subfunctions:

$$\begin{aligned} &pos = f_{N\backslash S}.2() \\ &(val_1, len) = f_{1\backslash N}(pos) \\ &val = f_{N\backslash S}.1(val_1) \\ &f_S() = val \end{aligned}$$

It is now clear that *pos* can be calculated. Inspection of $f_{N\backslash S}.2$ shows that its value is zero. If *pos*=0, then $f_{1\backslash N}(pos) = (val_1, len) = (1, 1)$. Then $f_{N\backslash S}.1(val_1) = val = 1$. Hence, $f_S() = val = 1$. □

Hence, the formal cycle in the definition of $\circ$ often disappears by taking into account the exact dependencies between attributes, or equivalently, by splitting the attribute functions into separate functions for each attribute. This does not always happen: circular attribute grammars do exist. However, even if the grammar is circular at the level of separate functions for each attribute, the internal structure of these separate functions may be such that functions composed in the above way are effective. Grammars for which this is the case are called *pseudo*-circular.

As an example, consider the following dressed derivation of the string 11: $(S, f_S) \rightarrow (N, f_N) \rightarrow (N, f_N^{(1)})(N, f_N^{(2)}) \rightarrow (1, f_1)(N, f_N^{(2)}) \rightarrow (1, f_1)(1, f_1)$, with rules from the above attribute grammar. Applying (8.2) we have

$$\begin{aligned}
&f_N^{(2)} = f_1 \circ f_{1\backslash N} = f_{1\backslash N} \\
&f_N^{(1)} = f_1 \circ f_{1\backslash N} = f_{1\backslash N} \\
&f_N = f_N^{(1)} \circ f_{N\backslash N}, \text{ where } f_{N\backslash N} = f_N^{(2)} \circ f_{NN\backslash N} \\
&f_S = f_N \circ f_{N\backslash S}
\end{aligned} \tag{8.3}$$

The first two functions equal $f_{1\backslash N}$ because terminal 1 has no attributes. Writing out the definition of $\circ$, one gets, after elimination of the functions $f_N^{(1)}$,$f_N^{(2)}$, f_N, and $f_{N\backslash N}$:

$$\begin{aligned}
&f_S() = val_1 \\
&(val_1, pos_1) = f_{N\backslash S}(val_2, len_1) \\
&(val_2, len_1, pos_2, pos_3) = f_{NN\backslash N}(pos_1, val_3, len_2, val_4, len_3) \\
&(val_3, len_2) = f_{1\backslash N}(pos_2) \\
&(val_4, len_3) = f_{1\backslash N}(pos_3)
\end{aligned} \tag{8.4}$$

Inspection of the definition of the attribute functions shows that

$$\begin{aligned}
&pos_1 = 0 \text{ (by } f_{N\backslash S}.2) \Rightarrow \\
&pos_3 = pos_1 = 0 \text{ (by } f_{NN\backslash N}.4) \Rightarrow \\
&val_4 = 1, len_3 = 1 \text{ (by } f_{1\backslash N}) \Rightarrow \\
&pos_2 = pos_1 + len_3 = 1 \text{ (by } f_{NN\backslash N}.3) \Rightarrow \\
&val_3 = 2, len_2 = 1 \text{ (by } f_{1\backslash N}) \Rightarrow \\
&val_2 = val_3 + val_4 = 3, len_1 = len_2 + len_3 = 2 \text{ (by } f_{NN\backslash N}) \Rightarrow \\
&val_1 = val_2 = 3 \text{ (by } f_{N\backslash S}) \Rightarrow \\
&f_S() = val_1 = 3
\end{aligned}$$

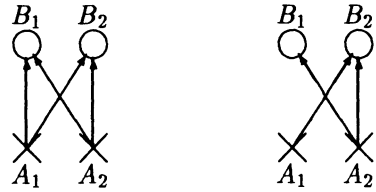

Figure 8.1 Atomic function types

Note that our example grammar is an ambiguous grammar. It associates various derivations with each binary number with length greater than two. The computation of $f_S()$ will, of course, yield the same value for every derivation of the same binary string.

8.4 FUNCTION GRAPHS

The above function manipulations can be illustrated nicely by drawing pictures. Such pictures are helpful in understanding the abstract definitions.

We use the following conventions. A function type is either atomic or composed. An atomic function, of type $A_1 \times A_2 \rightarrow B_1 \times B_2$, for instance, can be drawn as in figure 8.1. The picture "explains" that there are really two subfunctions that constitute a function of these types, each subfunction corresponding to one circle node. In the left picture there is one subfunction of type $A_1 \times A_2 \rightarrow B_1$ and another of type $A_1 \times A_2 \rightarrow B_2$. In the second picture there are subfunctions of the types $A_2 \rightarrow B_1$ and $A_1 \times A_2 \rightarrow B_2$. The second picture is not fully connected: the line from A_1 to B_1 is missing because there is no dependency between them.

Given a function of type $A_1 \times A_2 \rightarrow B$ and a function of, say, type $C \rightarrow A_1 \times D$, one may compose a function of type $C \times A_2 \rightarrow B \times D$. Such compositions of functions are of composed types. Figure 8.2 represents function composition pictorially, where the thick line connects nodes that must be identified. Function composition leads to function graphs, in which each node is associated with some type. Crossed circles, which in general have both incoming and outgoing arrows, are internal nodes. They are labeled with the types of intermediate results. Crosses, the input nodes, which have only outgoing arrows, constitute the domain of the composed function. Circles, the output nodes, which have no outgoing arrows, constitute the range of the function.

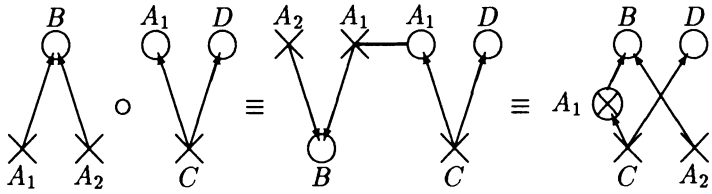

Figure 8.2 Function composition

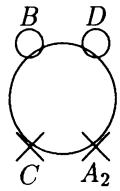

Figure 8.3 Abstract function type

A function that is composed according to some composition graph consists of the subfunctions of the atomic functions the graph is composed of, in the following way. To each internal and output node corresponds a subfunction that produces a value of the associated type. In the case of internal nodes, the produced value is to be transmitted along the outgoing arrows. The subfunction is applied to the values received along the incoming arrows. Some internal nodes may not have incoming arrows and are associated with nullary functions (constants). A function of a given composed type can be presented pictorially as the graph corresponding to the type, with each internal and output node labeled with the name of a subfunction.

Sometimes we want to abstract from the internal nodes. We do so by replacing these nodes by a big circle and writing only the input and output nodes, as in figure 8.3, which is an abstraction of the result type of figure 8.2. In a sense, such a picture denotes a type of function types, namely, all composed types with the same input and output nodes.

In the case of attribute grammars, the functions f_x for terminals x and the functions $f_{\alpha\setminus A}^{A\to\alpha}$ are the atomic functions: they are simply given and we do not care how they are implemented. For each derivation, we wish to construct a graph for f_S, with no input nodes.

A function f_a as described above may consist of many subfunctions, as many as there are attributes in $syn(a)$. In pictures we will always represent collections of attributes as pairs of attributes. For example, we draw an atomic

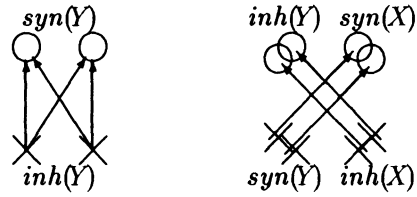

Figure 8.4 Atomic functions f_Y and $f_{Y\backslash X}$

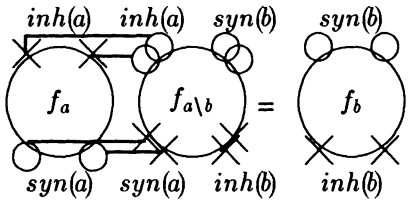

Figure 8.5 $f_b = f_a \circ f_{a\backslash b}$

function f_Y (in this case fully connected) as the left picture in figure 8.4. The second picture is an atomic function $f_{Y\backslash X}$ that is not fully connected.

If $f_b = f_a \circ f_{a\backslash b}$, function f_b is a kind of two-sided function composition of the functions f_a and $f_{a\backslash b}$. Pictorially it is represented by figure 8.5. The figure makes clear that $\circ$ may produce noneffective functions, corresponding to cyclic graphs, in the case of circular attribute grammars. For instance, if the graphs for f_Y and $f_{Y\backslash X}$ consist of atomic function graphs such as the first one of figure 8.1, i.e., if they are fully connected, the composed graph for f_X is surely cyclic. However, if lines are absent, as in the second picture of figure 8.1, there may well be no cycle. For example, if one substitutes the atomic graphs of figure 8.4 in figure 8.5, the result is cycle-free.

As a concrete example, consider the function f_S defined by (8.3). It can be represented by the graph of figure 8.6, which is a pictorial representation of (8.4). All internal nodes are represented in this picture only by the associated subfunction. There is one output node, drawn as the circle. From the picture it is immediately clear that $f_{N\backslash S}.2$ and $f_{1\backslash N}.2$ can be calculated, that to their results $f_{NN\backslash N}.4$ and $f_{NN\backslash N}.3$ can be applied, et cetera.

The above pictures not only illustrate the abstract formulae of the previous sections. They also show how to *implement* composed functions, for instance, in programming languages like Modula or C. Each node of a graph corresponds to one attribute and may be implemented as a record with fields that contain references to the nodes from which there are incoming arrows

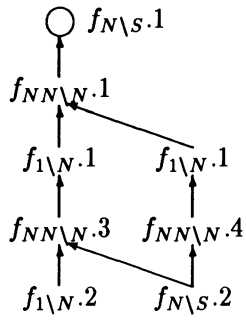

Figure 8.6 Graph of a function f_S

and a field that contains a function to compute the attribute. Execution of composed functions then goes as follows. Let there be a function *exec* that takes one argument, the node it is to execute, and delivers an attribute value. The computation of *exec*(N) consists of two steps. First it calls *exec*(N') for all nodes N' from which there is an arrow to N, in order to obtain the arguments for the attribute function associated with N. Subsequently, this function is applied to obtain the required attribute value. In general, the execution of node N may be called for many times. The reader should be able to guess the remedy against the loss of efficiency caused by repeatedly executing the same node: *exec* should be memoized. This way of executing function graphs only works for noncircular attribute grammars. In [Augusteijn,1990,1993] a refined version of *exec* is given that also works for pseudo-circular grammars.

8.5 ATTRIBUTE GRAMMAR PARSER

To summarize, the specification of an attribute grammar consists of

1. A context-free backbone grammar G
2. A function f_x for each terminal symbol x
3. A function $f_{\alpha\backslash A}$ (officially $f_{\alpha\backslash A}^{A\rightarrow\alpha}$) for each grammar rule $A \rightarrow \alpha$

For each derivation $A \rightarrow x_i ... x_j$ the above bottom-up composition procedure yields a function f_A in terms of the functions that correspond to the grammar rules and the terminals in the derivation.

The bottom-up procedure for constructing attribute functions can be combined simply with recognition procedures for the backbone grammar. In particular, the recognizers of chapter 6 (and chapter 9) can be adapted to produce attribute functions. Let us illustrate this with an adaptation of (6.2),(6.3):

$$
\begin{aligned}
&[A \to \alpha \cdot \beta](i) = \\
&\qquad \beta \overset{*}{\Rightarrow} x_{i+1}\gamma \ \triangleright \ \overline{[A \to \alpha \cdot \beta]}(x_{i+1}, i+1, f_{x_{i+1}}) \ | \\
&\qquad B \to \epsilon \wedge \beta \overset{*}{\Rightarrow} B\gamma \ \triangleright \ \overline{[A \to \alpha \cdot \beta]}(B, i, f_B^{B \to \epsilon}) \ | \\
&\qquad \beta = \epsilon \ \triangleright \ (i, f_{\alpha \backslash A}^{A \to \alpha}) \\
&\overline{[A \to \alpha \cdot \beta]}(X, i, f_X) = \\
&\qquad \beta = X\gamma \wedge (j, f_{\alpha X \backslash A}) \leftarrow [A \to \alpha X \cdot \gamma](i) \ \triangleright \ (j, f_X \circ f_{\alpha X \backslash A}) \ | \\
&\qquad \beta \overset{*}{\Rightarrow} C\gamma \wedge C \to X\delta \wedge (k, f_{X \backslash C}) \leftarrow [C \to X \cdot \delta](i) \\
&\qquad\qquad \triangleright \overline{[A \to \alpha \cdot \beta]}(C, k, f_X \circ f_{X \backslash C})
\end{aligned}
\tag{8.5}
$$

Remember that $f_{x_{i+1}}$ and $f_{\alpha \backslash A}^{A \to \alpha}$ are given functions and that function names without an argument list are not invoked. If

$$(n, f_{S'}) \leftarrow [S' \to \cdot S](0),$$

then the sentence $x_1 ... x_n$ was correct according to the backbone grammar. After the recognition, one may execute $f_{S'}()$ (which has no arguments because S' has no inherited attributes) to produce the required attributes, i.e., the synthesized attributes of S'. In a formula, using bunch notation, one has the following parser:

$$parser = \quad (n, f_{S'}) \leftarrow [S' \to \cdot S](0) \ \triangleright \ f_{S'}().$$

Inclusion of attribute functions in the LR recognizers that will be developed in the next chapter is similarly possible.

8.6 DIRECT ATTRIBUTE EVALUATION

In the above parser, attribute function evaluation is delayed until after recognition of the sentence. This is the only possibility for arbitrary attribute

grammars. However, if the grammar satisfies certain conditions, recognition and attribute evaluation can be interleaved. Below, we consider two special cases of this. First we notice that if there are only synthesized attributes, attribute evaluation can occur after the recognition of each grammar rule. In a sense, this is still a delayed evaluation. In some applications it is useful to be able to cut the attribute evaluation that belongs to a certain rule in parts, in such a way that attribute evaluation may be done incrementally while recognizing a grammar rule. In other applications, one wants direct attribute evaluation with inherited as well as synthesized attributes. The combination of both requirements (inherited and synthesized attributes; evaluation along with rule recognition) is provided by the OTF (On-The-Fly) attribute grammar formalism.

