Linguistic Parsing and Program Transformations

M.J. Nederhof

Linguistic Parsing and Program Transformations

M.J. Nederhof

Linguistic Parsing and Program Transformations

een wetenschappelijke proeve op het gebied
van de
Wiskunde en Informatica

Proefschrift

ter verkrijging van de graad van doctor
aan de Katholieke Universiteit Nijmegen,
volgens besluit van het College van Decanen
in het openbaar te verdedigen op
maandag 24 oktober 1994
des namiddags te **1.30 uur** precies

door

Mark Jan Nederhof

geboren op 23 oktober 1966 te Leiden

Promotor: Prof. C.H.A. Koster

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Nederhof, Mark Jan

Linguistic parsing and program transformations / Mark Jan
Nederhof. - [S.l. : s.n.]. - Ill.
Proefschrift Nijmegen. - Met lit. opg.
ISBN 90-9007607-7
Trefw.: computerlinguïstiek.

Preface

With the exception of Chapter 1, the chapters in this thesis consist of previously published material which has been revised according to new insights of the author and the requirements of continuity of this thesis.

The chapters originate from the following papers:

Chapter 2 This chapter is a slightly modified version of

[Ned93a] M.J. Nederhof. Generalized left-corner parsing. In *Sixth Conference of the European Chapter of the Association for Computational Linguistics, Proceedings of the Conference*, pages 305–314, Utrecht, The Netherlands, April 1993.

Chapter 3 Except Section 3.8, this chapter is a slightly improved version of

[Ned93b] M.J. Nederhof. A multidisciplinary approach to a parsing algorithm. In K. Sikkel and A. Nijholt, editors, *Natural Language Parsing: Methods and Formalisms*, Proc. of the sixth Twente Workshop on Language Technology, pages 85–98. University of Twente, 1993.

A shortened version has appeared as

[Ned94b] M.J. Nederhof. An optimal tabular parsing algorithm. In *32nd Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 117–124, Las Cruces, New Mexico, USA, June 1994.

Chapter 4 The main part of the text has appeared as

[NS93b] M.J. Nederhof and J.J. Sarbo. Increasing the applicability of LR parsing. In *Third International Workshop on Parsing Technologies*, pages 187–201, Tilburg (The Netherlands) and Durbuy (Belgium), August 1993.

except Section 4.3.2, which has been taken from

[NS93c] M.J. Nederhof and J.J. Sarbo. Increasing the applicability of LR parsing. Technical report no. 93–06, University of Nijmegen, Department of Computer Science, March 1993.

Chapter 5 Except the introduction and Section 5.7, the text has previously appeared as

[NK93] M.J. Nederhof and C.H.A. Koster. Top-down parsing for left-recursive grammars. Technical report no. 93–10, University of Nijmegen, Department of Computer Science, June 1993.

A shortened version has appeared as

[Ned93c] M.J. Nederhof. A new top-down parsing algorithm for left-recursive DCGs. In *Programming Language Implementation and Logic Programming, 5th International Symposium*, Lecture Notes in Computer Science, volume 714, pages 108–122, Tallinn, Estonia, August 1993. Springer-Verlag.

which also contains parts of the introduction of this chapter.

Chapter 6 The text has previously appeared as

[NS93a] M.J. Nederhof and J.J. Sarbo. Efficient decoration of parse forests. In H. Trost, editor, *Feature Formalisms and Linguistic Ambiguity*, pages 53–78. Ellis Horwood, 1993.

An earlier version of the text in a less mature form has appeared as

[DNS92] C. Dekkers, M.J. Nederhof, and J.J. Sarbo. Coping with ambiguity in decorated parse forests. In *Coping with Linguistic Ambiguity in Typed Feature Formalisms*, Proceedings of a Workshop held at ECAI 92, pages 11–19, Vienna, Austria, August 1992.

Chapter 7 The text has previously appeared as

[NK92] M.J. Nederhof and K. Koster. A customized grammar workbench. In J. Aarts, P. de Haan, and N. Oostdijk, editors, *English Language Corpora: Design, Analysis and Exploitation, Papers from the thirteenth International Conference on English Language Research on Computerized Corpora*, pages 163–179, Nijmegen, 1992. Rodopi.

Some paragraphs describing ongoing research have been updated. An earlier version which puts more emphasis on incremental evaluation has appeared as

[NKDvZ92] M.J. Nederhof, C.H.A. Koster, C. Dekkers, and A. van Zwol. The Grammar Workbench: A first step towards lingware engineering. In W. ter Stal, A. Nijholt, and H.J. op den Akker, editors, *Linguistic Engineering: Tools and Products*, Proc. of the second Twente Workshop on Language Technology, pages 103–115. University of Twente, April 1992. Memoranda Informatica 92-29.

Acknowledgements

Since 1990, the author of this thesis is being supported by the Dutch Organisation for Scientific Research (NWO), under grant 00-62-518 (the STOP-project: "Specification and Transformation of Programs").

I would like to thank my supervisor Kees Koster for encouragement and support.

For fruitful co-operation, I owe many thanks to the co-authors of the papers which resulted in chapters of this thesis, viz. Janos Sarbo, Christ Dekkers and Kees Koster. I am also grateful to the editor of [NS93a], Harald Trost, and to the editors of [NK92], Jan Aarts, Pieter de Haan and Nelleke Oostdijk.

No research can be performed without correspondence with colleagues working in related areas, for exchanging ideas, for receiving comments on one's papers, and for sharing knowledge of the literature. In particular, I am greatly indebted to Klaas Sikkel, Giorgio Satta, François Barthélemy, René Leermakers, Mikkel Thorup, and Eric Villemonte de la Clergerie, for many fruitful discussions.

I acknowledge valuable correspondence with John Carroll, Dick Grune, Mark Johnson, Bernard Lang, Baudouin Le Charlier, Katashi Nagao, Anton Nijholt, Jan Rekers, Yves Schabes, Ed Stabler, Masaru Tomita, Frédéric Voisin, Theo Vosse, and Denis Zampuniéris. The work in this thesis was further influenced by interesting discussions with Franc Grootjen, Hans van Halteren, Hans Meijer, and Arend van Zwol.

We received kind help from Job Honig, Theo Vosse, John Carroll, and Hans de Vreught in finding a practical grammar for testing our algorithms from Chapter 4 on. Franc Grootjen and Frank Nusselder have put tremendous effort in the successful compilation of the Grammar Workbench (Chapter 7).

I apologize to all those who have in some way contributed to this thesis but whose contributions have here not been properly acknowledged.

Contents

Chapter 1

Introduction

Development of reliable software may be achieved by *transformational programming* [Par90]. The starting-point is a formal specification of a task that should be performed by the software that is to be developed. This formal specification may be underspecified and is in general not directly executable. By applying a series of small *transformations* whose correctness is beyond doubt, the formal specification is gradually changed, until finally we obtain an executable (and hopefully efficient) program that is equivalent to the original specification in the sense that the task performed by the program is as was required by the original specification.

Textbooks such as [Par90] show that a very large amount of transformations may be needed to apply transformational programming to non-trivial tasks. In this thesis we investigate a relatively simple set of tasks, viz. given a grammar in a certain formalism, and given an input string, determine whether the input string is in the language described by the grammar (the *recognition problem*), and if so, determine the structure of the input with regard to the grammar (the *parsing problem*).

Note that a grammar can be seen as a formal specification of the tasks of recognition and parsing. The construction of recognizers and parsers from grammars can be investigated in the realm of transformational programming, provided this construction uses some well-founded program transformations. Examples of such transformations are provided by tabular parsing (Section 1.2) and lookahead (Section 3.1).

Because of the restriction to relatively simple tasks expressed in simple formalisms, the results we obtain are at first applicable to only a small area of computer science. We do however not exclude the possibility that the results presented in this thesis can be generalized to handle larger classes of problems. (In fact, it has been argued in [Par84, Par86, Par90] that parsing theory gives rise to interesting case studies for transformational program development.)

Beforehand I would like to inform the reader that the contents of this thesis are strongly motivated by practical considerations of natural language processing.

In the next section we present some notation which will be used in this chapter and the following ones. In Section 1.2 we give an overview of the literature of tabular parsing, which will play an important part in this thesis. That section also contains some ideas which have not been published before. An outline of the remainder of this thesis is given in Section 1.3.

1.1 Notation and terminology

In this chapter, and in Chapters 2, 3, 4 and 5 we use some notation and terminology for context-free grammars which is for the most part standard and which is summarized below.

A context-free grammar $G = (T, N, P, S)$ consists of two finite disjoint sets N and T of nonterminals and terminals, respectively, a start symbol $S \in N$, and a finite set of rules P. Every rule has the form $A \to \alpha$, where the left-hand side (lhs) A is an element from N and the right-hand side (rhs) α is an element from V^*, where V denotes $(N \cup T)$. P can also be seen as a relation on $N \times V^*$.

We generally use symbols $A, B, C, \ldots$ to range over N, symbols $a, b, c, \ldots$ to range over T, symbols X, Y, Z to range over V, symbols $\alpha, \beta, \gamma, \ldots$ to range over V^*, and symbols $v, w, x, \ldots$ to range over T^*. We let ϵ denote the empty string. A rule of the form $A \to \epsilon$ is called an *epsilon rule*.

The relation P is extended to a relation $\xrightarrow{G}$ on $V^* \times V^*$ as usual. We write $\to$ for $\xrightarrow{G}$ when G is obvious. The transitive closure of $\to$ is denoted by $\to^+$ and the reflexive and transitive closure is denoted by $\to^*$.

We define the relation $\lessdot$ between nonterminals such that $B \lessdot A$ if and only if $A \to B\alpha$ for some α. The transitive closure of $\lessdot$ is denoted by $\lessdot^+$, and the reflexive and transitive closure of $\lessdot$ is denoted by $\lessdot^*$, which is called the *left-corner relation*. We pronounce $B \lessdot^* A$ as "B is a left corner of A".

We distinguish between two cases of left recursion. The most simple case, which we call *plain left recursion*, occurs if there is a nonterminal A such that $A \lessdot^+ A$. The other case, which we call *hidden left recursion*, occurs if $A \to B\alpha$, $B \to^* \epsilon$, and $\alpha \to^* A\beta$, for some A, B, α, and β; the left recursion is "hidden" by the empty-generating nonterminal B. (An equivalent definition of hidden left recursion can be found in [Lee92b].)

A grammar is said to be *cyclic* if $A \to^+ A$ for some nonterminal A. Otherwise, a grammar is said to be *cycle-free*.

A nonterminal A is said to be *nullable* if $A \to^* \epsilon$. A nonterminal A is called a *predicate* if it is nullable and $A \to^* v$ only for $v = \epsilon$.[1]

1.2 Tabular parsing

The existing literature reveals a number of different but related methods to derive tabular parsing algorithms from non-tabular ones. In the following, we discuss these methods, one after the other.

The different variants of tabular algorithms described in this section are grouped according to the areas of the literature from which they originate, rather than according to algorithmic resemblance. We use the same notation and terminology as much as possible for all methods, in order to simplify comparison.

Valiant's algorithm [Har78] and similar algorithms in [GHR80, Ryt85] will be left out of consideration, because they are difficult to relate to the other approaches to tabular parsing, and because they are essentially recognition algorithms as opposed to parsing algorithms.

[1] The term "predicate" seems to have been used in this sense for the first time in [Kos71]. In previous publications we used the term "nonfalse" in place of "nullable", which was misleading because "nonfalse" has been used in [Knu71] in a slightly different meaning.

1.2.1 Graph-structured stacks

One way to describe tabular parsing is by means of *graph-structured stacks*. This notion was introduced in [Tom86, Tom87] for tabular LR parsing. Following [Tom88] we argue that graph-structured stacks can be used to realize any nondeterministic algorithm working on a stack.

1.2.1.1 Pushdown automata

The starting-point for the application of graph-structured stacks is a recognizer in the form of a (in general nondeterministic) *pushdown automaton* (we will later also discuss parsers in lieu of recognizers). A pushdown automaton (PDA) operates on a stack, while reading an input string $a_1 \ldots a_n$ from left to right. The current input position is indicated by the *input pointer*. The part of the input $a_{i+1} \ldots a_n$ after the current input position i is called the *remaining input*. A *configuration* of the automaton is a pair (δ, i) consisting of a stack δ and a position i of the input pointer. For a certain input string, the *initial configuration* $(X_{initial}, 0)$ consists of a stack formed by one fixed stack symbol $X_{initial}$, and of the input pointer pointing to the position just before the first symbol of the input string. The *final configuration* $(X_{initial}X_{final}, n)$ consists of a stack with a fixed stack symbol X_{final} on top, and of the input pointer pointing to the last symbol of the input. For technical reasons, we assume that the stack symbol $X_{initial}$ cannot be popped or pushed, and that X_{final} can only be pushed on top of $X_{initial}$.

We will assume that the allowable steps are described by a finite set of *transitions* of the form

$$X\alpha \overset{z}{\mapsto} XY$$

where α represents zero or more stack symbols, X and Y represent one stack symbol each, and z represents the empty string ϵ or a single terminal a.

The application of such a transition is described as follows. If the stack is of the form $\delta X\alpha$, then we may apply a transition $X\alpha \overset{z}{\mapsto} XY$, provided the input pointer points to an occurrence of a in the case that z is a (in the case that z is ϵ there is no extra condition). The result is the stack δXY, and the input pointer is shifted one position to the right if z is a (if z is ϵ then the input pointer is not shifted). Note that if $\alpha = \epsilon$, then the transition performs a push on the stack of one element Y, and if α consists of YZ, then the transition performs a pop of one element Z. Formally, for some fixed automaton and some fixed input, we define the relation $\vdash$ by $(\delta X\alpha, i) \vdash (\delta XY, i)$ if there is a transition $X\alpha \overset{\epsilon}{\mapsto} XY$, and $(\delta X\alpha, i) \vdash (\delta XY, i+1)$ if there is a transition $X\alpha \overset{a}{\mapsto} XY$ and $a = a_{i+1}$.

The recognition of a certain input is obtained if starting from the initial configuration for that input we can reach the final configuration by repeated application of transitions; formally, $(X_{initial}, 0) \vdash^* (X_{initial}X_{final}, n)$, where $\vdash^*$ denotes the reflexive and transitive closure of $\vdash$ (and $\vdash^+$ denotes the transitive closure of $\vdash$).

We define a subrelation $\models^+$ of $\vdash^+$ as: $(\delta, i) \models^+ (\delta\delta', j)$ if and only if $(\delta, i) \vdash (\delta\delta_1, i_1) \vdash \ldots \vdash (\delta\delta_m, i_m) = (\delta\delta', j)$, for some $m \geq 1$, where $|\delta_k| > 0$ for all k, $1 \leq k \leq m$. Informally, we have $(\delta, i) \models^+ (\delta', j)$ if configuration (δ', j) can be reached from (δ, i) without the bottom-most part δ of the intermediate stacks being affected by any of the transitions; furthermore, at least one element is pushed on top of δ. Note that $(\delta_1 X, i) \models^+ (\delta_1 X\delta', j)$

implies $(\delta_2 X, i) \models^+ (\delta_2 X \delta', j)$ for any δ_2, since the transitions do not address the part of the stack below X.

Example 1.2.1 Consider the PDA with the following transitions:

$$\begin{aligned}
\bot &\overset{a}{\mapsto} \bot\ 1 \\
1 &\overset{a}{\mapsto} 1\ 2 \\
2 &\overset{a}{\mapsto} 2\ 3 \\
3 &\overset{a}{\mapsto} 3\ 3 \\
\bot\ 1\ 2\ 3 &\overset{\epsilon}{\mapsto} \bot\ 1 \\
1\ 2\ 3\ 3 &\overset{\epsilon}{\mapsto} 1\ 2 \\
2\ 3\ 3\ 3 &\overset{\epsilon}{\mapsto} 2\ 3 \\
3\ 3\ 3\ 3 &\overset{\epsilon}{\mapsto} 3\ 3
\end{aligned}$$

As initial stack symbol we have $\bot$ and as final stack symbol we have 1. The attentive reader may notice that this automaton is an LR(0) recognizer for the grammar with the rules $A \to AAA$ and $A \to a$.

Recognition of input *aaaaa* may be achieved by three different sequences of transitions, one of which is

stack	remaining input
$\bot$	*aaaaa*
$\bot$ 1	*aaaa*
$\bot$ 1 2	*aaa*
$\bot$ 1 2 3	*aa*
$\bot$ 1 2 3 3	*a*
$\bot$ 1 2	*a*
$\bot$ 1 2 3	
$\bot$ 1	

□

1.2.1.2 Development of graph-structured stacks

As remarked before, pushdown automata may be nondeterministic. We must therefore first find a realization for the nondeterminism before pushdown automata can be implemented.

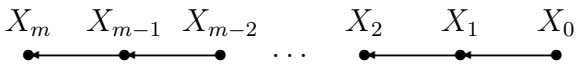
Let us first define a *search tree* for a certain PDA and input as follows. The root of the search tree represents the initial configuration. Each node in the tree has one son for each configuration derivable from the configuration of the father in one step. The input is accepted if at least one leaf of the search tree represents the final configuration.