8.6.1 Synthesized attributes only

Suppose that there are only synthesized attributes. This means that the functions associated with terminals x have no arguments: they become a fixed collections syn_x of attributes. Similarly, a collection syn_B corresponds to $B \rightarrow \epsilon$. After the bottom-up recognition of a grammar rule, the attributes of the left-hand side can now be computed directly. The above recognizer may be changed to perform such direct attribute evaluation:

$$
\begin{aligned}
&[A \rightarrow \alpha \cdot \beta](i) = \\
&\qquad \beta \overset{*}{\Rightarrow} x_{i+1}\gamma \ \rhd \ \overline{[A \rightarrow \alpha \cdot \beta]}(x_{i+1}, i+1, syn_{x_{i+1}}) \ | \\
&\qquad \beta \overset{*}{\Rightarrow} B\gamma \wedge B \rightarrow \epsilon \ \rhd \ \overline{[A \rightarrow \alpha \cdot \beta]}(B, i, syn_B) \ | \\
&\qquad \beta = \epsilon \ \rhd \ (i, f_{\alpha \backslash A}^{A \rightarrow \alpha})
\end{aligned}
$$

$$
\begin{aligned}
&\overline{[A \rightarrow \alpha \cdot \beta]}(X, i, syn_X) = \\
&\qquad \beta = X\gamma \wedge (j, f_{\alpha X \backslash A}) \leftarrow [A \rightarrow \alpha X \cdot \gamma](i) \ \rhd \ (j, syn_X \circ f_{\alpha X \backslash A}) \ | \\
&\qquad \beta \overset{*}{\Rightarrow} C\gamma \wedge C \rightarrow X\delta \wedge (k, f_{X \backslash C}) \leftarrow [C \rightarrow X \cdot \delta](i) \\
&\qquad\qquad \rhd \overline{[A \rightarrow \alpha \cdot \beta]}(C, k, (syn_X \circ f_{X \backslash C})())
\end{aligned}
$$

Here one argument of $\circ$ is a collection of attributes rather than a function that delivers attributes; the reader will have no trouble formally adapting the definition of $\circ$ to deal with this. The attribute evaluation is performed by the invocation

$$(syn_X \circ f_{X \backslash C})().$$

Since attribute functions are bunch valued, this invocation may yield *null*. From the distributivity of functions over bunches, it follows that

$$\overline{[A \rightarrow \alpha \cdot \beta]}(C, k, (syn_X \circ f_{X \backslash C})()) \tag{8.6}$$

then yields *null* as well. If an attribute function produces *null*, the attribute evaluation is said to fail, and such failure may be used to cut a recognition path: the function invocation (8.6) need not be performed.

8.6.2 The OTF formalism

In the previous algorithm, first the grammar rule is recognized and subsequently the associated attribute evaluation is performed. Especially in natural-language applications, attribute evaluation often fails, and detection of such failure cannot happen soon enough. Whereas for linguistic applications synthesized attributes are often enough, for some applications one surely needs inherited ones, and it would be nice to have direct attribute evaluation of inherited attributes as well as synthesized ones. The following attribute-grammar formalism allows these things.

Attribute functions

OTF attribute grammars are as above, but for the following.

- The attribute function for rule $A \rightarrow \alpha$ is a function $f_{A/\alpha}$ of type $F_{A/\alpha}$. That is, the attribute functions corresponding to the elements of α will be combined with the attribute function for the grammar rule in a different order than was the case above. Unlike $f_{\alpha \backslash A}$, $f_{A/\alpha}$ is not an atomic function: it is a *composition* of $2|\alpha| + 2$ atomic functions.

The atomic functions that make up the attribute functions for grammar rules are as follows:

- For each grammar rule $A \rightarrow \alpha$, there are functions

  $$g_{A \rightarrow \cdot \alpha} : inh(A) \mapsto T(A \rightarrow \cdot \alpha)$$

  and

$$h_{A \to \alpha \cdot} : T(A \to \alpha \cdot) \mapsto syn(A)$$

where $T(A \to \cdot\alpha)$ and $T(A \to \alpha \cdot)$ are some types, to be defined as part of the attribute grammar.

- If $A \to \alpha X \beta$, functions

$$g_{A \to \alpha X \cdot \beta} : T(A \to \alpha \cdot X\beta) \times syn(X) \mapsto T(A \to \alpha X \cdot \beta)$$

and

$$h_{A \to \alpha \cdot X\beta} : T(A \to \alpha \cdot X\beta) \mapsto inh(X)$$

are given. Types $T(A \to \alpha \cdot \beta)$ must thus be given for all items $A \to \alpha \cdot \beta$.

Hence, if one has a grammar rule $A \to XYZ$, one must specify atomic functions

$$\begin{array}{l}
g_{A \to \cdot XYZ} : inh(A) \mapsto T(A \to \cdot XYZ) \\
h_{A \to \cdot XYZ} : T(A \to \cdot XYZ) \mapsto inh(X) \\
g_{A \to X \cdot YZ} : T(A \to \cdot XYZ) \times syn(X) \mapsto T(A \to X \cdot YZ) \\
h_{A \to X \cdot YZ} : T(A \to X \cdot YZ) \mapsto inh(Y) \\
g_{A \to XY \cdot Z} : T(A \to X \cdot YZ) \times syn(Y) \mapsto T(A \to XY \cdot Z) \\
h_{A \to XY \cdot Z} : T(A \to XY \cdot Z) \mapsto inh(Z) \\
g_{A \to XYZ \cdot} : T(A \to XY \cdot Z) \times syn(Z) \mapsto T(A \to XYZ \cdot) \\
h_{A \to XYZ \cdot} : T(A \to XYZ \cdot) \mapsto syn(A)
\end{array}$$

and all the types that appear herein. Function $f_{A/XYZ}$ is then defined as follows ($T_{A \to \alpha \cdot \beta}$ is of type $T(A \to \alpha \cdot \beta)$):

$$\begin{array}{l}
f_{A/XYZ}(syn_X, syn_Y, syn_Z, inh_A) = \\
\quad T_{A \to \cdot XYZ} \leftarrow g_{A \to \cdot XYZ}(inh_A) \wedge \\
\quad inh_X \leftarrow h_{A \to \cdot XYZ}(T_{A \to \cdot XYZ}) \wedge \\
\quad T_{A \to X \cdot YZ} \leftarrow g_{A \to X \cdot YZ}(T_{A \to \cdot XYZ}, syn_X) \wedge \\
\quad inh_Y \leftarrow h_{A \to X \cdot YZ}(T_{A \to X \cdot YZ}) \wedge \\
\quad T_{A \to XY \cdot Z} \leftarrow g_{A \to XY \cdot Z}(T_{A \to X \cdot YZ}, syn_Y) \wedge \\
\quad inh_Z \leftarrow h_{A \to XY \cdot Z}(T_{A \to XY \cdot Z}) \wedge \\
\quad T_{A \to XYZ \cdot} \leftarrow g_{A \to XYZ \cdot}(T_{A \to XY \cdot Z}, syn_Z) \wedge \\
\quad syn_A \leftarrow h_{A \to XYZ \cdot}(T_{A \to XYZ \cdot}) \rhd (inh_X, inh_Y, inh_Z, syn_A).
\end{array}$$

According to this definition, inh_X depends only on inh_A (via $T_{A \rightarrow \cdot XYZ}$), inh_Y on inh_A and syn_X, and inh_Z on inh_A, syn_X, and syn_Y. This means that the attribute functions corresponding to the grammar symbols on the right-hand side are needed in left-to-right order. If inh_A is known, inh_X is calculated, a function f_X is applied to give syn_X, inh_Y is calculated, a function f_Y is applied, et cetera.

It will be clear how this generalizes to arbitrary grammar rules. Note in particular that for ϵ-rules, $f_{A/\epsilon}(inh_A) = h_{A \rightarrow \cdot}(g_{A \rightarrow \cdot}(inh_A))$. It will also be clear that this way of implementing attribute functions allows the attribute evaluation to proceed stepwise, along with the recognition of a grammar rule.

Let us consider a concrete example with only synthesized attributes. Take the grammar rule

$$< np > \rightarrow < det > < adj > < noun >,$$

which says that a noun phrase can consist of a determiner followed by an adjective, followed by a noun. In natural languages, not all determiners, adjectives and nouns can be combined in this way. There has to be *feature agreement*: features like gender and number must be the same for all constituents of the noun phrase. Let us assume that the types $syn(< det >)$ and $syn(< adj >)$ consist of such features. Let us assume that the types are both equal to some type *agreement*. Let the attribute types $syn(< noun >)$ and $syn(< np >)$ be equal to $agreement \times rest$, where *rest* is a collection of non-agreement features. Then the following attribute functions can be associated with the grammar rule to filter out wrong combinations of determiners, adjectives, and nouns (A-variables and T-variables have type *agreement*; variable R has type *rest*):

$$\begin{aligned}
&g_{<np> \rightarrow <det> \cdot <adj> <noun>}(A_{<det>}) = A_{<det>} \\
&g_{<np> \rightarrow <det> <adj> \cdot <noun>}(T_{<np> \rightarrow <det> \cdot <adj> <noun>}, A_{<adj>}) = \\
&\quad (T_{<np> \rightarrow <det> \cdot <adj> <noun>} = A_{<adj>}) \rhd A_{<adj>} \\
&g_{<np> \rightarrow <det> <adj> <noun> \cdot}(T_{<np> \rightarrow <det> <adj> \cdot <noun>}, A_{<noun>}, R) = \\
&\quad (T_{<np> \rightarrow <det> <adj> \cdot <noun>} = A_{<noun>}) \rhd (A_{<noun>}, R)
\end{aligned}$$

No h-functions are needed, because there are no inherited attributes to calculate. For the same reason, there is no $g_{<np> \rightarrow \cdot <det> <adj> <noun>}$. The three g-functions determine, according to the above composition prescription, the intended attribute function:

$$f_{<np>/<det><adj><noun>}(A_{<det>}, A_{<adj>}, A_{<noun>}, R) = \\ A_{<det>} = A_{<adj>} = A_{<noun>} \rhd (A_{<noun>}, R).$$

Due to the composed nature of this attribute function, the grammar rule may be rejected halfway. After having seen a determiner and an adjective, $g_{<np>\rightarrow<det><adj>\cdot<noun>}$ may fail because there is no agreement between the two. □

Recursive descent

As was mentioned in chapter 4, one of the practical advantages of recursive descent parsing is that the functions can be parametrized with attributes. The idea is to augment a descent recognizer with direct attribute evaluation by letting $[X]$ play the rôle of f_X: each function $[X]$ gets an extra argument of type $inh(X)$, and it produces an extra value of type $syn(X)$.

Let us repeat the recursive descent function (4.3) for nonterminal A,

$$[A](i) = \quad A \rightarrow \alpha \rhd [\alpha](i),$$

and the function for terminal symbols,

$$[x](i) = \quad x = x_{i+1} \rhd i + 1.$$

In order to be able to add attribute evaluation, we rewrite $[A]$:

$$\begin{aligned} &[A](i_0) = \\ &\qquad A \rightarrow X_1 ... X_k \rhd \\ &\qquad i_1 \leftarrow [X_1](i_0) \rhd \\ &\qquad i_2 \leftarrow [X_2](i_1) \rhd \\ &\qquad \quad . \\ &\qquad \quad . \\ &\qquad i_k \leftarrow [X_k](i_{k-1}) \rhd i_k. \end{aligned} \tag{8.7}$$

For terminals x, it is very simple to add attribute evaluation to $[x]$:

$$[x](i, inh_x) = \quad x = x_{i+1} \wedge syn_x \leftarrow f_x(inh_x) \rhd (i + 1, syn_x). \tag{8.8}$$

Due to the composed nature of the attribute functions associated with the grammar rules, the attribute evaluation can be distributed all over (8.7). Before each call $[X_m](i_{m-1})$, an h-function calculates the inherited attributes of X_m; after the call a g-function is computed. The result appears as follows:

$$
\begin{aligned}
&[A](i_0, inh_A) = \\
&\qquad A \rightarrow X_1 ... X_k \wedge \\
&\qquad\quad T_{A \rightarrow \cdot X_1 ... X_k} \leftarrow g_{A \rightarrow \cdot X_1 ... X_k}(inh_A) \wedge \\
&\qquad\quad inh_{X_1} \leftarrow h_{A \rightarrow \cdot X_1 X_2 ... X_k}(T_{A \rightarrow \cdot X_1 ... X_k}) \rhd \\
&\qquad (i_1, syn_{X_1}) \leftarrow [X_1](i_0, inh_{X_1}) \wedge \\
&\qquad\quad T_{A \rightarrow X_1 \cdot X_2 ... X_k} \leftarrow g_{A \rightarrow X_1 \cdot X_2 ... X_k}(T_{A \rightarrow \cdot X_1 ... X_k}, syn_{X_1}) \wedge \\
&\qquad\quad inh_{X_2} \leftarrow h_{A \rightarrow X_1 \cdot X_2 ... X_k}(T_{A \rightarrow X_1 \cdot X_2 ... X_k}) \rhd \\
&\qquad (i_2, syn_{X_2}) \leftarrow [X_2](i_1, inh_{X_2}) \wedge \\
&\qquad\quad . \\
&\qquad\quad . \\
&\qquad\quad . \\
&\qquad (i_k, syn_{X_k}) \leftarrow [X_k](i_{k-1}, inh_{X_k}) \wedge \\
&\qquad\quad T_{A \rightarrow X_1 ... X_k \cdot} \leftarrow g_{A \rightarrow X_1 ... X_k \cdot}(T_{A \rightarrow X_1 ... X_{k-1} \cdot X_k}, syn_{X_k}) \wedge \\
&\qquad\quad syn_A \leftarrow h_{A \rightarrow X_1 ... X_k \cdot}(T_{A \rightarrow X_1 ... X_k \cdot}) \rhd (i_k, syn_A).
\end{aligned}
\tag{8.9}
$$

Functions (8.8) and (8.9) constitute a parser for OTF attribute grammars. Note how heavily intertwined attribute evaluation and recognition are. All attribute functions are assumed to be bunch valued. At each stage, attribute evaluation may fail (produce *null*) and stop the further exploration of the grammar rule. Search paths are cut off in this way, and efficiency is enhanced compared to delayed attribute evaluation. It may even be the case that failures of the attribute evaluation make the recursive descent parser effective if the backbone is left recursive.