A well-known realization of nondeterminism is backtracking. However, this is known to result in algorithms which are exponential in the length of the input in the worst case. This is because the size of the search tree is exponential, and because a backtracking recognizer in effect visits depth-first each node in the search tree, one after the other.

There is however a more efficient method of investigating the search tree. The essential observation is that the steps of a PDA do not depend on the stack elements that occur deep below the top of the stack. This means that two subtrees of the search tree could to some extent be processed simultaneously provided the stack elements that occur near the

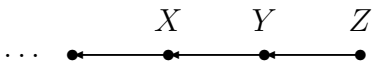
top of the stack are the same. Of course, if a number of stack elements is popped so that different stack elements are revealed for two search paths, then the computation needs to be performed separately again. This idea is realized by the *graph-structured stacks*, which we will develop gradually.

First, consider an alternative representation of a stack: a stack is not represented by a sequence of symbols $X_m \ldots X_0$ but by a number of nodes, representing the stack elements, and a number of arrows from one stack element to another just beneath it in the stack:

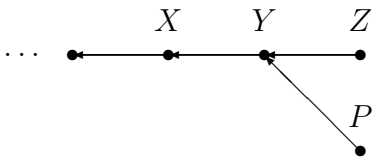


The next fase is to preserve a stack symbol and its associated arrows even if it is popped off the stack. In other words, the stack manipulations are performed non-destructively; only new nodes and arrows can be added, but none can be removed. The advantage is that if for some configuration the next step is not uniquely determined (i.e. the automaton is nondeterministic) then all possible next steps may be performed, one after the other, and in arbitrary order. Note that this is not allowed when a normal stack is used, since one step may pop the stack symbols needed for the application of another.

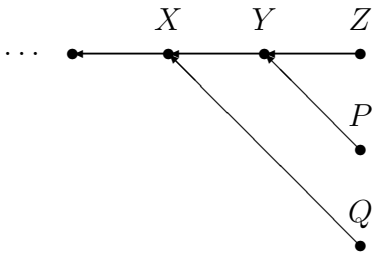
As an example, suppose at some point we have the stack



and suppose that we have both $Y\ Z \overset{\epsilon}{\mapsto} Y\ P$ and $X\ Y\ Z \overset{\epsilon}{\mapsto} X\ Q$. Then the first transition leads to



Subsequently, the second may be performed, which leads to[2]



In order to be able to record *when* the stack elements are conceptually on top of the stack we need to partition the nodes in the stack into the sets U_0, ..., U_n, for input of length n: a node belongs to U_i if it is conceptually on top of the stack after the i-th input symbol has been read and before the $(i+1)$-th is read.

We now have the following tentative realization of a nondeterministic pushdown automaton.

[2]Performing the two transitions in a different order leads to the same outcome.

Algorithm 1 (Naive graph-structured stack) For input $a_1 \dots a_n$, let the sets $U_1, \dots, U_n$ be all $\emptyset$. Create a node u labelled $X_{initial}$. Let $U_0 = \{u\}$. Perform the following as long as it is applicable.

1. Choose some i and some $XX_m \dots X_1 \overset{z}{\mapsto} XY$, with $z = \epsilon \vee z = a_{i+1}$, such that there is a path from some node in U_i to some node x in which the nodes are labelled $X_1, \dots, X_m, X$ in this order. The path must not have been treated before with the transition.
2. If $z = \epsilon$ then let $j = i$, else let $j = i + 1$.
3. Create a node y labelled Y.
4. Add y to U_j.
5. Create an arrow from y to x.

The input is accepted if there is at least one node labelled X_{final} in U_n.

This algorithm does not do much more than generalize backtracking: it allows the search tree to be investigated not only in depth-first order but also, for example, in breadth-first order. In terms of efficiency we have not gained much, since we still need to investigate each node of the search tree separately. The idea of sharing the computation of search paths which have the same few elements near the top of the stack can now however be realized very easily: for each U_i and each stack symbol, at most one node labelled with that stack symbol is henceforth added to U_i. We will also make sure that not more than one arrow will be between each pair of nodes. The result is

Algorithm 2 (Graph-structured stack) For input $a_1 \dots a_n$, let the sets $U_1, \dots, U_n$ be all $\emptyset$. Create a node u labelled $X_{initial}$. Let $U_0 = \{u\}$. Perform the following as long as it is applicable.

1. Choose some i and some $XX_m \dots X_1 \overset{z}{\mapsto} XY$, with $z = \epsilon \vee z = a_{i+1}$, such that there is a path from some node in U_i to some node x in which the nodes are labelled $X_1, \dots, X_m, X$ in this order. The path must not have been treated before with the transition.
2. If $z = \epsilon$ then let $j = i$, else let $j = i + 1$.
3. If there is a node labelled Y in U_j
   - then let y be that node,
   - else create a node y labelled Y. Add y to U_j.
4. If there is no arrow from y to x, then add such an arrow.

The input is accepted if there is a node labelled X_{final} in U_n.

I trust that the reader will be convinced that there is a one-to-one correspondence between paths from nodes in U_i to node u in U_0 on the one hand and normal linear stacks which can be reached by the PDA by reading the first i input symbols on the other. From this, the correctness of the above realization of pushdown automata immediately follows.

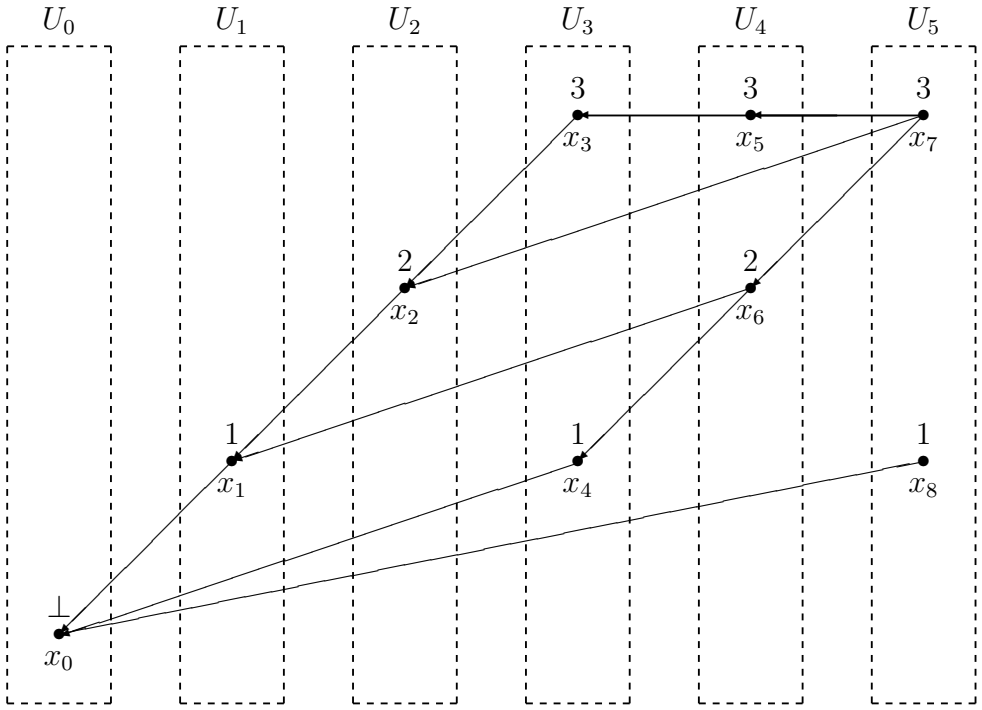


Figure 1.1: A graph-structured stack produced by Algorithm 2

Example 1.2.2 Consider again the automaton from Example 1.2.1. For the input *aaaaa* the result of Algorithm 2 is given by Figure 1.1. Note that, for instance, there is only one node labelled 1 in U_5, and only one arrow from x_8 to x_0, despite the fact that both transition $\perp\ 1\ 2\ 3 \overset{\epsilon}{\mapsto} \perp\ 1$ on path $x_7x_6x_1x_0$ and the same transition on path $x_7x_6x_4x_0$ may have caused a node labelled 1 in U_5 and an arrow from that node to x_0. □

Note that the sets $U_0, \ldots, U_n$ can be computed strictly from left to right, which we call *synchronous* computation. Other ways of computing elements in $U_0, \ldots, U_n$ we call *asynchronous* computation. Some advantages of one kind of computation over the other are discussed in [Bar93, pages 208–211]. In the next section we discuss one particular advantage of synchronous computation.

1.2.1.3 Synchronous computation

In Algorithm 2 we have not indicated how we record which paths have been treated, and which remain to be dealt with. For synchronous computation, such as presented for LR parsing in [NF91, Rek92], this task is simplified, although still the following problematic computation is required: if an arrow from some node $y \in U_i$ is added to the graph-structured stack, we must make sure that paths starting at some node in U_i and leading through that arrow will be treated subsequently. Finding all such paths, at least if implemented naively, is very expensive.

Therefore, a solution has been found which leads to a practical algorithm, although it is only applicable to pushdown automata which do not have any transitions of the form

$X \overset{\epsilon}{\mapsto} XY$. The absence of such transitions means that there are no steps which may push symbols on the stack without reading input. Consequently, there is never a non-trivial path between two nodes in the same set U_i, and if a new arrow is created from a node y in U_i to another node, then this arrow is the *first* arrow in any possible path starting at any node in U_i. This prevents the algorithm from having to follow arrows "backwards": we only have to look for paths which *start* with some new arrow, not for just any paths going through it.

This suggests the following way of keeping track of the untreated paths. We keep a set R_i of untreated arrows from some node in U_i. If an arrow r is added from some node y in U_i then r is added to R_i. Such an arrow r is removed from R_i if all non-trivial paths starting with r have been treated. For untreated paths of length zero we have a set A_i of untreated nodes in U_i. The result is the following algorithm.

Algorithm 3 (Restricted synchronous graph-structured stack) For input $a_1 \dots a_n$, let the sets $U_1, \dots, U_n$ and $R_1, \dots, R_n$ and $A_1, \dots, A_n$ be all $\emptyset$. Create a node u labelled $X_{initial}$. Let $U_0 = A_0 = \{u\}$. For $i = 0, 1, \dots, n$ in this order, perform one of the following two steps until $R_i = \emptyset$ and $A_i = \emptyset$:

1. (a) Take some node $x \in A_i$ labelled, say X, and remove it from A_i.
   (b) For each transition $X \overset{a_{i+1}}{\mapsto} XY$ do
      i. If there is a node labelled Y in U_{i+1}
         - then let y be that node,
         - else create a node y labelled Y. Add y to U_{i+1} and to A_{i+1}.
      ii. If there is no arrow from y to x, then add an arrow r' from y to x and add r' to R_{i+1}.
2. (a) Take some arrow $r \in R_i$, and remove it from R_i.
   (b) For each transition $XX_m \dots X_1 \overset{z}{\mapsto} XY$, with $m > 0$ and $z = \epsilon \vee z = a_{i+1}$, and for each path starting with arrow r and ending in some node x in which the nodes are labelled $X_1, \dots, X_m, X$ in this order, do
      i. If $z = \epsilon$ then let $j = i$, else let $j = i + 1$.
      ii. If there is a node labelled Y in U_j
         - then let y be that node,
         - else create a node y labelled Y. Add y to U_j and to A_j.
      iii. If there is no arrow from y to x, then add an arrow r' from y to x and add r' to R_j.

The input is accepted if there is a node labelled X_{final} in U_n.

This algorithm applied to LR parsing is the original algorithm from [Tom86]. Some attempts have been made in [Tom86] and [Kip91] to extend this algorithm to automata which do have transitions of the form $X \overset{\epsilon}{\mapsto} XY$.[3] The proposed solutions may introduce more than one node in each U_i with the same label. Intuitively, overlooking treatment of paths going

[3]Remark that for LR parsing such transitions represent reductions with epsilon rules. Chapter 4 discusses how such transitions may be avoided for grammars with epsilon rules.

partly through old arrows and partly through new arrows is avoided by sometimes keeping new and old arrows separate. However, this does not always work: the parsing process may loop for some automata, since new nodes with the same label in the same set U_i may be introduced indefinitely. For details we refer the interested reader to the original texts.

We can however find a solution which always works, even for asynchronous computation. The idea is that attached to each node x there is a set of tuples of the form $(XX_m \dots X_k, Y, j)$. Such a tuple indicates that there is some transition $XX_m \dots X_1 \overset{z}{\mapsto} XY$ and a path starting in some node in U_i and ending in x in which the nodes are labelled $X_1, \dots, X_{k-1}$, and where z satisfies the usual requirement with respect to i (and if $z = \epsilon$ then $j = i$, and if $z = a_{i+1}$ then $j = i + 1$). Intuitively, such a tuple gives the prospect of execution of the transition if ever we succeed in completing recognition of a path in which the nodes are labelled $X_1, \dots, X_m, X$. We continue this recognition as soon as an arrow is added from x to a node x' labelled X_k, and then $(XX_m \dots X_{k+1}, Y, j)$ is added to the set of tuples attached to x'. Actual execution of a transition can be performed when we obtain a tuple of the form (ϵ, Y, j). The reader may fill in the details for himself.

A similar idea has been proposed in [Lan74], which deals with the dynamic programming approach to tabular parsing (see also Section 1.2.2). This idea was further developed in [VdlC93], where it was called (translated into English) *currying*, as it is related to the notion of currying as known in the realm of functional programming. We will adopt the word "currying" for the method for graph-structured stacks outlines above.

An additional form of synchronous computation is described (for tabular LR parsing) in [PB91]: for a certain set U_i all arrows from some node in U_i are computed in a certain order. The purpose of this is to make sure that arrows are computed in all possible ways (see for example Steps 1.(b)(ii) and 2.(b)(iii) in Algorithm 3) before they are used for the application of a transition (i.e. taken from R_i in Step 2.(a)).

One way of achieving this[4] is to arrange each set R_i as a priority queue so that arrows to nodes in U_j are selected before arrows to nodes in $U_{j'}$, with $j' < j$, are selected. Furthermore, an arrow from a node labelled X is to be selected before an arrow from a node labelled Y is selected if there is a transition $ZX \overset{\epsilon}{\mapsto} ZY$ (this holds transitively).

The motivation for this kind of synchronous computation is that information may be attached to the arrows (see e.g. Section 1.2.1.5). Such information at a single arrow is gathered when that arrow is found in different ways (Steps 1.(b)(ii) and 2.(b)(iii) in Algorithm 3). The gathered information attached to the arrow may then be used when the arrow is considered as part of a path in Step 2.(b); but for this we may need to ensure that that arrow may not receive additional information henceforth, by being found in yet another way.

1.2.1.4 Parsing with cubic time complexity

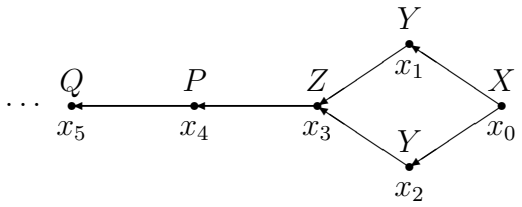
In a graph with m nodes, with at most one arrow between each pair of nodes, there may be at worst m^{p+1} paths of length p. This implies that if the longest transition of a pushdown automaton is $XX_p \dots X_1 \overset{z}{\mapsto} XY$, then there are $\mathcal{O}(n^{p+1})$ steps to be performed to process input of length n using Algorithm 2, since the number of nodes in a graph-structured stack

[4] As explained in [PB91] for the case of tabular LR parsing, in some cases this idea fails. In particular, cyclic grammars and some grammars with epsilon rules cause difficulties.

is $\mathcal{O}(n)$.

We may however optimize the application of the transitions in order to obtain a complexity of $\mathcal{O}(n^3)$, independent of p. The idea, described for LR parsing in [Kip91], is that memoizing is applied when a node is found to be reachable from a second node, following a path labelled with a certain sequence of stack symbols.

As an example, consider the following graph-structured stack, in which we look for paths from the node x_0 in which the nodes are labelled $X\ Y\ Z\ P\ Q$.



During the process of finding the path $x_0x_1x_3x_4x_5$ it may be stored in some table that there is a path from x_3 to x_5 in which the nodes are labelled $Z\ P\ Q$. When the algorithm then sets out to find $x_0x_2x_3x_4x_5$ starting from x_0 it may use that fact, so that the path $x_3x_4x_5$ does not need to be found a second time.

A similar approach which does not involve changing Algorithm 2 is to replace each transition $XX_m \dots X_1 \overset{z}{\mapsto} XY$ with $m > 2$ by the set of transitions

$$\begin{aligned}
X_3X_2X_1 &\overset{\epsilon}{\mapsto} X_3X_{1,2} \\
X_4X_3X_{1,2} &\overset{\epsilon}{\mapsto} X_4X_{1,2,3} \\
X_5X_4X_{1,2,3} &\overset{\epsilon}{\mapsto} X_5X_{1,2,3,4} \\
&\dots \\
X_mX_{m-1}X_{1,\dots,m-2} &\overset{\epsilon}{\mapsto} X_mX_{1,\dots,m-1} \\
XX_mX_{1,\dots,m-1} &\overset{z}{\mapsto} XY
\end{aligned}$$

where $X_{1,2}$, $X_{1,2,3}$, $\dots$, $X_{1,\dots,m-1}$ are fresh stack symbols.[5] Since now $p = 2$, the time complexity of Algorithm 2 is $\mathcal{O}(n^3)$. Note the similarity of this transformation to currying, discussed in Section 1.2.1.3.