Recursive ascent

In linguistic applications, recursive descent parsing is not a logical choice. The reason is that natural language grammars usually are full of left recursive derivations. Therefore, let us see whether bottom-up recognizers can also be augmented with OTF attribute evaluation. Define the following sets:

$$
\begin{aligned}
&\overline{(A \rightarrow \alpha \cdot \beta, T_{A \rightarrow \alpha \cdot \beta})} = \{(A \rightarrow \alpha \cdot \beta, T_{A \rightarrow \alpha \cdot \beta})\} \cup \\
&\quad \{(C \rightarrow \cdot \delta, T_{C \rightarrow \cdot \delta}) | (B \rightarrow \mu \cdot C\nu, T_{B \rightarrow \mu \cdot C\nu}) \in \overline{(A \rightarrow \alpha \cdot \beta, T_{A \rightarrow \alpha \cdot \beta})} \wedge \\
&\quad C \rightarrow \delta \wedge inh_C \leftarrow h_{B \rightarrow \mu \cdot C\nu}(T_{B \rightarrow \mu \cdot C\nu}) \wedge T_{C \rightarrow \cdot \delta} \leftarrow g_{C \rightarrow \cdot \delta}(inh_C)\}
\end{aligned}
$$

This is a recursive definition of a kind that we encountered before. Again we intend to define the smallest sets that satisfy the recursion. The following is an ascent recognizer with direct evaluation of OTF attributes:

$$
\begin{array}{l}
[A \to \alpha \cdot \beta](i, T_{A\to\alpha\cdot\beta}) = \\
\quad (C \to \nu \cdot x_{i+1}\delta, T_{C\to\nu\cdot x_{i+1}\delta}) \in \overline{(A \to \alpha \cdot \beta, T_{A\to\alpha\cdot\beta})} \wedge \\
\quad\quad inh_{x_{i+1}} \leftarrow h_{C\to\nu\cdot x_{i+1}\delta}(T_{C\to\nu\cdot x_{i+1}\delta}) \wedge \\
\quad\quad syn_{x_{i+1}} \leftarrow f_{x_{i+1}}(inh_{x_{i+1}}) \\
\quad\quad \triangleright \overline{[A \to \alpha \cdot \beta]}(x_{i+1}, i+1, inh_{x_{i+1}}, syn_{x_{i+1}}, T_{A\to\alpha\cdot\beta}) \mid \\
\quad (C \to \nu \cdot B\delta, T_{C\to\nu\cdot B\delta}) \in \overline{(A \to \alpha \cdot \beta, T_{A\to\alpha\cdot\beta})} \wedge \\
\quad\quad B \to \epsilon \wedge \\
\quad\quad inh_B \leftarrow h_{C\to\nu\cdot B\delta}(T_{C\to\nu\cdot B\delta}) \wedge \\
\quad\quad syn_B \leftarrow h_{B\to\cdot}(g_{B\to\cdot}(inh_B)) \\
\quad\quad \triangleright \overline{[A \to \alpha \cdot \beta]}(B, i, inh_B, syn_B, T_{A\to\alpha\cdot\beta}) \mid \\
\quad \beta = \epsilon \wedge syn_A = h_{A\to\alpha\cdot}(T_{A\to\alpha\cdot\beta}) \triangleright (i, syn_A)
\end{array}
\tag{8.10}
$$

$$
\begin{array}{l}
\overline{[A \to \alpha \cdot \beta]}(X, i, inh_X, syn_X, T_{A\to\alpha\cdot\beta}) = \\
\quad \beta = X\gamma \wedge \\
\quad\quad inh_X \leftarrow h_{A\to\alpha\cdot X\gamma}(T_{A\to\alpha\cdot\beta}) \wedge \\
\quad\quad T_{A\to\alpha X\cdot\gamma} \leftarrow g_{A\to\alpha X\cdot\gamma}(T_{A\to\alpha\cdot\beta}, syn_X) \\
\quad\quad \triangleright [A \to \alpha X \cdot \gamma](i, T_{A\to\alpha X\cdot\gamma}) \mid \\
\quad (D \to \nu \cdot C\delta, T_{D\to\nu\cdot C\delta}) \in \overline{(A \to \alpha \cdot \beta, T_{A\to\alpha\cdot\beta})} \wedge \\
\quad\quad C \to X\delta \wedge \\
\quad\quad inh_C \leftarrow h_{D\to\nu\cdot C\delta}(T_{D\to\nu\cdot C\delta}) \wedge \\
\quad\quad T_{C\to\cdot X\delta} \leftarrow g_{C\to\cdot X\delta}(inh_C) \wedge \\
\quad\quad inh_X \leftarrow h_{C\to\cdot X\delta}(T_{C\to\cdot X\delta}) \wedge \\
\quad\quad T_{C\to X\cdot\delta} \leftarrow g_{C\to X\cdot\delta}(T_{C\to\cdot X\delta}, syn_X) \wedge \\
\quad\quad (k, syn_C) \leftarrow [C \to X \cdot \delta](i, T_{C\to X\cdot\delta}) \\
\quad\quad \triangleright \overline{[A \to \alpha \cdot \beta]}(C, k, inh_C, syn_C, T_{A\to\alpha\cdot\beta})
\end{array}
\tag{8.11}
$$

If all attribute types have bounded size, memoization of this adaptation of (6.2),(6.3) still yields a cubic algorithm. The sets $\overline{(A \to \alpha \cdot \beta, T_{A\to\alpha\cdot\beta})}$ encode the leftmost symbol derivations. They should be treated in a lazy way: only the sets that are needed should be computed. And, of course, they should be computed only once. Note again, that failure of attribute evaluation will cut off search paths. For instance, if $g_{A\to\alpha X\cdot\gamma}(T_{A\to\alpha\cdot\beta}, syn_X)$ fails, the invocation $[A \to \alpha X \cdot \gamma](i, T_{A\to\alpha X\cdot\gamma})$ is not performed.

Often, it is convenient to assume that the T-types depend only on the grammar rule, not on the position of the dot. That is,

$$T(A \rightarrow \alpha \cdot \beta) = T(A \rightarrow \gamma \cdot \delta)$$

if $\alpha\beta = \gamma\delta$. In particular, with such an assumption, it is not difficult to generalize the above to EBNF grammars by defining g and h functions for each EBNF item. This is left as an exercise.

Direct attribute evaluation can be combined with delayed evaluation by specifying for each grammar rule $A \rightarrow \alpha$ two attribute functions, $f_{A/\alpha}$ and $f_{\alpha\backslash A}$, in principle for two disjunct collections of attributes. The recognizer (8.10),(8.11) can be augmented to produce attribute functions that are evaluated afterwards. We leave this also as an exercise. In practical systems, it is natural to allow the delayed attribute evaluation to depend on attributes that have been already evaluated directly. It is arguable that it is really an implementation issue whether or not a particular attribute can or should be evaluated on-the-fly, as long as termination of recognition does not depend on it. If one takes such a view, one should automatically dissect the maximal OTF part of a general attribute grammar, in order to gain maximal recognition efficiency.

Just as direct attribute evaluation may make a recursive descent parser effective even though the backbone is left recursive, the recursive ascent parser may be effective even though the backbone is cyclic. This situation is not at all exceptional; it occurs quite often in natural language parsing. Cyclic context-free grammars as such are useless. In practice, either they contain an error, or the cycles are broken by attribute evaluation. This is why the problems with cyclic grammars are not a serious drawback to recursive ascent parsing.

Exercises

1. Define a function $compose_{right}$ that combines a function of type $F_{b/a}$ and a function of type F_a into a function of type F_b.

2. Verify that the binary string 110 can be derived in two ways by the backbone of the example attribute grammar, and that for both derivations $f_S() = 6$.

3. Write an attribute grammar that computes the values of binary strings on the basis of a backbone grammar that is not ambiguous.

4. Find a version of (8.5) in which function compositions happen in the functions $[A \rightarrow \alpha \cdot \beta]$ and not in $\overline{[A \rightarrow \alpha \cdot \beta]}$.

5. Simplify (8.10),(8.11) for the special case in which there are only synthesized attributes.

6. Generalize (8.10),(8.11) for EBNF grammars, assuming that T-types depend only on the grammar rule (see section 6.5).

7. Combine the direct attribute evaluation of (8.10),(8.11) with the composition of attribute functions for delayed attribute evaluation, as in (8.5).

9

LR PARSERS

There are purposes for which the recursive ascent Earley-like parsers are not suitable, because they are not deterministic for most grammars. In (6.3), for instance, at the same position in the sentence, various functions $[C \rightarrow X \cdot \delta]$ can be called for the same X. If the grammar at hand does not allow a deterministic parser, this is not a disqualification; this is typical for grammars for natural languages. However, grammars for programming languages usually do allow deterministic parsing. Moreover, in this application of grammars, avoidance of nondeterminism pays off because the efficiency of compilers is a very critical issue. Because a program can be very large, it is particularly convenient if the recognizer for a programming language consumes as little space as possible.

It is possible to combine various function calls $[C \rightarrow X \cdot \delta]$ for the same X into one, and this leads to LR parsing. LR parsers can be built with or without look ahead. We will present the variants that go under the names LR(0) and LALR(1). Out of all grammars that allow deterministic analysis, only very few cannot be processed deterministically with an LALR(1) parser.

9.1 LR(0) RECOGNIZER

The mechanism for reducing nondeterminism is the merging of functions corresponding to a number of competing items into one function. Let the set of all items of G be given by I_G. Subsets of I_G are called states, and we use q to denote an arbitrary state. We associate with each state q a function $[q]$ that meets the specification

$$[q](i) = \quad A \to \alpha \cdot \beta \in q \wedge \beta \xrightarrow{*} x_{i+1} \ldots x_j \ \triangleright (A \to \alpha \cdot \beta, j). \tag{9.1}$$

The function produces pairs as did the corresponding revised Earley function (7.3). For each item in q, function (9.1) reports which chunks of the sentence that start at position $i+1$ can be derived from the part after the dot. If we define the initial state $q_0 = \{S' \to \cdot S\}$, we have that $S \xrightarrow{*} x_1 \ldots x_n$ is equivalent to $(S' \to \cdot S, n) \leftarrow [q_0](0)$.

To be able to construct a recursive ascent implementation of $[q]$, we need some auxiliary functions. Let $ini(q)$ be the set of predict items for state q, derived from q as the smallest solution of

$$ini(q) = \{B \to \cdot\nu | B \to \nu \wedge A \to \alpha \cdot B\beta \in q \cup ini(q)\}.$$

An alternative nonrecursive definition, similar to the definition of (7.5), is

$$ini(q) = \{B \to \cdot\nu | B \to \nu \wedge A \to \alpha \cdot \beta \in q \wedge \beta \overset{*}{\Rightarrow} B\gamma\}. \tag{9.2}$$

The so-called transition function is defined by

$$goto(q, X) = \{A \to \alpha X \cdot \beta | A \to \alpha \cdot X\beta \in (q \cup ini(q))\}. \tag{9.3}$$

Consider again the grammar $S \to xSy$, $S \to xy$. For this grammar there are six states accessible via *goto* transitions, starting from q_0. They are depicted in figure 9.1, together with the *goto* transitions. Above the horizontal line in a box named q, the items of the state q appear; below the line appear the predict items of $ini(q)$. If there is an arrow labeled X from state q to state p, then $p = goto(q, X)$. These six states and the transitions between them are together called the LR(0) automaton of the grammar.

A recursive ascent recognizer is obtained by relating to each state q not only the above $[q]$ but also a function $\overline{[q]}$ with the specification

$$\begin{aligned} &\overline{[q]}(X, i) = \\ &\qquad A \to \alpha \cdot \beta \in q \wedge \beta \overset{*}{\Rightarrow} X\gamma \wedge \gamma \xrightarrow{*} x_{i+1} \ldots x_j \ \triangleright (A \to \alpha \cdot \beta, j) \end{aligned} \tag{9.4}$$

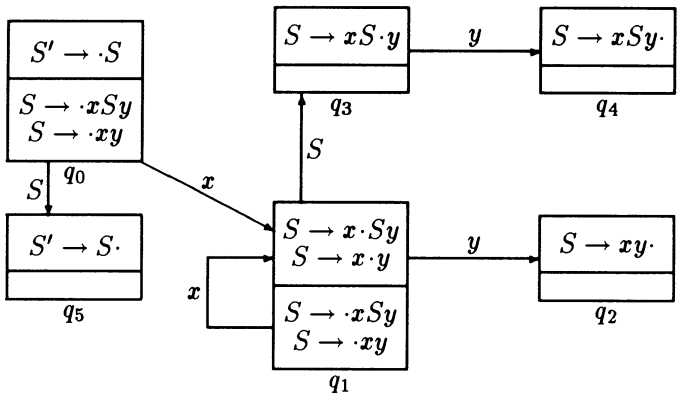

Figure 9.1 LR(0) automaton for $S \to xSy$, $S \to xy$

Assuming, as before, that $x_{n+1} \notin V$, it is impossible that $A \to \alpha \cdot \beta \in q \wedge \beta \overset{*}{\Rightarrow} x_{n+1}\gamma$; hence $\overline{[q]}(x_{n+1}, n+1) = null$. For $i \leq n$, the above functions are recursively implemented by

$$
\begin{aligned}
&[q](i) = \\
&\qquad \overline{[q]}(x_{i+1}, i+1) \mid \\
&\qquad B \to \cdot \in ini(q) \rhd \overline{[q]}(B, i) \mid \\
&\qquad A \to \alpha \cdot \in q \rhd (A \to \alpha \cdot, i) \\
&\overline{[q]}(X, i) = \\
&\quad A \to \alpha \cdot X\gamma \in q \wedge (A \to \alpha X \cdot \gamma, j) \leftarrow [goto(q, X)](i) \\
&\qquad \rhd (A \to \alpha \cdot X\gamma, j) \mid \\
&\quad C \to \cdot X\delta \in ini(q) \ \wedge (C \to X \cdot \delta, k) \leftarrow [goto(q, X)](i) \rhd \overline{[q]}(C, k)
\end{aligned}
\tag{9.5}
$$

The proof is isomorphic to the proof of the first recognizer in chapter 6. First we use (6.4) again, to prove that

$$
\begin{aligned}
&[q](i) = \\
&\qquad \overline{[q]}(x_{i+1}, i+1) \mid \\
&\qquad B \to \epsilon \rhd \overline{[q]}(B, i) \mid \\
&\qquad A \to \alpha \cdot \in q \rhd (A \to \alpha \cdot, i)
\end{aligned}
$$

This is equivalent to the earlier version because we may replace the clause $B \to \epsilon$ by $B \to \cdot \in ini(q)$. Indeed, if state q has item $A \to \alpha \cdot \beta$ and if there is a leftmost-symbol derivation $\beta \overset{*}{\Rightarrow} B\gamma$, then all items $B \to \cdot\mu$ are included in $ini(q)$.

For establishing the correctness of $\overline{[q]}$, we again use (6.5) to prove

$$\begin{aligned}
&\overline{[q]}(X,i) = \\
&\qquad A \to \alpha \cdot X\gamma \in q \wedge \gamma \overset{*}{\to} x_{i+1}...x_j \ \rhd (A \to \alpha \cdot X\gamma, j) \mid \\
&\qquad A \to \alpha \cdot \beta \in q \wedge \beta \overset{*}{\Rightarrow} C\gamma \wedge C \to X\delta \wedge \delta \overset{*}{\to} x_{i+1}...x_k \wedge \\
&\qquad\ \ \gamma \overset{*}{\to} x_{k+1}...x_j \ \rhd (A \to \alpha \cdot \beta, j).
\end{aligned}$$

By the definition of $goto$, if $A \to \alpha \cdot X\gamma \in q$ then $A \to \alpha X \cdot \gamma \in goto(q,X)$. Hence, with the specification of $[goto(q,X)]$ and $\overline{[q]}(C,k)$,

$$\begin{aligned}
&\overline{[q]}(X,i) = \\
&\quad A \to \alpha \cdot X\gamma \in q \wedge (A \to \alpha X \cdot \gamma, j) \leftarrow [goto(q,X)](i) \\
&\qquad\quad \rhd (A \to \alpha \cdot X\gamma, j) \mid \\
&\quad C \to \cdot X\delta \in ini(q) \ \wedge (C \to X \cdot \delta, k) \leftarrow [goto(q,X)](i) \ \rhd \overline{[q]}(C,k),
\end{aligned}$$

where we used the facts that $\beta \overset{*}{\Rightarrow} C\gamma$ implies that all items $C \to \cdot\mu$ are in $ini(q)$, and the existence of $C \to \cdot X\delta$ in $ini(q)$ implies $C \to X \cdot \delta \in goto(q,X)$. □

The recognition functions may be cast in a form that will be convenient below. Define the auxiliary functions

$$\begin{aligned}
&pop(A \to \alpha X \cdot \beta) = A \to \alpha \cdot X\beta \\
&lhs(A \to \alpha \cdot \beta) = A \\
&final(A \to \alpha \cdot \beta) = (|\beta| = 0).
\end{aligned}$$

Then it is easy to confirm that $[q]$ and $\overline{[q]}$ may be written as

$$
\begin{aligned}
[q](i) = & \\
& \overline{[q]}(x_{i+1}, i+1) \mid \\
& B \rightarrow \cdot \in ini(q) \rhd \overline{[q]}(B, i) \mid \\
& I \in q \wedge final(I) \rhd (I, i) \\
\overline{[q]}(X, i) = & \\
& (I, j) \leftarrow [goto(q, X)](i) \wedge pop(I) \in q \rhd (pop(I), j) \mid \\
& (J, k) \leftarrow [goto(q, X)](i) \wedge pop(J) \in ini(q) \rhd \overline{[q]}(lhs(J), k)
\end{aligned}
\tag{9.6}
$$

Note that $goto(q, x_i)$ may be $\emptyset$ (e.g., if q is the q_0 of figure 9.1 and $x_i = y$) and that formally $[\emptyset](i)$ may be invoked. However, $[\emptyset](i) = null$ for all $i \geq 0$, so $[\emptyset]$ need never be called in practice.