The time complexity can also be measured in properties of the pushdown automata, assuming the length of the input to be fixed. This has been investigated in the realm of dynamic programming (see Section 1.2.2) in [Lan74], and specifically for LR parsing in [Joh91].

1.2.1.5 Parse forests

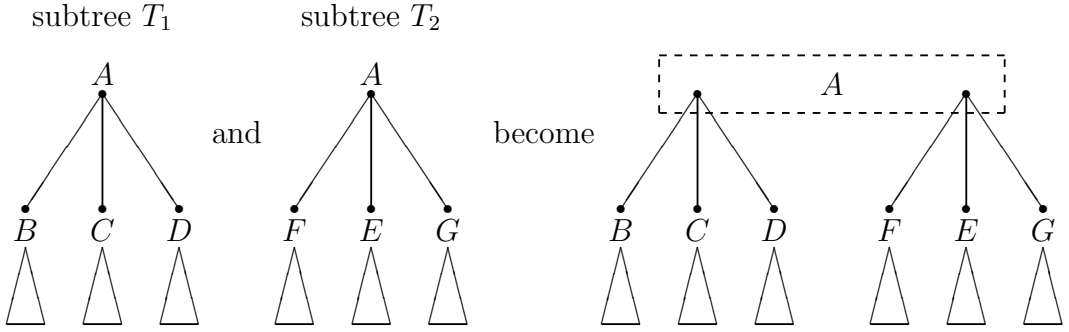
The algorithms discussed above are all recognition algorithms, since we have not indicated how parse trees or any other kind of value reflecting the structure of the input can be constructed by the algorithms as a side-effect of recognition. The obvious way to extend a pushdown automaton so that it becomes a parsing algorithm is to attach fragments of parse tree to the stack symbols. When a transition pops a number of symbols off the stack,

[5] For LR parsers, this transformation may also be described as a transformation of the grammar into "two normal form" [NS94].

the fragments attached to those symbols may be combined into a larger fragment of parse tree, which is attached to the stack symbol which is pushed on the stack.

When we realize such an automaton using a graph-structured stack then we come across the following problem. Assume that, at Step 3 of Algorithm 2, we have constructed a fragment of parse tree to be attached to a node labelled Y in U_j, although we already have a node labelled Y in U_j, which has his own fragment of parse tree. We now have to merge both fragments of parse tree in some way and attach the result to the unique node labelled Y in U_j. We call this merging of fragments of parse tree *packing*. (Sometimes this is more specifically called *local ambiguity packing*).

The best known form of packing is that of introducing *packed nodes* for the purpose of merging subtrees of a parse tree, as discussed in [Tom86]. As an example, suppose that we have two subtrees T_1 and T_2, whose roots are labelled with the same nonterminal A. Then we can introduce a new (packed) node, whose label is also A, and which has two sons, namely the roots of T_1 and T_2. We will indicate a packed node by a dashed box, containing its sons. For example:



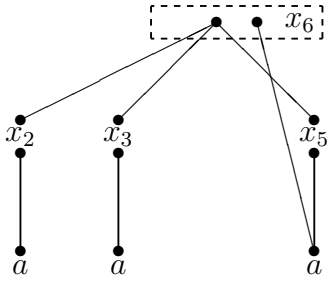
Packing of two subtrees can be generalized to packing of any number of subtrees. Other kinds of packing can be found in [Lei90] and [LAKA92], which discuss more compact representations allowing packing of prefixes and suffixes, respectively, of right-hand sides of rules.[6] We call a packed representation of all parses of a certain input a *parse forest*. (Sometimes, parse forests are more specifically called *shared, shared-packed*, or *packed shared* (parse) forests.) The first paper describing the ideas leading to parse forests seems to be [Kun65].[7] The first mature and fully explicit treatment of parse forests can be found in [CS70].

Packing is defective however if it is applied to Algorithm 2 as it is. The reason is that two fragments of parse tree may be packed even if they cover a different part of the input.

Example 1.2.3 To node x_6 of Figure 1.1 a packed node is attached, which packs two subtrees, one covering the second, third, and fourth symbol of the input, and the other covering only the fourth symbol of the input. If, for the sake of clarity, we identify nodes in the forest with the nodes of the graph-structured stack to which they are attached, then we have

[6] The computational aspects of normal packing versus packing of prefixes, suffixes, and possibly infixes of right-hand sides are given in an abstract way in [She76], in terms of *well formed constituent table (WFCT) parsers* versus *well formed state table (WFStT) parsers*.

[7] This paper is predated by [BHPS64], which suggests a more general construction of parse forests, but in a less explicit way. See also Section 1.2.5.



□

One way to deal with this problem is to attach fragments of parse tree to the arrows in a graph-structured stack instead of to the nodes. In fact, this is done in the original algorithm in [Tom86]. This prevents defective packing, but regrettably also prevents some *desired* packing. This is why [Tom86] applies an unexpected optimization, in order to identify some arrows, resulting in the fragments of parse tree attached to those arrows to be packed. We will explain this aspect of the algorithm, but with the fragments of parse tree attached to nodes as usual, following [TN91b]. We first present our own solution, and then we explain how the solution is dealt with in [TN91b].

The underlying idea is that if an (unpacked) fragment of parse tree is attached to a node x in, say, U_i, and if there is an arrow from this node to some node y in, say, U_j, then this fragment represents a parse of the part of the input from position j to position i. If a new arrow would be added from x to some node y' in, say, $U_{j'}$, with $j' \neq j$, then we would need to pack two fragments of parse tree covering a different part of the input, which we consider to be defective. In order to prevent this, we do not create an arrow from x to y', but we introduce a new node x' in U_i, and create an arrow from x' to y', thus avoiding the need to pack incompatible fragments of parse tree.

By slightly modifying Algorithm 2 we obtain the algorithm below. The differences lie in the new definition of h in Step 1, and in the extra condition concerning U_h in Step 3.

Algorithm 4 (Graph-structured stack for parse forest) For input $a_1 \ldots a_n$, let the sets $U_1, \ldots, U_n$ be all $\emptyset$. Create a node u labelled $X_{initial}$. Let $U_0 = \{u\}$. Perform the following as long as it is applicable.

1. Choose some i and h and some $XX_m \ldots X_1 \overset{z}{\mapsto} XY$, with $z = \epsilon \vee z = a_{i+1}$, such that there is a path from some node in U_i to some node x in U_h in which the nodes are labelled $X_1, \ldots, X_m, X$ in this order. The path must not have been treated before with the transition.

2. If $z = \epsilon$ then let $j = i$, else let $j = i + 1$.

3. If there is a node labelled Y in U_j from which there are arrows to nodes in U_h

   - then let y be that node,
   - else create a node y labelled Y. Add y to U_j.

4. If there is no arrow from y to x, then add such an arrow.

The input is accepted if there is a node labelled X_{final} in U_n.

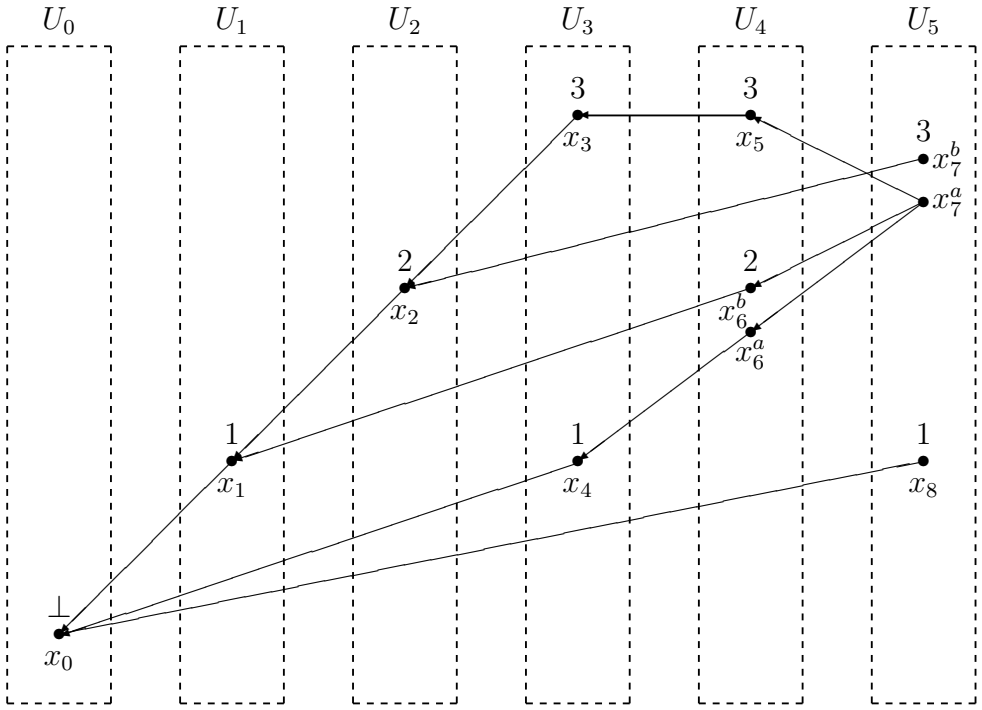


Figure 1.2: A graph-structured stack produced by Algorithm 4

Example 1.2.4 If we apply Algorithm 4 on the automaton from Example 1.2.1 and input *aaaaa*, then we obtain the graph-structured stack in Figure 1.2. Note the difference with the graph-structured stack from Figure 1.1: now more than one node with the same label may exist in the same set U_i. For example, there are two nodes labelled 3 in U_5; from x_7^a there are arrows to nodes in U_4, and from x_7^b there are arrows to nodes in U_2.

We make some reasonable assumptions about how a forest should be constructed as a side-effect of recognition:

> We first associate each input symbol with a fragment of parse tree consisting of just one node.
>
> Further, suppose we apply a transition $XX_m \dots X_1 \overset{z}{\mapsto} XY$ to a path, such that the fragments of parse tree attached to the nodes in the path are $T_1, \dots, T_m$, in this order (the fragment attached to the node labelled X is irrelevant). Then we create a fragment of parse tree consisting of a new node, of which the sons are the roots of $T_1, \dots, T_m$, in this order. If $z = a$ then the root has an $(m+1)$-th son which is the node corresponding to the appropriate input symbol. This fragment is attached to the new node to be created by this transition, or is packed with an already existing node.

This results in the forest from Figure 1.3. This parse forest still looks a little suspicious. For example, it seems strange that there are two subtrees at x_5 and x_6^a deriving in the same way the fourth a of the input. The reason for such anomalies is the LR parsing technique, on which the PDA from Example 1.2.1 is based. To overcome this aspect of LR parsing

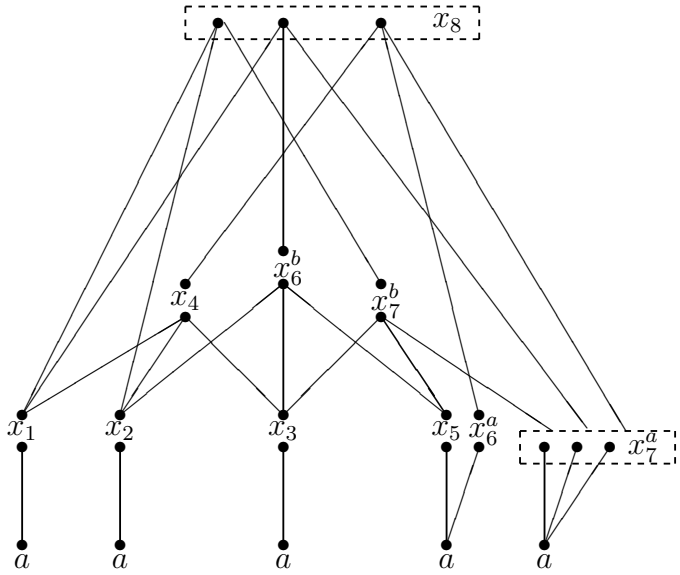


Figure 1.3: A parse forest produced by Algorithm 4

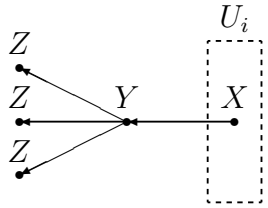
(also discussed in Chapters 2 and 3), [Rek92] has proposed that the computation of forests should be guided by the structure of the grammar, rather than by the structure of the PDA and its realization using a graph-structured stack.

The use of packing at x_7^a of three identical subtrees may be avoided by a trivial optimization, which can be applied for transitions reading input and only changing the stack by pushing one element. □

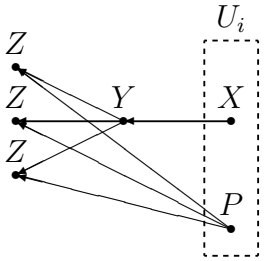
In Algorithm 4 we may have that there are a number of nodes (at worst n) with the same label in the same set U_i. This means that there may be at worst n paths ending in these nodes where in Algorithm 2 there was only one such path. Therefore, the time complexity is slightly worse now, viz. $\mathcal{O}(n^{p+2})$, if $p+1$ is the number of stack symbols in the left-hand side of the longest transition. By introducing an optimization allowing the nodes with the same label in the same set U_i to have a common representation with regard to incoming arrows, we can again achieve a time complexity of $\mathcal{O}(n^{p+1})$.

In Algorithm 4 we needed to investigate to which U_h a certain node x belonged. For some obscure reason, in [Tom86, TN91b] this was considered not possible. Therefore, a different solution was used in order to avoid packing of incompatible fragments of parse tree, which regrettably is too weak to achieve the same amount of packing as our Algorithm 4 does. The main idea behind the algorithms in [Tom86, TN91b] is that we make sure from the start that all arrows from a fixed node x end in nodes in a single set U_j. (This invariant is actually a direct consequence of our algorithm above, but here it is achieved more implicitly.) If at some point we need to add a number of arrows ending in a number of nodes, and we know that there are arrows to those nodes from one single node x, then we may conclude that the aforementioned nodes are all in a single set U_j, and this may be used to maintain the invariant.

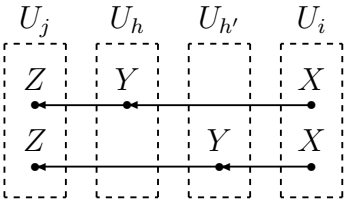
For example, suppose we have the graph-structured stack



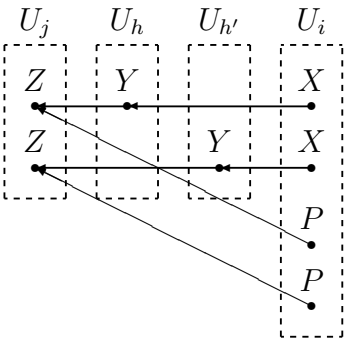
Now we want to apply the transition $Z\ Y\ X \overset{\epsilon}{\mapsto} Z\ P$. We may assume that all three nodes labelled Z are in the same U_j, because we have a node (viz. the one labelled Y) from which there are arrows to all three of them. We may therefore safely create a single node labelled P in U_i and add arrows from that node to the three nodes labelled Z, thus securing the invariant:



It is obvious that this trick does not always work. For example, if we have a graph-structured stack of the form



then after two applications of $Z\ Y\ X \overset{\epsilon}{\mapsto} Z\ P$ we obtain



where we have two nodes in U_i labelled P, instead of one as desired. Consequently, the packing here is not optimal.

1.2.2 Dynamic programming

The graph-structured stacks provide a good intuition of tabular parsing based on pushdown automata. However, the invention of such stacks was predated by many years by a similar

technique presented in [Lan74], which was later to be known as an application of *dynamic programming* to parsing [BL89].

The dynamic programming technique from [Lan74] is similar to the graph-structured stacks. In both cases the starting-point is a stack automaton. One important difference is that the stack automata from [Lan74] are pushdown transducers, which means that each transition may not only manipulate the stack and move the input pointer, it may also append zero or more output symbols to an output string. This requires extending tabular recognition to tabular parsing, as will be explained in Section 1.2.2.2. Further, the pushdown transducers also make use of *states*, which are redundant however,[8] and will therefore not be discussed further.[9]

1.2.2.1 Recognition

In order to explain the dynamic programming approach to recognition, we first remark that in the graph-structured stacks produced by Algorithm 2, an arrow from a node labelled X in U_i to a node labelled Y in U_j can be uniquely identified by a tuple (Y, j, X, i). The reason for this is that in each set U_i there is at most one node with a certain label. We call a tuple of the form (Y, j, X, i) an *item*. Further we remark that, except possibly for the initial node u in U_0, there are no nodes not connected to any arrows. This means that nodes are redundant, and that we can represent a graph-structured stack by only a set of items.

If we introduce the artificial item $(\bot, \bot, X_{initial}, 0)$ in order to represent the node u in U_0, then we can reformulate Algorithm 2 as follows:[10]

Algorithm 5 (Dynamic programming) Assume the input is $a_1 \dots a_n$. Let the set U be $\{(\bot, \bot, X_{initial}, 0)\}$. Perform the following as long as it is applicable.