Example

To give an explicit example, consider the state q_1 in figure 9.1. Its functions are

$$[q_1](i) = \overline{[q_1]}(x_{i+1}, i+1)$$

and

```
[q1](X,i) =  result:=null;
             if X = y then (* q2 = goto(q1,y) *)
                for all (S → xy·,k) ← [q2](i) (* of course k = i *) do
                   result:=result | (S → x · y,k)
                od
             else if X = S then (* q3 = goto(q1,S) *)
                for all (S → xS · y,k) ← [q3](i) do
                result:=result | (S → x · Sy,k)
                od
             else if X = x then (* q1 = goto(q1,x) *)
                for all (S → x · α,k) ← [q1](i) (* α is Sy or y *) do
                   result:=result | [q1](S,k)
                od
             fi;
             return result;
```

Naturally, the last function can be simplified a lot, but it was written in this way to show the general schema.

9.2 THE DETERMINISTIC CASE

For the LR(0) recognizer (9.5) to be deterministic, no function $[q]$ may return more than one value. There are two possible causes for states to produce an ambiguous result:

1. A state q has a *reduce-reduce* conflict: it has more than one item of the form $A \to \alpha\cdot$. Clearly, in this case the result of $[q](i)$ involves each such item.
2. A state q has a *shift-reduce* conflict: it has an item $A \to \alpha\cdot$, and there is a transition to another state p labeled by a terminal x, i.e., $p = goto(q, x)$. Then $[q](i)$ produces $(A \to \alpha\cdot, i)$, but if $x_{i+1} = x$ it may produce other pairs as well.

It is not difficult to see that if no LR(0) states have such conflicts, all functions produce at most one result and (9.5) is deterministic. Grammars for which this is the case are called LR(0).

For LR(0) grammars, the functions (9.6) for nonempty q may be rephrased in an imperative way as

$[q](i) =$ if, for some I, $I \in q \wedge final(I)$ then return (I, i)
else if, for some B, $B \to \cdot \in ini(q)$ then return $\overline{[q]}(B, i)$
else if $i < n$ then return $\overline{[q]}(x_{i+1}, i+1)$
else return *null*
fi

$\overline{[q]}(X, i) =$ if $[goto(q, X)](i) =$ *null* then return *null*
else let $(I, j) = [goto(q, X)](i)$. Then:
if $pop(I) \in q$ then return $(pop(I), j)$
else return $\overline{[q]}(lhs(I), j)$
fi
fi

Functions like these are readily implemented in a programming language like C. To improve efficiency, some additional changes are in order, like the ones for LL(1) recognition in chapter 4. Each function can be replaced by a procedure that, instead of returning a function result, assigns the result to global variables. A function result may be represented by two variables: an item R and an integer i. If a function would have resulted in (I, j), the global variables are set to $R = I$ and $i = j$. As in chapter 4, we use *sym* for x_{i+1} and *nextsym* for updating integer i, and if the function value would have been *null* the procedure *error* is called. So, $[\emptyset] = error$. All function arguments are superfluous now (we write $lhs(R)$ instead of argument X of $\overline{[q]}$). All this leads to the procedures

$[q]$: if, for some I, $I \in q \wedge final(I)$ then $R := I$
else if, for some B, $B \rightarrow \cdot \in ini(q)$ then $R := B \rightarrow \cdot$; $\overline{[q]}$
else if "not end of input" then $R := sym \rightarrow sym\cdot$; $nextsym$; $\overline{[q]}$
else *error*
fi

$\overline{[q]}$: $[goto(q, lhs(R))]$;
if $pop(R) \in q$ then $R := pop(R)$
else $\overline{[q]}$
fi

formally adding $x \rightarrow x\cdot$ to the set of items, for all $x \in V_T$. Note that these procedures do not depend on the details of the right-hand side of R. Only the number of symbols before the dot is relevant for the test "$pop(R) \in q$." Therefore, R can be replaced by two variables $X \in V$ and an integer l, making the following substitutions in the previous procedures:

$$\begin{array}{lcl@{\qquad}lcl} R := A \rightarrow \alpha\cdot & \Longrightarrow & X := A; l := |\alpha| & R := pop(R) & \Longrightarrow & l := l - 1 \\ pop(R) \in q & \Longrightarrow & l \neq 1 \vee X = S' & lhs(R) & \Longrightarrow & X \end{array}$$

One further optimization is the replacement of the call of the tail-recursive procedure $\overline{[q]}$ by an iterative loop:

$[q]$: if $A \rightarrow \alpha\cdot \in q$, then $X := A$; $l := |\alpha|$
else if $B \rightarrow \cdot \in ini(q)$ then $X := B$; $l := 0$; $\overline{[q]}$
else if "not end of input" then $X := sym$; $l := 1$; $nextsym$; $\overline{[q]}$
else *error*
fi

```
[q̄] :  [goto(q, X)];
       while l = 1 ∧ X ≠ S' do
          [goto(q, X)]
       od;
       l := l − 1;
```

The conditions $A \to \alpha\cdot \in q$ and $B \to \cdot \in ini(q)$ must not be performed at run-time. For each individual function $[q]$, these conditions are false or true. When true, the corresponding actions are to performed without any guard. When false, the actions do not appear in $[q]$. That is, if q is a reduce state and $A \to \alpha\cdot \in q$, then

```
[q] :  X := A; l := |α|.
```

If q is a reduce state and $B \to \cdot \in ini(q)$, then

```
[q] :  X := B;
       [goto(q, X)];
       while l = 1 ∧ X ≠ S' do
          [goto(q, X)]
       od;
       l := l − 1;
```

Lastly, if q is a shift state, then

```
[q] :  if "end of input" then error else
          X := sym; nextsym;
          [goto(q, X)];
          while l = 1 ∧ X ≠ S' do
             [goto(q, X)]
          od;
          l := l − 1;
       fi
```

The actions $l := 0$ and $l := 1$ were discarded because after these assignments no reference to l is made until after it has gotten a new value.

9.3 IMPLEMENTATION WITH STACKS

In the deterministic case, LR(0) parsing can be implemented in a nonrecursive way. In (9.5), two stacks can be discerned:

1. The recursion stack that implements recursive function calls. This stack is hidden in the recursive formulation by the programming language that supports recursive functions. It can also be handled explicitly, and then it is called a *parse stack*, in our application. The parse stack is a stack of names of states.

2. The input stack. The integer numbers in (9.5) refer to the input, i.e., i stands for $x_{i+1}...x_n$. It is possible to see such a string as a stack, with x_{i+1} at the top and x_n at the bottom. We take the convention to write a stack as a string of symbols separated by semicolons. The leftmost symbol of the string is the top. Rightmost is the symbol $\perp$. So i stands for $(x_{i+1}; ...; x_n; \perp)$. The pair $(A \rightarrow \alpha \cdot \beta, i)$ can then be seen as the stack $(A \rightarrow \alpha \cdot \beta; x_{i+1}; ...; x_n; \perp)$. So the item is at the top, above the input symbols. Likewise, the argument (X, i) of $\overline{[q]}$ is the stack $(X; x_{i+1}; ...; x_n; \perp)$.

In this view, $[q]$ and $\overline{[q]}$ each have one argument, the input stack. Also, their output is an input stack. Hence, both functions may seen as input stack transformers. If the grammar is LR(0), the input stack can be a global data structure that is continually changed by these stack transformers. The parse stack is changed by invocation and termination of functions.

Now, let us see what happens in (9.5), in terms of the input and parse stacks. The initial invocation is $[q_0](0)$, which means that while the input stack is $(x_1; ...; x_n; \perp)$, q_0 is pushed on the empty parse stack, which then becomes $(q_0; \perp)$. In general, if some state symbol q is pushed on the parse stack while the input stack is $(x_{i+1}; ...; x_n; \perp)$, this corresponds to a call of $[q](i)$. Let us look at the three cases handled in $[q](i)$, in (9.5):

1. The call of $\overline{[q]}(x_{i+1}, i+1)$. Nothing happens here. The same q remains at the top of the stack; also the input stack does not change since i and $(x_{i+1}, i+1)$ correspond to the same input stack: $(x_{i+1}; ...; x_n; \perp)$. The first thing $\overline{[q]}(x_{i+1}, i+1)$ will subsequently do, however, will change both stacks. It calls $[p](i+1)$, with $p = goto(q, x_{i+1})$, if p exists. This means that p is pushed onto the parse stack, on top of q, and that x_{i+1} is popped

off the input stack, which then becomes $(x_{i+2}; ...; x_n; \perp)$. This is usually called a shift action.

2. The call of $\overline{[q]}(B, i)$, in case $B \rightarrow \cdot \in ini(q)$. This call means that B is pushed onto the input stack. Subsequently, $\overline{[q]}(B, i)$ will call $[p](i)$, with $p = goto(q, B)$: B is popped off again and p is pushed onto the parse stack. This is a reduce action for an ϵ-rule.

3. If $A \rightarrow \alpha\cdot \in q$, a reduce action for the non-ϵ rule $A \rightarrow \alpha$ takes place: $[q](i)$ produces $(A \rightarrow \alpha\cdot, i)$. This corresponds to pushing $A \rightarrow \alpha\cdot$ onto the input stack and popping q from the parse stack (the function terminates).

If a state symbol p is popped, another state symbol q appears at the top of the parse stack. This q refers to the $\overline{[q]}$ function that invoked $[p]$, because there was a $goto$ transition from q to p. The top of the input stack will always be an item in this case, but there are the two different cases treated in $\overline{[q]}$:

1. Item $A \rightarrow \alpha X \cdot \gamma$ is at the top of the input stack, and $A \rightarrow \alpha \cdot X\gamma$ is an element of state q. In this case two things happen: $A \rightarrow \alpha X \cdot \gamma$ is replaced by $A \rightarrow \alpha \cdot X\gamma$ and q is popped off the parse stack. After each reduce action for a non-ϵ rule $A \rightarrow \alpha$, this will happen $|\alpha| - 1$ times, so that $|\alpha| - 1$ state symbols are popped. In standard terminology, popping these state symbols also belongs to the reduce action for $A \rightarrow \alpha$.

2. The item $C \rightarrow X \cdot \delta$ is the top of the input stack, after the $|\delta|$ pops of the reduce action for $C \rightarrow X\delta$. The symbol underneath $C \rightarrow X \cdot \delta$ is x_{k+1}. Item $C \rightarrow \cdot X\delta$ is not in q but in $ini(q)$. The call of $\overline{[q]}(C, k)$ means that the item on top of the input stack is replaced by C. The first thing $\overline{[q]}(C, k)$ will do is call $[p](k)$, with $p = goto(q, C)$. In other words, C is popped and p is pushed onto the parse stack.

The sentence is correct if and only if the parse stack becomes empty, leaving behind an input stack $(S' \rightarrow \cdot S; \perp)$.

All this leads to the following algorithm for deterministic LR(0) parsing (assuming $V_N \cap V_T = \emptyset$):

```
parse:=(q0; ⊥);
input:=(x1; ...xn; ⊥);
while parse = (q; ...) and q ≠ ∅ do
  if input = (X...) then
    if A → α· ∈ q then
      push(A → α·, input); pop(parse)
    else if B → · ∈ ini(q) and X ∈ V_T then
      push(B, input)
    else
      pop(input); push(p, parse), where p = goto(q, X)
    fi
  else if input = (A → αX · β; ...) then
    if A → α · Xβ ∈ q then
      pop(input); push(A → α · Xβ, input); pop(parse)
    else (* α = ε, A → ·Xβ ∈ ini(q) *)
      pop(input); push(A, input)
    fi
  fi
od;
if parse = (⊥) and input = (S' → ·S; ⊥), the sentence is correct.
```

Here $\perp$ is formally added to V_T, so that X may take this value. Some global differences between the stack-based parser and its recursive predecessor are noteworthy. The stack-based parser is not recursive but iterative instead. Whereas the functional algorithms have two interpretations, the interpretative and the compiled view, the stack-based parser is only interpretative: no properties of the grammar are translated into code. In regard to efficiency, the elimination of function calls is an improvement, but the interpretative character of the result is a disadvantage.

As was the case in the previous section for the recursive version, refinements exist to get a more efficient stack-based implementation. For instance, items need never be pushed onto the input stack. If an item is at the top of the input stack, the "else if input = $A \rightarrow \alpha X \cdot \beta; \ldots$ then" part of the above algorithm is active. What happens here is a number of pops of the parse stack, replacing the item by another one at each step. Finally, the last item is replaced by a nonterminal. We can do away with this step-by-step procedure, if in a state q with $A \rightarrow \alpha\cdot \in q$ one immediately pops $|\alpha|$ states from the parse stack (including q) and pushes A onto the input stack:

```
parse:=(q0; ⊥);
input:=(x1; ...xn; ⊥);
while parse = (q; ...) and q ≠ ∅ and input = (X...) and X ≠ S' do
   if A → α· ∈ (q ∪ ini(q)) and X ∈ V_T then
      push(A, input); pop(parse), |α| times
   else
      pop(input); push(p, parse), where p = goto(q, X)
   fi
od;
if parse = (q0; ⊥) and input = (S'; ⊥), the sentence is correct.
```

Note that ϵ-rule reduction could be combined with the reduction of other rules here.

LR parsing has always been formulated in a stack-based way. There are many variants; the one we just discussed is among the relatively elegant ones. What should be learned here is that it is simple to deduce a stack-based implementation from a recursive one, in a systematic way. The reverse is more complicated, which is why it took so long before recursive ascent parsing made its appearance. The reason has to do with the level of specification. The stack-based formulations are at a lower level of abstraction, and it is always easy to go from a higher-level specification to a lower level, whereas it is usually very hard to do the reverse.

9.4 SOME VARIANTS

Two useful variants of LR(0) parsing exist, parallel to the variants of Earley-like recursive ascent parsing discussed before. In the first variant, ϵ-rules are treated differently to ensure that the recognition does not loop on grammars with derivations displayed in (6.9). The other variant is important only in the nondeterministic case and facilitates the construction of a compact parse forest after recognition.

9.4.1 First variant

Here is the LR variant of (6.14). Inspired by (6.13), let us try to implement a single-function recursive ascent LR parser:

$$
\begin{aligned}
&[q](\delta) = \\
&\quad \delta \in V_T^* \wedge A \to \alpha \cdot \beta \in q \wedge \beta \xrightarrow{*} \epsilon \rhd (A \to \alpha \cdot \beta, \delta) \mid \\
&\quad \delta = X\zeta \wedge A \to \alpha \cdot \beta \in q \wedge \beta \xrightarrow{*} X\eta \wedge \zeta = \eta\rho \rhd (A \to \alpha \cdot \beta, \rho).
\end{aligned} \tag{9.7}
$$

For this purpose we need function ini', the definition of which employs the conventional rewriting "$\to$" instead of "$\Rightarrow$",

$$ini'(q) = \{B \to \cdot\nu | B \to \nu \wedge A \to \alpha \cdot \beta \in q \wedge \beta \xrightarrow{*} B\gamma\},$$

and the function $goto'$ in terms of ini':

$$goto'(q, X) = \{A \to \alpha X \cdot \beta | A \to \alpha \cdot X\beta \in (q \cup ini'(q))\}.$$

Then the following is an implementation of (9.7):

$$
\begin{aligned}
&[q](\delta) = \\
&\quad \delta \in V_T^* \wedge A \to \alpha \cdot \beta \in q \wedge \beta \xrightarrow{*} \epsilon \rhd (A \to \alpha \cdot \beta, \delta) \mid \\
&\quad \delta = X\zeta \wedge A \to \alpha \cdot \mu X\gamma \in q \wedge \mu \xrightarrow{*} \epsilon \wedge \\
&\qquad (A \to \alpha\mu X \cdot \gamma, \rho) \leftarrow [goto'(q, X)](\zeta) \rhd (A \to \alpha \cdot \mu X\gamma, \rho) \mid \\
&\quad \delta = X\zeta \wedge C \to \cdot\mu X\nu \in ini'(q) \wedge \mu \xrightarrow{*} \epsilon \wedge \\
&\qquad (C \to \mu X \cdot \nu, \rho) \leftarrow [goto'(q, X)](\zeta) \rhd [q](C\rho).
\end{aligned} \tag{9.8}
$$

This algorithm terminates for any noncyclic grammar.

Note that if there are no ϵ-rules, (9.8) is functionally equivalent to (9.5), but of course it still consists of one function per state:

$$
\begin{aligned}
&[q](\delta) = \\
&\quad \delta \in V_T^* \wedge A \to \alpha\cdot \in q \rhd (A \to \alpha\cdot, \delta) \mid \\
&\quad \delta = X\zeta \wedge A \to \alpha \cdot X\gamma \in q \wedge \\
&\qquad (A \to \alpha X \cdot \gamma, \rho) \leftarrow [goto(q, X)](\zeta) \rhd (A \to \alpha \cdot X\gamma, \rho) \mid \\
&\quad \delta = X\zeta \wedge C \to \cdot X\nu \in ini(q) \wedge \\
&\qquad (C \to X \cdot \nu, \rho) \leftarrow [goto(q, X)](\zeta) \rhd [q](C\rho).
\end{aligned}
$$

It should be clear that a sentence ξ is correct if and only if

$$(S' \to \cdot S, \epsilon) \leftarrow [q_0](\xi).$$

9.4.2 Second variant

Here is an LR variant of (7.7) and (7.8). It implements the specifications

$$[q](i) = \quad A \to \alpha \cdot \beta \in \overline{q} \wedge \beta \xrightarrow{*} x_{i+1} \ldots x_j \ \rhd (A \to \alpha \cdot \beta, j)$$

$$\overline{[q]}(X, i) =$$
$$A \to \alpha \cdot \beta \in \overline{q} \wedge \beta \overset{*}{\Rightarrow} X\gamma \wedge \gamma \xrightarrow{*} x_{i+1} \ldots x_j \ \rhd (A \to \alpha \cdot \beta, j)$$

where $\overline{q} = q \cup ini(q)$. The recognizer is given by

$$\begin{aligned}
&[q](i) = \\
&\qquad \overline{[q]}(x_{i+1}, i+1) \mid \\
&\qquad B \to \cdot \in ini(q) \ \rhd \overline{[q]}(B, i) \mid \\
&\qquad A \to \alpha \cdot \in \overline{q} \ \rhd (A \to \alpha \cdot, i) \\
&\overline{[q]}(X, i) = \\
&\quad (A \to \alpha X \cdot \gamma, j) \leftarrow [goto(q, X)](i) \ \rhd (A \to \alpha \cdot X\gamma, j) \mid \\
&\quad C \to \cdot X\delta \in ini(q) \wedge (C \to X \cdot \delta, k) \leftarrow [goto(q, X)](i) \ \rhd \overline{[q]}(C, k)
\end{aligned} \tag{9.9}$$

The set Q defined in (7.4) to enable the calculation of a parse forest can alternatively be obtained from this algorithm by

$$\begin{aligned}
Q = \{&(\beta, i, j) | [q](i) \text{ was invoked and } (A \to \alpha \cdot \beta, j) \leftarrow [q](i)\} \cup \\
&\{(A, i, j) | [q](i) \text{ was invoked and } (A \to \cdot\beta, j) \leftarrow [q](i)\} \cup \\
&\{(x_{i+1}, i, i+1) | 0 \leq i < n\},
\end{aligned}$$

assuming that the functions were memoized and that access to the memoized results is possible after recognition.

9.5 LOOK AHEAD

As in recursive descent parsing, the efficiency and the "amount of determinism" of an LR(0) parser may be improved by adding look ahead. In the LR(0) case this amounts to the following. If an item $A \to \alpha \cdot$ is in some $\overline{q}$,

nonterminal A is recognized if $[q]$ is invoked, independent of the part of the input that has not been processed yet. It may occur that the recognition of this A does not lead to a successful derivation and that this can already be decided by looking at the next input symbol. Hence, we are interested in the set of terminals $f(q, A \rightarrow \alpha\cdot)$ that may directly follow an A that is recognized in state q. If this followers set is known, then the function

$$\begin{array}{l} [q](i) = \\ \quad x_{i+1} \in dir(q) \ \triangleright \ \overline{[q]}(x_{i+1}, i+1) \ \mid \\ \quad B \rightarrow \cdot \in ini(q) \wedge x_{i+1} \in f(q, B \rightarrow \cdot) \ \triangleright \ \overline{[q]}(B, i) \ \mid \\ \quad A \rightarrow \alpha\cdot \in q \wedge x_{i+1} \in f(q, A \rightarrow \alpha\cdot) \ \triangleright \ (A \rightarrow \alpha\cdot, i) \end{array} \tag{9.10}$$

is just as good as the one in (9.5), if $dir(q)$ is the set of terminals that are useful for q to start with:

$$dir(q) = \{x | A \rightarrow \alpha \cdot x\beta \in \overline{q}\}.$$

Using (9.10) instead of the corresponding function in (9.5), a number of reduce actions are prohibited that would not have led to consumption of the next input symbol anyway. To make optimal use of the look ahead, the grammar is usually supplemented with a rule $S' \rightarrow S \perp$ instead of $S' \rightarrow S$, with $\perp$ an end-of-input marker that is not in V. Hence, we assume $x_{n+1} = \perp$. The initial state is now $q_0 = \{S' \rightarrow \cdot S \perp\}$, and $(S' \rightarrow \cdot S \perp, n+1) \leftarrow [q_0](0)$ is equivalent to $S \stackrel{*}{\rightarrow} x_1 ... x_n$.

For some grammars the look ahead may be used to make the recognizer deterministic. An LR(0) automaton may be such that for each state q there is no overlap between the look ahead sets $f(q, A \rightarrow \alpha\cdot)$ and the set $dir(q)$. If this is the case, the grammar is called LALR(1). If a grammar has the the LALR(1) property, the above function (9.10), together with function $\overline{[q]}$ of (9.5), forms a deterministic recognizer.

How do we find the followers sets $f(q, A \rightarrow \alpha\cdot)$? Clearly, to find what may follow A we must look for the states p from which q can be reached and which contain the item $A \rightarrow \cdot\alpha$. For that purpose, let us generalize the function $goto$ so that its second argument can be an arbitrary string of symbols:

$$\begin{array}{l} goto(q, \epsilon) = q, \\ goto(q, XY\alpha) = goto(goto(q, X), Y\alpha), \end{array} \tag{9.11}$$

and $goto(q, X)$ satisfies the original definition (9.3). Then, if $A \rightarrow \alpha \cdot \in \overline{q}$ and $q = goto(p, \alpha)$, the item $A \rightarrow \cdot\alpha$ must be in $\overline{p}$, and hence there must be an item $B \rightarrow \gamma \cdot A\delta \in \overline{p}$. The terminals in $first(\delta)$ (defined in (4.7)) are followers of A and must be included in $f(q, A \rightarrow \alpha\cdot)$. If $\delta \xrightarrow{*} \epsilon$, however, the followers of B are followers of A as well, and one must go on and look for states r such that $p = goto(r, \gamma)$, for which $B \rightarrow \cdot\gamma A\delta \in \overline{r}$ must hold, et cetera.

To define f in a formal way, it turns out that it is convenient to consider a function f not only for items ending with a dot. In accordance with the above considerations, we define $f(q, A \rightarrow \alpha \cdot \beta)$ for all $q, A \rightarrow \alpha \cdot \beta$ such that $A \rightarrow \alpha \cdot \beta \in \overline{q}$, as follows (see also (4.9)):

$$\begin{aligned} f(q, A \rightarrow \alpha \cdot \beta) = {} & first(\beta) \cup \\ & \{x | \beta \xrightarrow{*} \epsilon \wedge q = goto(p, \alpha) \wedge B \rightarrow \gamma \cdot A\delta \in \overline{p} \wedge \\ & \quad x \in f(goto(p, A), B \rightarrow \gamma A \cdot \delta)\} \end{aligned} \tag{9.12}$$

This equation in general has more than one solution. Whereas (9.10) is correct for every solution, the maximum efficiency is obtained with the smallest possible followers sets. The smallest solution of (9.12) can be found by approximating it from below. Define the functions f_i, $i = 0, 1, 2...$, by

$$\begin{aligned} f_{i+1}(q, A \rightarrow \alpha \cdot \beta) = {} & first(\beta) \cup \\ & \{x | \beta \xrightarrow{*} \epsilon \wedge q = goto(p, \alpha) \wedge B \rightarrow \gamma \cdot A\delta \in \overline{p} \wedge \\ & \quad x \in f_i(goto(p, A), B \rightarrow \gamma A \cdot \delta)\} \end{aligned}$$

and $f_0(q, A \rightarrow \alpha \cdot \beta) = \emptyset$. From f_0 one can compute f_1, from f_1 one can compute f_2, et cetera. For some i this sequence must converge, i.e., $f_{i+1} = f_i$, and f_i is the smallest solution of (9.12).

The above can be generalized by looking ahead k terminal symbols instead of only one. The ensuing algorithms are named LALR(k) parsers in the literature; they are rarely used in practice for $k > 1$.

9.6 ATTRIBUTES

Here is the LR variant of the attribute grammar parser (8.5):

$$parser = \quad (S' \rightarrow \cdot S, n, f_{S'}) \leftarrow [q_0](0) \rhd f_{S'}(),$$

with

$$
\begin{array}{l}
[q](i) = \\
\qquad \overline{[q]}(x_{i+1}, i+1, f_{x_{i+1}}) \mid \\
\qquad B \rightarrow \cdot \in ini(q) \rhd \overline{[q]}(B, i, f_B^{B \rightarrow \epsilon}) \mid \\
\qquad A \rightarrow \alpha \cdot \in q \rhd (A \rightarrow \alpha \cdot, i, f_{\alpha \backslash A}^{A \rightarrow \alpha})
\end{array}
$$

$$
\begin{array}{l}
\overline{[q]}(X, i, f_X) = \\
\quad A \rightarrow \alpha \cdot X\gamma \in q \wedge (A \rightarrow \alpha X \cdot \gamma, j, f_{\alpha X \backslash A}) \leftarrow [goto(q, X)](i) \\
\qquad \rhd (A \rightarrow \alpha \cdot X\gamma, j, f_X \circ f_{\alpha X \backslash A}) \mid \\
\quad C \rightarrow \cdot X\delta \in ini(q) \ \wedge (C \rightarrow X \cdot \delta, k, f_{X \backslash C}) \leftarrow [goto(q, X)](i) \\
\qquad \rhd \overline{[q]}(C, k, f_X \circ f_{X \backslash C})
\end{array}
$$

The analogy is so immediate that it needs no further comments. We thus find that delayed attribute evaluation is very simple in LR parsing. By contrast, direct evaluation of OTF attributes cannot be done in LR recognizers, at least not in a natural way.

9.7 CONTINUATIONS

Until now we have used fairly simple functional techniques to specify all algorithms. Arguments of functions were generally of very simple types, except in the parser for attribute grammars. There is a style of functional programming that makes heavy use of passing functions as arguments to functions: the *continuation-passing* style.

Suppose the definition of a function f consists of invoking function g, followed by some other actions. These other actions will depend in general on the result of g, so they can be seen as a function that takes the result of g and produces the result of f. Let us call this function d. In the normal style, f calls g, waits until g has finished, becomes active again, and performs d. In the continuation style, f calls g and passes to g function d as an extra argument. When g has produced its result, then it applies d to this value instead of returning it: d is the *continuation*. In this way it can be avoided that functions return control to the function that invoked them (until after the

final result has been computed), and this has certain advantages. Continuation passing can be applied to all preceding recognition algorithms. We here illustrate the technique by showing how it affects LR(0) parsing. In the next section, an LALR(1) variant with continuation passing is given.

9.7.1 Extending the use of let

The quantifier **let** was used to bind variables of simple types. In this section, variables can just as well range over functions spaces, so we generalize its use. An example is

$$let\ f \cdot \forall_{i,j}[f(i,j) = i^2 + j^2] \rhd g(f),$$

where g has one argument that itself is a function with two arguments. In this case g is invoked with the function that computes the sum of the squares of its arguments as an actual argument. The expression $let\ f \cdot \forall_{i,j}[f(i,j) = i^2 + j^2]$ amounts to the definition of a "local" function f with formal parameters i, j. We keep writing **let** quantifications over function variables explicitly.