1. Choose some i and some $XX_m \dots X_1 \overset{z}{\mapsto} XY$, with $z = \epsilon \vee z = a_{i+1}$, such that
   - if $m > 0$, there is a sequence (X, i_0, X_m, i_m), $(X_m, i_m, X_{m-1}, i_{m-1})$, $\dots$, (X_3, i_3, X_2, i_2), $(X_2, i_2, X_1, i) \in U$, not treated before with the transition, or such that
   - if $m = 0$, there is $(Z, h, X, i_0) \in U$ with $i_0 = i$, for some Z and h, such that i has not been treated before with the transition.
2. If $z = \epsilon$ then let $j = i$, else let $j = i + 1$.
3. Add item (X, i_0, Y, j) to U if it is not already there.

The input is accepted if $(X_{initial}, 0, X_{final}, n) \in U$.

[8] The states can be encoded into the stack symbols.

[9] The kinds of transition that the pushdown transducers from [Lan74] allow in terms of how each may manipulate the stack are also a little different from those proposed in Section 1.2.1.1. This however bears little relevance to the relation between graph-structured stacks and dynamic programming.

[10] It may be noted that in [Lan74] the set of items U in Algorithm 5 is partitioned into the sets $U_0, \dots, U_n$ as before. This however has little theoretical relevance. It is merely an optimization in order to allow efficient synchronous parsing (see Section 1.2.1.3).

For some pushdown automata we may simplify this algorithm so that items of the form (i, Y, j) are used instead of items of the form (X, i, Y, j); in other words, the X is redundant: it can be determined from other items in the table, when needed.

Before we can discuss the automata which allow this simplification, we first consider the formal meaning of an item (X, i, Y, j): an item (X, i, Y, j), $X \neq \bot$, is eventually added to U by Algorithm 5 if and only if

1. $(X_{initial}, 0) \vdash^* (\delta X, i)$, for some δ, and

2. $(X, i) \models^+ (XY, j)$.

The first condition states that some configuration can be reached from the initial configuration by reading the input up to position i, and in this configuration, an element labelled X is on top of the stack.

The second condition, which is most important in this discussion, states that if a stack has an element labelled X on top then the pushdown automaton can, by reading the input between i and j and without ever popping X, obtain a stack with one more element, labelled Y, which is on top of X. If we add 3-tuple items (i, Y, j) to U, instead of 4-tuple items (X, i, Y, j), we lose the information about which element Y is pushed on; we merely know that $(X', i) \models^+ (X'Y, j)$, for some X' with $(X_{initial}, 0) \vdash^* (\delta X', i)$.

The automata which allow tabular realization using 3-tuple items instead of 4-tuple items can more elegantly be characterized if we change the form of allowed transitions. In Section 1.2.1.1 we assumed that all transitions had the form $X\alpha \overset{z}{\mapsto} XY$, and most tabular algorithms we have seen until now can indeed be expressed most elegantly if transitions have that form. However, for the remaining part of this section it is more convenient if transitions have one of the following two forms: either $\alpha \overset{z}{\mapsto} Y$, with $\alpha \neq \epsilon$, or $X \overset{z}{\mapsto} XY$; in other words, the allowed transitions either replace a number of stack symbols by another, or push a stack symbol. It is straightforward to see that the class of accepted languages does not change if automata should be defined using these new kinds of transition.

We introduce the following auxiliary notion. We say a stack symbol X *springs from* Y if $(Y, i) \vdash^* (X, j)$ for some input $a_1 \ldots a_n$, and $0 \leq i \leq j \leq n$; informally, a stack element X can come into being by first having a Y which is replaced by X after performing a sequence of transitions. Note that $(Y, i) \vdash^* (X, j)$ implies $(\delta Y, i) \vdash^* (\delta X, j)$ for any δ.

The condition which an automaton should satisfy if 3-tuple items (i, Y, j) are to be used instead of 4-tuple (X, i, Y, j) is given by the following.

Property 1 (Context-independence) A pushdown automaton is called *context-independent* if the following holds. If X, Y, Y', z_1 are such that

1. there is a transition of the form $\alpha XY\beta \overset{z_2}{\mapsto} Z$, and

2. Y springs from Y', and

3. there is a transition of the form $X' \overset{z_1}{\mapsto} X'Y'$

then we have that $X \overset{z_1}{\mapsto} XY'$ is a transition of the automaton.

The rationale behind this property is as follows. Suppose that we want to apply a transition $\alpha XY\beta \overset{z_2}{\mapsto} Z$. Then one of the things we need to verify is whether $(X, i, Y, j) \in U$ for some i and j (see Algorithm 5). When using 3-tuple items we may only establish $(i, Y, j) \in U$, and $(h, X, i) \in U$ for some h.[11] This is translated into 4-tuple items as that there are items $(X', i, Y, j) \in U$ and $(W, h, X, i) \in U$, for some X' and W. We have to show next that we can derive from this that also $(X, i, Y, j) \in U$, on the assumption that the automaton is context-independent. We do this as follows.

From $(W, h, X, i) \in U$ we derive $(X_{initial}, 0) \vdash^* (\delta X, i)$, for some δ. From $(X', i, Y, j) \in U$ we derive $(X', i) \models^+ (X'Y, j)$. The first step in this sequence is the application of, say, transition $X' \overset{z_1}{\mapsto} X'Y'$, for some Y' which Y springs from. So we have $(X', i) \vdash (X'Y', i')$, where $i' = i$ if $z_1 = \epsilon$, and $i' = i + 1$ if $z_1 = a$ (then $a = a_{i+1}$). We further have $(Y', i') \vdash (Y, j)$ because of the definition of $\models$. The context-independence now informs us that also $(X, i) \vdash (XY', i')$, because we may assume the existence of transition $X \overset{z_1}{\mapsto} XY'$. This can be combined with $(Y', i') \vdash (Y, j)$ to give $(X, i) \models^+ (XY, j)$. Together with $(X_{initial}, 0) \vdash^* (\delta X, i)$ this gives us enough information to conclude $(X, i, Y, j) \in U$.

Note that it is decidable whether Property 1 holds for a given automaton, contrary to a similar property given in [VdlC93, Definition 9.2.3 on page 155] for a more general class of stack automata. The definition of "springs from" can even be simplified so as to ease the proof obligation of Property 1 as follows. We define the relation $\rightsquigarrow$ by $X \rightsquigarrow Y$ if there is a transition $X\alpha \overset{z}{\mapsto} Y$. We now say X *weakly springs from* Y if $Y \rightsquigarrow^* X$ (where $\rightsquigarrow^*$ is defined to be the reflexive and transitive closure of $\rightsquigarrow$). For automata which do not contain any useless transitions (i.e. those which cannot be used in recognition of any input), "weakly springs from" is equivalent to "springs from"; in general "X springs from Y" implies "X weakly springs from Y".

For a context-independent automaton we can simplify Algorithm 5 to the following.

Algorithm 6 (Simplified dynamic programming) Assume the input is $a_1 \dots a_n$. Let the set U be $\{(\bot, X_{initial}, 0)\}$. Perform the following as long as it is applicable.

1. Choose some i and
   - some $X_m \dots X_1 \overset{z}{\mapsto} Y$, with $z = \epsilon \vee z = a_{i+1}$, such that there is a sequence $(i_0, X_m, i_m), (i_m, X_{m-1}, i_{m-1}), \dots, (i_3, X_2, i_2), (i_2, X_1, i) \in U$, not treated before with the transition, or
   - some $X \overset{z}{\mapsto} XY$, with $z = \epsilon \vee z = a_{i+1}$, such that there is $(h, X, i_0) \in U$ with $i_0 = i$, for some h, such that i has not been treated before with the transition.
2. If $z = \epsilon$ then let $j = i$, else let $j = i + 1$.
3. Add item (i_0, Y, j) to U if it is not already there.

The input is accepted if $(0, X_{final}, n) \in U$.

Example 1.2.5 As an example of context-independent automata, we give the following construction of top-down recognizers in the style of recursive-descent parsers [Kos74, ASU86].

[11] Note that if we are using 4-tuple items then $(X, i, Y, j) \in U$ implies the existence of $(W, h, X, i) \in U$, for some W and h.

Let a context-free grammar be given. We *augment* this grammar by adding a rule $S' \to S$ and making the fresh symbol S' the new start symbol.