9.7.2 Example: the factorial

Let us take a simple function f with integer argument, the factorial function:

$$\begin{array}{l} f(i) = \\ \qquad i = 0 \rhd 1 \mid \\ \qquad i > 0 \rhd i * f(i-1). \end{array}$$

If this function is given a continuation c, this c should be applied to the results, so that we get

$$\begin{array}{l} f'(i,c) = \\ \qquad i = 0 \rhd c(1) \mid \\ \qquad i > 0 \rhd c(i * f(i-1)). \end{array}$$

It follows that $c(f(i)) = f'(i,c)$. To recursively call f' instead of f, we must determine the function d that performs the actions after $f(i-1)$ has produced

its result. In this case it is $d(j) = c(i * j)$, where j is a formal parameter that stands for the result of $f(i-1)$. The factorial function may thus be written as

$$
\begin{array}{l}
f'(i,c) = \\
\quad i = 0 \rhd c(1) \mid \\
\quad i > 0 \rhd let\ d \cdot \forall_j [d(j) = c(i * j)] \rhd d(f(i-1))
\end{array}
$$

and subsequently, because $d(f(i-1)) = f'(i-1, d)$, as

$$
\begin{array}{l}
f'(i,c) = \\
\quad i = 0 \rhd c(1) \mid \\
\quad i > 0 \rhd let\ d \cdot \forall_j [d(j) = c(i * j)] \rhd f'(i-1, d)
\end{array}
$$

which makes no use of the old function f anymore. The original factorial function can be obtained from the rewritten one by $f(i) = f'(i, id)$, where id is the identity function on integers.

9.7.3 LR(0) recognizer

Let us now in the same vein reformulate (9.5):

$$
\begin{array}{l}
[q]'(i,c) = \\
\qquad \overline{[q]}'(x_{i+1}, i+1, c) \mid \\
\qquad B \rightarrow \cdot \in ini(q) \rhd \overline{[q]}'(B, i, c) \mid \\
\qquad A \rightarrow \alpha \cdot \in q \rhd c(A \rightarrow \alpha \cdot, i)
\end{array}
$$

$$
\begin{array}{l}
\overline{[q]}'(X, i, c) = \\
\qquad let\ d \cdot \forall_{A \rightarrow \alpha X \cdot \gamma, j} [\\
\qquad\qquad\qquad\qquad d(A \rightarrow \alpha X \cdot \gamma, j) = \\
\qquad\qquad\qquad\qquad\quad A \rightarrow \alpha \cdot X\gamma \in q \rhd c(A \rightarrow \alpha \cdot X\gamma, j) \mid \\
\qquad\qquad\qquad\qquad\quad A \rightarrow \alpha \cdot X\gamma \in ini(q) \rhd \overline{[q]}'(A, j, c) \\
\qquad\qquad\qquad\qquad] \\
\qquad \rhd [goto(q, X)]'(i, d)
\end{array}
$$

Continuation d is passed to $[goto(q, X)]'$ to be applied to item-number pairs that are produced. Note that invocations $\overline{[q]}'$ are always the last action that is

to be performed. Therefore, the continuation that is passed to this function is identical to the continuation that is passed to $[q]'$. Because $[q](i) = [q]'(i, id)$, where id is the identity on item-number pairs, the recognizer should be invoked as $[q_0]'(0, id)$. The invocation $c(A \rightarrow \alpha\cdot, i)$ does exactly what is usually called a reduce action. It finds out in which state p the A was 'ordered' and calls $\overline{[p]}'(A, i, c')$, where c' is the continuation passed to $[p]'$. Note that recognition functions never return control to the function that invoked them, so there is no reason to implement them with a recursion stack. The classical parse stack is now implemented by the construction of continuation functions.

It is difficult to combine the continuation-passing technique with function memoization. Thus, continuation passing is most useful when memoization is not needed, i.e., if parsing is a deterministic or almost deterministic process.

9.8 ERROR RECOVERY

Parsing with continuations is especially useful for error recovery. There are many approaches to error recovery. The scheme presented below is not intended to be the optimal one. Rather, it is a very simple method that makes clear how functional methods can be used for this purpose.

The following deterministic continuation-passing LALR(1) parser without error recovery, is easily obtained from the previous sections:

$[q]'(i, c) =$
  if $x_{i+1} \in dir(q)$ then $\overline{[q]}'(x_{i+1}, i+1, c)$
  else if $B \rightarrow \cdot \in ini(q) \wedge x_{i+1} \in f(q, B \rightarrow \cdot)$ then $\overline{[q]}'(B, i, c)$
  else if $A \rightarrow \alpha\cdot \in q \wedge x_{i+1} \in f(q, A \rightarrow \alpha\cdot)$ then $c(A \rightarrow \alpha\cdot, i)$
  else *error*
  fi

$\overline{[q]}'(X, i, c) = [goto(q, X)]'(i, d)$
  where
    $d(A \rightarrow \alpha X \cdot \gamma, j) =$
      if $A \rightarrow \alpha \cdot X\gamma \in q$ then $c(A \rightarrow \alpha \cdot X\gamma, j)$
      else if $A \rightarrow \alpha \cdot X\gamma \in ini(q)$ then $\overline{[q]}'(A, j, c)$
      fi

To be able to dress this parser with error recovery, we need a few definitions. Firstly, we assume a set *landmarks*, as in section 4.5, of distinguished terminal symbols (among which must be $\perp$) that can be used as anchoring points for resuming parsing after an error. Secondly, we define a function F that takes a state q and a grammar symbol X (relevant if the error occurs in $\overline{[q]}(i, X)$, for some i) and produces a set of landmark terminals:

$$F(q, X) = \{x | x \in landmarks \wedge A \rightarrow \alpha \cdot \delta Y \beta \in \overline{q} \wedge x \in first(Y) \wedge \delta \overset{*}{\Rightarrow} X\gamma\}.$$

These are the terminals with which something can be done if an error occurs, after skipping δ (or rather, inserting γ). Thirdly, function *skip* gives for each state q and $x \in F(q, X)$ a string of grammar symbols that must be skipped before parsing can be resumed:

$$skip(q, x, X) = \delta, \text{ with } \delta \text{ such that there is an item } A \rightarrow \alpha \cdot \delta Y \beta \in \overline{q} \text{ and } x \in first(Y) \text{ and } \exists_\gamma \delta \overset{*}{\Rightarrow} X\gamma.$$

This δ need not be unique: any value that fits the description will do. As a heuristic, one may choose δ such that the derivation $\delta \overset{*}{\Rightarrow} X\gamma$ is as small as possible. Note the degrees of freedom here, both w.r.t. the set *landmarks* and the function *skip*. This freedom can be used to experiment with a given grammar to optimize the average recovery performance.

A version of the above LALR(1) parser with error recovery is given below. It has two extra parameters. One is F, with the same role as in the recursive descent parser of section 4.5: it contains the landmark terminals that must be looked for if an error occurs. Skipping input after the occurrence of an error until an element of F is found is done by the same function *delete* that was used in section 4.5. The other extra parameter is a function e, called the *error continuation*. The normal continuation c must be called if no errors are encountered. After an error, however, c is overruled, as it were, by the error continuation. Overruling the normal continuation may be seen as the functional way to manipulate the parse stack. Explicit manipulation of the parse stack is part of standard techniques to add error recovery to stack-based LR parsers. The error continuation consists of various pieces of code, one piece for every q that is "in the parse stack." For a given landmark terminal, it tries to find a nearby state at which parsing can be resumed, using the function *skip*. Here is the full parser with error recovery:

$[q]'(F, e, i, c) =$
  if $x_{i+1} \in dir(q)$ then $\overline{[q]}'(F, e, x_{i+1}, i+1, c)$
  else if $B \to \cdot \in ini(q) \wedge x_{i+1} \in f(q, B \to \cdot)$ then
    $\overline{[q]}'(F, e, B, i, c)$
  else if $A \to \alpha\cdot \in q \wedge x_{i+1} \in f(q, A \to \alpha\cdot)$ then $c(A \to \alpha\cdot, i)$
  else $error(F, e, q, i)$
  fi

$\overline{[q]}'(F, e, X, i, c) = [goto(q, X)]'(F \cup F(q, X), e_q, i, d)$
  where
    $d(A \to \alpha X \cdot \gamma, j) =$
      if $A \to \alpha \cdot X\gamma \in q$ then $c(A \to \alpha \cdot X\gamma, j)$
      else if $A \to \alpha \cdot X\gamma \in ini(q)$ then $\overline{[q]}'(F, e, A, j, c)$
      fi
  and
    $e_q(i) =$
      if $x_{i+1} \in F(q, X)$ and $skip(q, x_{i+1}, X) = \delta$ then
        $[goto(q, \delta)]'(F \cup F(q, X), e_q, i, h)$
      else $e(i)$
      fi
    where
      $h(A \to \alpha\delta \cdot \beta, j) =$
        if $A \to \alpha \cdot \delta\beta \in q$ then $c(A \to \alpha \cdot \delta\beta, j)$
        else if $A \to \alpha \cdot \delta\beta \in ini(q)$ then $\overline{[q]}'(F, e, A, j, c)$
        fi

Here $goto(q, \delta)$ is as in (9.11). Function *error* reports the error and subsequently calls the error continuation, after deleting input symbols until the first landmark terminal that is in F:

$error(F, e, q, i) =$ *output*('error at "i", x expected'); $e(delete(F, i))$

where x is some element of $dir(q)$ or, if $A \to \alpha\cdot \in \overline{q}$, of $f(q, A \to \alpha\cdot)$. The parser should be invoked as $[q_0]'(\emptyset, id, 0, id)$. It is left as an exercise to verify that, if the grammar is LALR(1), the above parser terminates for any input.

9.9 THE METHODS BY LANG AND TOMITA

In the literature, two other approaches to dealing with nondeterministic LR parsers have been proposed. Superficially they are rather different, but in fact they have a lot in common. Both are so-called breadth-first simulations of a nondeterministic parsing process. The term 'breadth-first' here means that the various branches of the parsing process are pursued in parallel.

In [Lang,1974] a general technique for simulating nondeterministic stack-based parsers is presented. The generality of this paper's method is in a sense also its weakness, because as a general method it cannot include refinements and simplifications that work only for specific parsing techniques. Lang's simulation uses a parse matrix as its central data-structure. Its main merits are that it establishes that cubic complexity can be achieved with many parsing methods, and thus that the Earley algorithm is not special in this respect, and that a cubic parse forest can be created as an 'output grammar', a set of rule applications.

In [Tomita,1986] a more practical, if no less complicated, implementation was worked out in some detail, which uses a device called the graph-structured stack. Tomita's method can be seen as a particular simulation of memoized functional LR-parsing. The simulation requires that the recursion stack is made explicit. If a nondeterministic LR-parser encounters a parsing conflict, in the Tomita simulation the parsing process is split into sub-processes by splitting the stack into as many branches as there are conflicting choices. All sub-processes (branches) are synchronized on the input: no branch consumes input symbol x_{i+1} until all others have consumed x_i. In this way, the simulation of nondeterministic functions employs what Tomita calls a tree-structured stack. If the functions are memoized, branches of the tree may come together; the tree becomes a graph. Tomita's technique produces a parser that has the same complexity as the corresponding memoized functional parser. This means that Tomita's technique in itself not necessarily produces cubic parsers, since memoized functional parsers not always have cubic complexity. For instance, as stated in section 9.2 it is efficient to replace the function $\overline{[q]}$ by an iterative loop, if the parser is deterministic. This can also be done if the parser is nondeterministic. Only, it is not wise to do so because, without some additional mechanism for result reuse, memoization of the resulting functions no longer yields a cubic algorithm.

Just as Lang's method applies not only to LR parsing but to a general class

of stack-based parsing methods, Tomita's technique is also quite general. It not only works for all functional parsers, but for other recursive algorithms as well. If it is applied to recursive ascent Earley parsing, the result is a stack-based Earley parser, which is in fact akin to standard implementations of the Earley parser; the parse matrix is replaced by a graph-structured stack.

9.10 EVALUATION W.R.T. STANDARD APPROACHES

The reader may have come to wonder about the merits and demerits of this book's presentation of LR parsing as compared to more conventional treatments. Let us, therefore, compare the properties of the above recursive ascent implementations of LR parsing with known ones.

The imperative implementation of section 9.2 is virtually identical to the recursive ascent parser discussed in [Kruseman,1988]. In fact, what has happened historically is that what Kruseman Aretz presented as an implementation trick, still based on standard theory, was the starting point of the research reported in this book.

The standard stack-based implementation of LR parsers is related to the functional ones in that the recursion stack, an implicit data structure in a functional program, is made explicit. This was illustrated in section 9.3. While explicit handling of the stack may sometimes have some practical advantages (see below), it should be presented as it is: a low-level implementation technique. In reality, however, pushdown automata are in the realm of theory, and the parse stack figures at the highest levels of abstraction. It is difficult to understand why simple recursive implementations of LR parsers were found as late as 1988. The theoreticians' historical preoccupation with automata probably is a major factor. As we saw above, the theory behind functional implementations is very simple, whereas the theory of standard implementations is quite complicated (see, for instance, [Hesselink,1992]). The reason is that a lot of proof obligations in the standard theory are related to the parse stack. The situation becomes more extreme if one studies LR parsing in the nondeterministic case, i.e., for arbitrary grammars. The standard implementation must be revised totally to accommodate nondeterminism, and one gets algorithms like Lang's and Tomita's. In the functional theory, by contrast, the implementation of deterministic LR parsers is related in very simple way to the nondeterministic version. In fact, we derived the former from the latter

in section 9.2. Function memoization, which does not change the abstract algorithm at all, is used to improve efficiency of the nondeterministic parser.

The following table summarizes the main differences between stack-based and recursive ascent implementation techniques:

Stack-based implementations	Functional implementations
A state is an elementary symbol	A state 'is' a pair of functions
State transitions	Function calls
Parse stack	Recursion stack
Lang's technique, graph-structured stack	Function memoization, bunch-valued functions
Correctness proofs tedious	Proofs by simple calculation

An advantage of the methods discussed in section 9.9 is that they are better suited to online-parsing, i.e., for parsing during sentence production. The reason is that both these methods are completely breadth-first: at each point in the sentence, everything that can be computed is computed before going to the next word. However, the price to pay is high. Whereas the implementation of a functional (recursive ascent) parser is completely trivial, especially in a functional language (see [Norvig,1991] for an implementation of memo-functions in LISP), the specification and implementation of for instance the Tomita parser is much more difficult and error-prone.

A further possible advantage of stack-based approaches is due to their iterative nature. If a programming language is used with a relatively expensive function-call mechanism, efficiency could be gained by avoiding such calls where possible. A disadvantage of the iterative solutions is that they are purely interpretative. Each parse action can only be performed after consultation of a parse table. In a compiled implementation of a functional parser, by contrast, the parse table is converted into code by explicitly creating for every state its associated functions.

A last point concerns the construction of a cubic parse forest. This seems to be problematic in any case. We propose that the simplest way to construct such a forest is to derive it from a revised recognizer. In [Tomita,1986] a more direct procedure was advocated, which does not lead to a cubic forest for arbitrary grammars. If one is prepared to settle for less than a cubic

parser, it is easy to modify our LR parser to make it a parser with $O(n^4)$ time-complexity that computes on-the-fly a cubic parse forest. The modification involves a global data-structure to store (partial) parse forests, for each subsequence $x_{i+1}...x_j$ of the input, for nonterminals and postfixes of right-hand sides of grammar rules that derive that subsequence. In addition, the function $\overline{[q]}$ gets an extra integer parameter (which makes the time-complexity go to $O(n^4)$) to encode exactly which part of the input is derived from its parameter X, so that the parse forest for X in the global data-structure can be accessed. The construction of an output grammar, part of Lang's simulation method and reproposed in [Billot,Lang,1989], gives a cubic parse forest in cubic time. The drawback of this method is that its practicality remains to be shown.

9.11 EARLEY VERSUS LR

For ambiguous grammars, Earley-like parsing is in general more efficient than nondeterministic LR-parsing. The reason is that the number of functions in the recognizers of chapter 6 is a linear function of the grammar size, whereas the number of functions in an LR-parser is exponential in the grammar size. On the other hand, for LR(0) or LALR(1) grammars it is more efficient to use LR parsing. Also for grammars that are "almost" LR(0), it can be wise to use nondeterministic LR(0) parsing instead of Earley-like parsers. In general, an LR(0) parser is to be preferred if either

1. the number of states in an LR(0) automaton of the grammar is less than the number of items, or
2. the grammar is "almost" LR(0).

If the first condition is true, LR(0) parsing is more efficient even if the grammar is very ambiguous. The second condition is not a formal one, but it is clear that if it holds true, the grammar cannot be very ambiguous. It should be noted that should a grammar be very ambiguous and lead to a *huge* LR(0) automaton, LR(0) parsing is the last thing one should consider.

It has been claimed ([Tomita,1986]) that for some grammars of English LR(0) parsing is more efficient than Earley parsing. It is not quite clear, however, on which grounds. In [Tomita,1986], both the size argument and the "almost LR" argument are used, but it is the last argument that seems to be the most

serious one. However, it is the author's experience that natural language grammars are *very* ambiguous, and that the ambiguity increases with the sophistication of the grammar. Also, grammars for Dutch, and languages like it, tend to be more ambiguous than grammars for English, because of word-order differences between the two languages. Moreover, it is important to realize that natural language grammars are *artifacts*. If one is writing an attribute grammar for a natural language, one has choices of expression that influence the ambiguity of the backbone grammar. For instance, if a subdivision of noun phrases is needed, one may introduce a new nonterminal, for a new kind of noun phrase, or one may add some attribute to the existing attributes of noun phrases. The second option will result in a more ambiguous backbone grammar, in general.

An important difference between Earley and LR parsers is that in LR parsers one tries to postpone commitment to specific grammar rules by trying to treat rules collectively in states. This makes LR parsers less extensible than Earley parsers. A case in point is the extension to attribute grammars. Earley parsers allow simple OTF-like attribute evaluation (see previous chapter); LR parsers do not.

Exercises

1. Prove by induction that the absence of shift-reduce and reduce-reduce conflicts guarantees that the LR(0) recognizer is deterministic.

2. Is the grammar $S \rightarrow xSy$, $S \rightarrow xy$ an LR(0) grammar?

3. Construct the LR(0) automaton for the grammar $S' \rightarrow S \perp$, $S \rightarrow xSy$, $S \rightarrow \epsilon$. This grammar is not LR(0) because of the shift-reduce conflicts in the states $\{S' \rightarrow \cdot S \perp\}$ and $\{S \rightarrow x \cdot Sy\}$. Show that

   $$f(\{S' \rightarrow \cdot S \perp\}, S \rightarrow \cdot) = \{\perp\}$$
   $$f(\{S \rightarrow x \cdot Sy\}, S \rightarrow \cdot) = \{y\}$$

   and that this implies that the grammar is LALR(1).

4. Apply the look-ahead technique to one of the recognizers of chapter 6.

5. Verify that there is no elegant way to add OTF attribute evaluation to LR recognizers.

6. Assume that each grammar symbol has one synthesized attribute, of type "parse tree." What should the attribute function involving these

attributes look like, for an arbitrary grammar rule, such that the parser of section 9.6 produces a parse tree for each correct sentence?

7. Reformulate (6.2),(6.3) in continuation-passing style (which is easy). Do the same for (4.3) (which requires more labor).

8. Prove that the error recovery in the above LALR(1) parser is such that, after an error, there will always be progress in terms of input consumption. Hint: Prove by induction that invocations $[q]'(F, e, i, c)$ can only lead to function calls $e(j)$ with $x_j \in F$, and that from this it follows that $e(j)$ leads to the consumption of at least one terminal.

9. Prove that if the grammar is LALR(1), the LALR(1) parser with error recovery terminates for any input.

10. Define an LR(0) parser for EBNF grammars (see section 6.5).

10

SOME NOTES

This book is an unconventional account of conventional subjects of parsing theory. It differs from existing treatises in a number of aspects. Most importantly, of course, the functional theory of (recursive ascent) parsing is new. Also, the way context-free grammars are introduced and interpreted is fairly nonstandard, as is the usage of bunch-valued functions and (less importantly) Lambek types.

The choice of some examples and various comments reveal my connection with natural language research. I developed my ideas doing such research at Philips Research Laboratories and at the Institute for Perception Research (IPO), both in Eindhoven, The Netherlands.

Below, I list a number of comments on the material provided in the preceding text, and I cite relevant references (without trying to be complete), following more or less the order of presentation.

10.1 CONTEXT-FREE GRAMMARS

The way context-free grammars have been formally introduced is unconventional. Normally, they are presented as rewriting systems - as one level in the Chomsky hierarchy (see [Chomsky,1959]). I always felt uneasy about this injustice: the formalism of context-free grammars is not part of one hierarchy; it is the mother of all grammar hierarchies. Therefore, I decided to start differently. Among other things, my approach makes it easier to formulate the theorem of section 3.3. The Lambek types were introduced in [Lam-

bek,1958]. This work led to the notion of categorial grammars, which gained considerable popularity in linguistics ([Oehrle,Bach,Wheeler,1988]). Most of my understanding of Lambek calculus is due to papers by Buszkowski, from whom I also learned the term 'natural semantics'. I was particularly inspired by the work reported in [Buszkowski,1982]. A good exposition of the theory of Galois connections is [Backhouse,van der Wouden,1992]. The relation between Galois connections and Lambek types was pointed out to me by Lex Augusteijn. The inequalities that we called type tautologies appear in many contexts, for instance, in the program calculus of [Hoare et al.,1987], a functional variant of which is applicable to the bunch-functional notation we have used. The standard reference to linear logic is [Girard,1987]. For some consequences of the relation between linear logic and Lambek calculus, see [Roorda,1991].

10.2 NAMES

The term *recursive descent* parser normally denotes *deterministic* parsers *with* look ahead, whereas I introduced the term for *nondeterministic* parsers *without* look ahead. I know of some workers in the field that object to this, but I feel that it is unjustified to use elegant terminology "X" to parsers with certain mathematically unnatural restrictions, and to invent names like "generalized X" or "extended X" for the parsers without such restrictions. The same considerations apply to terms like *recursive ascent* and *LR parsing*.

Historically, a parser is a recognizer that also produces parse trees. In some contexts, I have used the word parser for algorithms that do not construct parse trees. In particular, the word parser was used for recognizers with attribute evaluation but without parse tree construction. This is very appropriate, though, for two reasons. On the one hand, a parse tree can be constructed as an attribute. On the other, parse trees are not necessary to compute attributes, and in many cases (especially in compilers) it is the attributes that count as the ultimate output of the parsing process. So, the historical link between the word parser and the creation of parse trees has become somewhat unfortunate.

10.3 BUNCHES

The notion of *bunches* comes from [Hehner,1984]; I adapted the notation from [Norvell,Hehner,1992]. [Hehner,1993] gives an illuminating explanation of the bunch concept, and of the relation between bunches and sets. This work also proposes to make a distinction between strings and sequences, which also exists between bunches and sets: strings have the singleton property, but sequences do not. I followed this convention, referring to elements of V^* as strings. Sets with nondeterministic interpretation, like bunches, were also proposed in [Hughes,O'Donnell,1990]. In [Wadler,1992] a kind of bunch-valued lambda-calculus is discussed. Throughout most of this book, bunches should be interpreted as collections, not as nondeterministic values. Nevertheless, a general bunch-valued function is referred to as being nondeterministic, unless it is defined in terms of mutually excluding guards. From a formal-semantics point of view the difference is not so big: if a language is literally nondeterministic, the natural formal meaning of a program is the set of values that it could produce. This is why multiple-valued algorithms have been called nondeterministic by many authors; an early reference is [Floyd,1967]. Anyway, I hope that the terminology does not lead the reader to the wrong interpretation of bunches.

Our functional language is "nondeterministic" and its variables are definite. This necessarily means that functions are not unfoldable. For a discussion on this, and on alternative interpretations of nondeterministic languages, see [Sondergard,Sesoft,1990] and [Wadler,1992].

10.4 FUNCTIONAL PROGRAMMING

In using a functional programming style to define algorithms, I intend to promote the functional approach to programming. I hope I have made clear, however, that this does not mean that functional programming languages must be used, although at least for rapid prototyping functional languages are a joy. In our case it would be nice to have a language that supports bunch notation. I wish to emphasize, however, that imperative third-generation programming languages have many practical advantages - that is, if they have enough primitives to implement recursive functions and higher-order functions straightforwardly.

Throughout this book, implementations are derived from specifications. A

specification is a function definition from which the input-output relation is immediately clear. Its implementation is a recursive definition of the same function, from which it is immediately clear how to formulate it in a programming language. In our usage of the term, implementations are not necessarily constructive: they need not terminate.

10.5 GRAMMAR TRANSFORMATIONS

The grammar transformation $G \mapsto F_G$ studied in chapter 5, to eliminate left recursion, is almost identical to one in the second volume of [Sippu,Soisalon-Soininen,1990], where it plays the same role: a top-down parser applied to the transformed grammar is a parser "between" LL and LR parsers for the original grammar. In [Sippu,Soisalon-Soininen,1990], a top-down parser on the transformed grammar is called a *Predictive LR* (PLR) parser, so that the first variants (5.25) and (6.2),(6.3) of the recursive ascent recognizer parser could be called (nondeterministic, recursive ascent, memoized) PLR(0) parsers, ignoring some small differences between $G \mapsto F_G$ and the transformation in [Sippu,Soisalon-Soininen,1990]. The subject of grammar transformations is intimately related to parsing theory (see, for example, [Nijholt,1980]). We concentrated on language equivalence between grammars ("weak" equivalence) and did not study structure preservation, i.e., the connection between parse trees. This was not necessary because we used grammar transformation only to derive a recursive ascent *recognition* algorithm.

An alternative method to link recursive descent and ascent techniques, using program transformation rather than grammar transformation, is given in [Augusteijn,1993].

10.6 MEMO-FUNCTIONS

A recent reference to the notion of memo-functions is [Hughes,1985]. Lex Augusteijn introduced them to me. The application of memoization to parsing theory is not new. In [Sheil,1976], top-down parsers are discussed for general grammars, with a "dynamic programming" method. Had the paper been written in a functional way, this would have amounted to memoization. More recently, in [Norvig,1991], memoization was applied explicitly to top-

down parsing. For readers that would like to implement recursive ascent parsers in LISP, the last reference presents an interesting implementation of memo-functions.

10.7 PARSE FORESTS

The construction of parse forests is not new. Our treatment can be seen as an elaboration of the partial description of parse forests in [Earley,1970]. In [Billot,Lang,1989], parse forests are represented as sets of rule applications. [Tomita,1986] is also a popular reference, although the claim in this work that cubic representations of parse forests in general do not exist is mistaken. The use of memoization in the construction of parse forests comes from [Leermakers,Augusteijn,Kruseman Aretz,1992].

10.8 EARLEY

Earley's original paper appeared more than two decades ago [Earley,1970]. The algorithm is akin to the algorithm attributed to Cocke, Younge, and Kasami (CYK), presented, for instance, in [Aho,Ullman,1972]. The relationship involves the grammar transformation $G \mapsto \overline{E}_G$ and was first reported as a "deep" result in [Graham,Harrison,Ruzzo,1980]. A recursive ascent implementation first appeared in [Leermakers,1992a]. I took the liberty of attaching Earley's name to algorithms that are not totally functionally equivalent to the original, and of distinguishing them by the qualifier "recursive ascent." This is to avoid the confusion caused by the unnecessary usage of different names for what is essentially the same thing. An example in this area is a technique called *chart parsing* in the computational linguistics community. It differs from Earley parsing only in the way the algorithm is displayed.

10.9 ATTRIBUTE GRAMMARS

The notion of attribute grammars is from [Knuth,1968]. Among other things it contains an algorithm for checking whether or not an attribute grammar is circular. [Augusteijn,1990,1993] contains detailed information about how to

implement function graphs and how to conduct their evaluation, and in particular how to handle pseudo-circular attribute grammars. [Augusteijn,1993] gives a slightly different view on attribute grammars and another way to turn recursive ascent LR parsers into parsers for attribute grammars.

The use of bunch-valued attribute functions instead of ordinary attribute functions amounts to a generalization of the usual formalism that can also be formulated by using attribute relations instead of functions. Perhaps the first to propose such attribute grammars was [Hemerik,1984].

The attribute grammar that can be defined by the OTF attribute grammar formalism are called L-attribute grammars in the literature (see [Deransart, Jourdan, Lorho,1988]). These are defined as the most powerful attribute grammars such that attribute evaluation is possible during LL-parsing. Our treatment shows that, for these grammars, direct attribute evaluation is also possible in recursive ascent Earley-like parsers.

10.10 NATURAL LANGUAGE

I mentioned compiler technology and natural language processing as the two areas in which attribute grammars are useful. However, the usefulness to natural language is limited: the attribute grammar formalism (in particular the OTF kind) is suitable for natural language, but only if one is pragmatic. The formalism does not facilitate concise expression of many types of generalizations that exist in natural languages. As a result, a comprehensive attribute grammar for natural language will contain too much redundancy. Moreover, it is difficult to interpret natural language compositionally, if the (backbone) grammar is context-free (see [Rosetta]). Nevertheless, attribute grammars play a rôle in many practical natural-language processors. In [Rosetta], for instance, the component 'surface parser' is a parser for an attribute grammar with ambiguous, i.e., bunch-valued, attribute functions.

10.11 OTHER APPLICATIONS

I mentioned no application areas for (attribute) grammars other than programming and natural languages. The reason is that the corresponding two scientific communities contributed most to the literature on parsing. This

does not mean that other applications do not exist. Grammars are used in biology and music theory, and for visual applications like pattern recognition.

10.12 LR PARSING

The original paper on LR parsing is [Knuth,1965]. Ever since, LR parsing has been a lively subject and books about it keep appearing. General textbooks like [Aho,Ullman,1977] and [Aho,Sethi,Ullman,1986] contain extensive treatments of the subject. Two recent books devoted largely to LR parsing are [Sippu,Soisalon-Soininen,1990] and [Grune,Jacobs,1990]. In existing textbooks one will not find the functional approach of this essay. For some reason or other it took almost 25 years for someone to find out that recursive implementation techniques, standard for top-down parsing (recursive descent), are equally applicable to LR parsing. After all these years the invention was made in triplicate. Completely independently and simultaneously, the same ideas were reported in [Kruseman Aretz,1988] and [Barnard,Cordy,1988]. In [Roberts,1988] and [Roberts,1989] a similar recursive implementation was presented as an abstraction from earlier work by Pennello ([Pennello,1986]). In a rudimentary sense, the latter work is the birth of recursive ascent parsing. The above theory behind recursive ascent LR(0) parsing originally appeared in [Leermakers,1991] and [Leermakers,Augusteijn,Kruseman Aretz,1992]. This book's presentation of parsing is an outgrowth of [Leermakers,1992b]. The first approach to the simulation of nondeterministic pushdown automata was [Lang,1974]. A recent implementation for nondeterministic stack-based LR parsing is [Tomita,1986]. The calculation of LALR(1) look ahead sets by solving a recursive equation comes from [Kristensen,Madsen,1981]. An efficient way of performing such calculation can be found in [DeRemer,Pennello,1982]. For literature on continuations, see [Appel,1992]. The presented parsers with error-recovery were inspired by [Augusteijn,1993].

10.13 EBNF

For reasons of simplicity we studied parsers for standard context-free grammars. Most of the theory carries through for grammars with regular expressions over grammar symbols at right-hand sides of grammar rules (EBNF grammars). In [Earley,1970] it was already noted that the Earley algorithm

can be applied to such grammars. For the (recursive ascent) LR(0) parsing of grammars with regular expressions, see [Leermakers,1991]. Attribute grammars are an exception to the generalizability of results: it is difficult to make a general attribute grammar formalism with EBNF backbones. OTF-like EBNF grammars may be written, however. For details, see [Rosetta].

10.14 CONCLUSION

The algorithms of this book are presented in a formal way. Yet, unlike too many parts of theoretical computer science, its theory is an abstraction from real practice. The chapter on attribute grammars, for instance, gives the essence of practical tools for compiler construction and natural language parsing. If one wants to apply LR parsing to natural-language grammars, chapter 9 provides the simplest method available. As for the relevance of functional techniques to artificial-language parsing, it suffices to recall that recursive ascent parsing was originally discovered in an application-driven endeavor to find efficient implementations of LR parsing. Indeed, recursive ascent LR parsers can be made extremely efficient, using techniques developed for optimizing recursive descent parsers (see [Penello,1986]). There seem to be few rational reasons to use anyting other than recursive parsing algorithms. Let me end with a quotation from [Wirth,1976], which very aptly describes the raison d'être of this book:

> The fact that implementations of recursive procedures on essentially non-recursive machines exist proves that for practical purposes every recursive program can be transformed into a purely iterative one. This, however, involves the explicit handling of a recursion stack, and these operations will often obscure the essence of a program to such an extent that it becomes most difficult to comprehend. The lesson is that algorithms which by their nature are recursive rather than iterative should be formulated as recursive procedures.

REFERENCES

[Aho,Sethi,Ullman,1986] Aho A.V., Sethi R., and Ullman J.D., *Compilers, Principles, Techniques, and Tools* (Addison-Wesley, Reading, Mass., 1986).

[Aho,Ullman,1972] Aho A.V. and Ullman J.D., *The Theory of Parsing, Translation, and Compiling* (Prentice Hall, Englewood Cliffs, 1972).

[Aho,Ullman,1977] Aho A.V. and Ullman J.D., *Principles of Compiler Design* (Addison-Wesley, Reading, Mass., 1977).

[Appel,1992] Appel A.W., *Compiling with Continuations* (Cambridge University Press, Cambridge, 1992).

[Augusteijn,1990] Augusteijn L., The Elegant compiler generator system, in *Attribute Grammars and their Applications* edited by P. Deransart and M. Jourdan, *Springer Lecture Notes in Computer Science* 461 (Springer-Verlag, Berlin, 1990).

[Augusteijn,1993] Augusteijn A.A., *Functional Programming, program transformations and compiler construction*, Ph.D. Thesis, Technical University, Eindhoven, The Netherlands.

[Backhouse,van der Wouden,1992] Backhouse R. and van der Wouden J., Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithms, Part 3: A Relational Theory of Data Types - Part 1.

[Barnard,Cordy,1988] Barnard D.T. and Cordy J.R., SL parses the LR languages, *Computer Languages* (1988) 13(2), 65-74.

[Billot,Lang,1989] Billot S. and Lang B., The structure of shared forests in ambiguous parsing, *Proceedings of* 27th *meeting of the Association for Computational Linguistics* (Vancouver 1989), 143-151.

[Buszkowski,1982] Buszkowski W., Compatibility of a categorial grammar with an associated category system, *Zeitschr. f. math. Logik und Grundlagen d. Math.* (1982) 28, 229-238.

[Chomsky,1959] Chomsky N., On certain formal properties of grammars, *Information and Control* (1959) 2(2), 137-167.

[Deransart, Jourdan, Lorho,1988] Deransart P., Jourdan M., and Lorho B., Attribute grammars, *Lecture Notes in Computer Science* 323 (Springer-Verlag, Berlin, 1988).

[DeRemer,Pennello,1982] DeRemer F. and Pennello T., Efficient computation of LALR(1) lookahead sets, *ACM Transactions on Programming Languages and Systems* (1982) 4, 615-649.

[Dijkstra,1976] Dijkstra E.W., *A Discipline of Programming* (Prentice Hall, Englewood Cliffs, 1976).

[Earley,1970] Earley J.C., An efficient context-free parsing algorithm, *Communications of the ACM* (1970) 13(2), 94-102.

[Floyd,1967] Floyd R.W., Nondeterministic algorithms, *Journal of the ACM* (1967) 14(4), 636-644.

[Girard,1987] Girard J-Y., Linear logic, *Theoretical Computer Science* (1987) 50, 1-102.

[Graham,Harrison,Ruzzo,1980] Graham S.L., Harrison M.A., and Ruzzo W.L., An improved context-free recognizer, *ACM Transactions on Programming Languages and Systems* (1980) 2, 415-462.

[Grune,Jacobs,1990] Grune D. and Jacobs C., *Parsing Techniques, a Practical Guide* (Ellis Horwood, Chichester, 1990).

[Hehner,1984] Hehner E.C.R., *The Logic of Programming* (Prentice Hall, Englewood Cliffs, 1984).

[Hehner,1993] Hehner E.C.R., *A Practical Theory of Programming* (Springer-Verlag, Berlin, 1993).

[Hemerik,1984] Hemerik C., *Formal Definitions of Programming Languages as a Basis for Compiler Construction*, Ph.D. Thesis, Technical university, Eindhoven, The Netherlands.

[Hesselink,1992] Hesselink W.H., LR-parsing derived, *Science of Computer Programming* (1992) 19, 171-196.

[Hoare et al.,1987] Hoare C.A.R. et al., Laws of programming, *Communications of the ACM* (1987) 30, 672-686.

[Horspool,Whitney,1990] Horspool R.N. and Whitney M., Even faster LR parsing, *Software-Practice and Experience* (1990) 20(6), 515-535.

[Horspool,1991] Horspool R.N., Recursive ascent-descent parsers, *Lecture Notes in Computer Science* 477, 1-10 (Springer-Verlag, Berlin, 1991).

[Hughes,1985] Hughes J., Lazy memo-functions, in *Functional Programming Languages and Computer Architecture*, edited by J.-P. Jouannaud, *Springer Lecture Notes in Computer Science* 201 (Springer-Verlag, Berlin, 1985).

[Hughes,O'Donnell,1990] Hughes J. and O'Donnell J., Expressing and reasoning about non-deterministic functional programs, *Functional Programming*, (Glasgow 1989), edited by K. Davis and J. Hughes, Workshops in Computing (Springer-Verlag, Berlin, 1990).

[Knuth,1965] Knuth D.E., On the translation of languages from left to right. *Information and Control* (1965) 8, 607-639.

[Knuth,1968] Knuth D.E., Semantics of context-free languages, *Mathematical Systems Theory* (1968) 2(2), 127-145.

[Kristensen,Madsen,1981] Kristensen B.B. and Madsen O.L., Methods for computing LALR(k) lookahead, *ACM Transactions on Programming Languages and Systems* (1981) 3, 60-82.

[Kruseman Aretz,1988] Kruseman Aretz F.E.J., On a recursive ascent parser, *Information Processing Letters* (1988) 29, 201-206.

[Lambek,1958] Lambek J., The mathematics of sentence structure, *American Mathematical Monthly* (1958) 65, 154-170.

[Lang,1974] Lang B., Deterministic techniques for efficient non-deterministic parsers, *Springer Lecture Notes in Computer Science* 14, 255-269 (Springer-Verlag, Berlin, 1974).

[Leermakers,1991] Leermakers R., Non-deterministic recursive ascent parsing, *Fifth Conference of the European Chapter of the Association of Computational Linguistics* (Berlin 1991), 63-68.

[Leermakers,1992a] Leermakers R., A recursive ascent Earley parser, *Information Processing Letters* (1992) 41, 87-91.

[Leermakers,1992b] Leermakers R., Recursive ascent parsing: from Earley to Marcus, *Theoretical Computer Science* (1992) 104, 299-312.

[Leermakers,Augusteijn,Kruseman Aretz,1992] Leermakers R., Augusteijn L., and Kruseman Aretz F.E.J., A functional LR parser, *Theoretical Computer Science* (1992) 104, 313-323.

[Nijholt,1980] Nijholt, A., Context-free grammars: Covers, normal Forms, and parsing, *Springer Lecture Notes in Computer Science* 93 (Springer-Verlag, Berlin, 1980).

[Norvell,Hehner,1992] Norvell T.S. and Hehner E.C.R., Logical specifications for functional programs, *Proceedings of the Second International Conference on the Mathematics of Program Construction* (Oxford, 1992).

[Norvig,1991] Norvig P., Techniques for automatic memoization with applications to context-free parsing, *Computational Linguistics* 17 (1991), 91-98.

[Oehrle,Bach,Wheeler,1988] Oehrle R.T., Bach E., and Wheeler D. (eds), *Categorial Grammars and Natural Language Structures* (Reidel, Dordrecht, 1988).

[Pennello,1986] Pennello T.J., Very fast LR parsing, *Sigplan Notices* (1986) 21(7), 145-151.

[Roberts,1988] Roberts G.H., Recursive Ascent: An LR analog to recursive descent, *SIGPLAN Notices* (1988) 23(8), 23-29.

[Roberts,1989] Roberts G.H., Another note on recursive ascent, *Information Processing Letters* (1989) 32(5), 263-266.

[Roorda,1991] Roorda D., *Resource logics: proof-theoretical investigations*, Ph.D. Thesis, University of Amsterdam.

[Rosetta] Rosetta M.T., *Compositional Translation* (Kluwer Academic Publishers, to appear). This is a monograph on Philips' Machine Translation system Rosetta.

[Salomaa,1973] Salomaa A., *Formal Languages* (Academic Press, New York, 1973).

[Sheil,1976] Sheil B.A., *Observations on Context Free Parsing* in *Statistical Methods in Linguistics* (Stockholm, Sweden, 1976). Also: Technical Report TR 12-76, Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard University, Cambridge, Mass.

[Sippu,Soisalon-Soininen,1990] Sippu S. and Soisalon-Soininen E., *Parsing Theory*, two volumes (Springer-Verlag, Berlin, 1990).

[Sondergard,Sesoft,1990] Sondergard H. and Sesoft P., Referential Transparency, definiteness and unfoldability, *Acta Informatica* (1990) 27, 505-517.

[Tomita,1986] Tomita M., *Efficient Parsing for Natural Language, A Fast Algorithm for Practical Systems* (Kluwer Academic Publishers, Boston, 1986).

[Wadler,1992] Wadler P., The essence of functional programming, *19th Annual Symposium on Principles of Programming Languages* (Santa Fe, 1992).

[Wirth,1976] Wirth N., *Algorithms + Data Structures = Programs*, p. 130 (Prentice Hall, Englewood Cliffs, 1976).

INDEX

A

Arb, 80
Attribute grammar, 31, 41, 91, 130, 147
Automaton, xiii, 116, 149

B

Bilinear grammar, 36, 45, 57, 78
  definition, 36
Bottom up
  composition of attribute functions, 95
  recognition/parsing, 64, 75, 79, 87
Bunch, 7–8, 12, 145
  interpretation of, 8, 12, 145
Bunch notation, 7, 11–12, 96, 104
  algorithmic interpretation of, 12
  laws for, 11

C

Chomsky hierarchy, 29, 143
Closure, 20, 52, 95
Compiled view, 37, 61, 68
Complexity, 35, 45, 68, 82
Compose, 93
Composition
  of attribute functions, 93, 95, 98, 100, 102, 108
  of bunch-valued functions, 27
Concat, 93
Continuations, 131
Cut rule, 19, 24
Cyclic grammar, 45, 56–57, 65, 69, 80, 112

D

Definiteness of variables, 10, 145
Delete, 43, 135
Derivation, 20, 47, 54, 91, 95
Deterministic algorithm, 7, xiii–14, 39–40, 115, 124, 129, 144
Distributivity property of bunches, 7–8, 11, 16, 34, 62, 67, 70, 106
Division by a language, 23
Dressed derivation, 95

E

Earley parser, 61, 68, 85, 87, 115, 138, 140
  standard formulation of, 85
EBNF grammar, 69, 112, 149
  attributed version, 113
  recursive ascent recognition, 72, 142
Efficiency, 37, 41, 64, 69, 103, 110, 112, 115, 121, 125, 128, 130, 139
Epsilon-rules, 3, 45, 56, 65, 78, 96, 108, 124, 126–127
Error, 41–42, 121, 136
Error continuation, 135
Error recovery
  recursive ascent, 134
  recursive descent, 42
Exec, 103

F

First, 39, 130
Followers set, 40, 129

For, 81

G

Galois connection, 25
Goto, 116, 127, 129, 136
Grammar as abstract program, 15
Grammar transformation, 37, 45, 61, 77, 85
Guard, 7, 38, 40, 64, 122, 145

H

Homomorphism
  from symbols to functions, 27
  from symbols to languages, 17

I

Imperative programming, 12, 34, 38, 65, 119
Inherited attributes, 92, 105
Ini, 116, 127
Interpretation of grammars, 15, 29
Interpretative view, 37, 61, 125

L

LALR(k), 129
Lambek type, 23
  used for attribute functions, 92
  used for grammar transformations, 47, 55
Landmarks, 42, 135
Left corner, 58, 61, 64
  definition of, 52
Left recursion, 35, 40, 45, 49, 55, 66, 110
LL(k), 40
Look ahead
  recursive ascent, 128
  recursive descent, 38
LR parsing, 115

M

Memoization, xiv, 35, 61, 83, 139
  definition of, 35
  in recognizers, 35, 68
  to obtain parse forest, 81

N

Natural language, 143
  attribute evaluation and, 93, 95, 106, 148
  trees in, 31
Natural semantics/interpretation
  of context-free grammars, 7, 15, 29–30
  of EBNF grammars, 70
Nondeterminism, 8, 28, 39, 138, 145

P

Parse forest, 75, 81, 128
Parse matrix, 68
  definition of, 79
  filtered, 82
Parse stack, xiii, 123, 134–135
Predict items/prediction, 84, 86, 88
Product of languages, 16
Pseudo-circular attribute grammars, 99, 103, 148

R

Recursion stack, 123, 134, 150
Reduce(shift)-reduce conflict, 120
Rewriting, 3, 20, 29
Rule application, 21, 78

S

Singleton property of bunches, 7–8
Skip, 135
State, 58, 115
Synthesized attributes, 92, 105, 108

T

Tautology, 23, 47, 55, 144
Type identity, 32, 92