Construction 1 (Top-down) Construct the PDA with the transitions below. The stack symbols are $X_{initial} = \Box$ and symbols of the form $[A \to \alpha \bullet \beta]$, where $A \to \alpha\beta$; as X_{final} we take $[S' \to S \bullet]$.

$$
\begin{array}{lcll}
\Box & \overset{\epsilon}{\mapsto} & \Box\ [S' \to \bullet\ S] & \\
[A \to \alpha \bullet B\beta] & \overset{\epsilon}{\mapsto} & [A \to \alpha \bullet B\beta]\ [B \to \bullet\ \gamma] & \text{for } A \to \alpha B\beta, B \to \gamma \\
[A \to \alpha \bullet a\beta] & \overset{a}{\mapsto} & [A \to \alpha a \bullet \beta] & \text{for } A \to \alpha a\beta \\
[A \to \alpha \bullet B\beta]\ [B \to \gamma \bullet] & \overset{\epsilon}{\mapsto} & [A \to \alpha B \bullet \beta] & \text{for } A \to \alpha B\beta, B \to \gamma
\end{array}
$$

Context-independence (Property 1) of an automaton resulting from this construction can be argued as follows. Consider transitions of the form $[A \to \alpha \bullet B\beta]\ [B \to \gamma \bullet] \overset{\epsilon}{\mapsto} [A \to \alpha B \bullet \beta]$. The stack symbol $[B \to \gamma \bullet]$ (weakly) springs from a number of other symbols (with the dot at other positions of γ), but from those only $[B \to \bullet\ \gamma]$ occurs as Y' in a transition of the form $X' \overset{z_1}{\mapsto} X'Y'$, namely in transitions $[A' \to \alpha' \bullet B\beta'] \overset{\epsilon}{\mapsto} [A' \to \alpha' \bullet B\beta']\ [B \to \bullet\ \gamma]$. We now have that also $[A \to \alpha \bullet B\beta] \overset{\epsilon}{\mapsto} [A \to \alpha \bullet B\beta]\ [B \to \bullet\ \gamma]$, which indicates context-independence.

When we specialize Algorithm 6 to automata resulting from Construction 1, we obtain an asynchronous variant of Earley's algorithm [Ear70, BPS75, Har78, GHR80, KA89, Lei90, Leo91].

Algorithm 7 (Earley's algorithm) Assume the input is $a_1 \ldots a_n$. Let the set U be $\{(\bot, \Box, 0), (0, [S' \to \bullet\ S], 0)\}$. Perform one of the following steps as long as one of them is applicable.

predictor For some $(j, [A \to \alpha \bullet B\beta], i) \in U$ not considered before, add $(i, [B \to \bullet\ \gamma], i)$ to U (provided it is not already in U) for all $B \to \gamma$.

scanner For some $(j, [A \to \alpha \bullet a\beta], i) \in U$ not considered before, add $(j, [A \to \alpha a \bullet \beta], i+1)$ to U, provided $a = a_{i+1}$.

completer For some pair $(h, [A \to \alpha \bullet B\beta], j), (j, [B \to \gamma \bullet], i) \in U$ not considered before, add $(h, [A \to \alpha B \bullet \beta], i)$ to U.

The input is accepted if $(0, [S' \to S \bullet], n) \in U$.

Context-independence seems to be a natural property of pushdown automata describing context-free recognition algorithms. Also left-corner and LR recognizers (see Chapters 2, 3 and 4) in the form of pushdown automata satisfy this property, although in the case of LR parsing, the notion "springs from" needs to be replaced by a different notion, to allow for the different kind of transition that LR parsing requires (the one defined in Section 1.2.1.1). □

For most of the remaining discussion of dynamic programming we will again use graph-structured stacks, instead of the alternative formulation using sets of items.

1.2.2.2 Parsing

In Section 1.2.1.5 we have argued that an algorithm using a graph-structured stack can be extended to construct a parse forest. The idea was that fragments of parse tree are associated with nodes in the graph-structured stack. In Example 1.2.4 we have made some reasonable assumptions about how a parse forest should be constructed. However, we may want to have more control over the exact form of a forest, e.g. in order to determine the labels of the leaves in the forest.

Such control over the construction of forests is possible using *pushdown transducers*. We assume that a transition of such an automaton is of the form

$$X\alpha \overset{z}{\mapsto}_{\delta} XY$$

where α, X, Y, and z are as usual, and δ is a string consisting of zero or more elements from an *output alphabet*. Following [Lan74], the execution of a pushdown transducer uses an *output string*, which is initially empty. If a transition is applied, then the specified string of output symbols is appended behind the output string.

However, our goal is to realize pushdown transducers by means of tabular parsing, which requires some form of packing of the output value, and this can only be done if the output value has a more structured form. For now we will assume that this form again consists of structures similar to parse forests. Such forests contain two kinds of nodes: the *packed nodes*, whose (unordered) sons represent alternative parses of the same part of the input string, and the *plain nodes*, whose (ordered) sons represent parses of consecutive parts of the input. The main difference with the parse forests from Section 1.2.1.5 is that the exact form is directly influenced by the transitions of the automaton: the yields of the subforests are determined by the output symbols of the applied transitions.

In order to be able to form correct output strings from the complete forest, we need to associate the subforests to the arrows of the graph-structured stack instead of to the nodes. We now have

Algorithm 8 (Graph-structured stack for transducer) For input $a_1 \dots a_n$, let the sets $U_1, \dots, U_n$ be all $\emptyset$. Create a node u labelled $X_{initial}$. Let $U_0 = \{u\}$. Perform the following as long as it is applicable.

1. Choose some i and some $XX_m \dots X_1 \overset{z}{\mapsto}_{b_1 \dots b_k} XY$, with $z = \epsilon \vee z = a_{i+1}$, such that there is a path from some node in U_i to some node x in which the nodes are labelled $X_1, \dots, X_m, X$ in this order. The path must not have been treated before with the transition.

2. Construct a subforest T as follows. Let $T_1, \dots, T_m$ be the subforests associated with the arrows in the abovementioned path. Create new nodes $R_1, \dots, R_k$, labelled with the output symbols $b_1, \dots, b_k$, respectively. Create a new node. The sons of this node are the roots of $T_1, \dots, T_m$, and $R_1, \dots, R_k$, in this order.

3. If $z = \epsilon$ then let $j = i$, else let $j = i + 1$.

4. If there is a node labelled Y in U_j

   - then let y be that node,

- else create a node y labelled Y. Add y to U_j.

5. If there is no arrow from y to x, then add such an arrow and associate T with this arrow. Otherwise, pack T with the subforest associated with the existing arrow from y to x.

The input is accepted if there is a node labelled X_{final} in U_n. The forest associated with the arrow from this node to u is the required representation of all parses of the input.

In [Lan74], the forest is given in a specific form, which does not require an artificial distinction between plain nodes and packed nodes: the nodes in the forest are represented by nonterminals of a new context-free grammar, the *output grammar*. These nonterminals correspond to the arrows in the graph-structured stack. That a plain node A has a number of sons $A_1 \dots A_m$ is indicated by a context-free rule $A \to A_1 \dots A_m$. The difference between plain and packed nodes is now merely that a packed node is represented by a nonterminal which occurs more than once as a lhs of a rule.

The advantages of using output grammars to represent parse forests lie in their theoretical properties. For example, the language generated by such a grammar consists exactly of the set of output strings.

Example 1.2.6 We can extend the pushdown automaton from Example 1.2.1 to be a pushdown transducer with transitions:

$$\begin{aligned}
\bot &\overset{a}{\mapsto}_b \bot\ 1\\
1 &\overset{a}{\mapsto}_b 1\ 2\\
2 &\overset{a}{\mapsto}_b 2\ 3\\
3 &\overset{a}{\mapsto}_b 3\ 3\\
\bot\ 1\ 2\ 3 &\overset{\epsilon}{\mapsto}_{)} \bot\ 1\\
1\ 2\ 3\ 3 &\overset{\epsilon}{\mapsto}_{)} 1\ 2\\
2\ 3\ 3\ 3 &\overset{\epsilon}{\mapsto}_{)} 2\ 3\\
3\ 3\ 3\ 3 &\overset{\epsilon}{\mapsto}_{)} 3\ 3
\end{aligned}$$

If applied to the input *aaaaa*, the pushdown transducer may yield one of the output strings *bbb*)*bb*), *bbbb*)*b*), or *bbbbb*)), the second of which is obtained as follows.

stack	remaining input	output string
$\bot$	*aaaaa*	ϵ
$\bot$ 1	*aaaa*	*b*
$\bot$ 1 2	*aaa*	*bb*
$\bot$ 1 2 3	*aa*	*bbb*
$\bot$ 1 2 3 3	*a*	*bbbb*
$\bot$ 1 2	*a*	*bbbb*)
$\bot$ 1 2 3		*bbbb*)*b*
$\bot$ 1		*bbbb*)*b*)

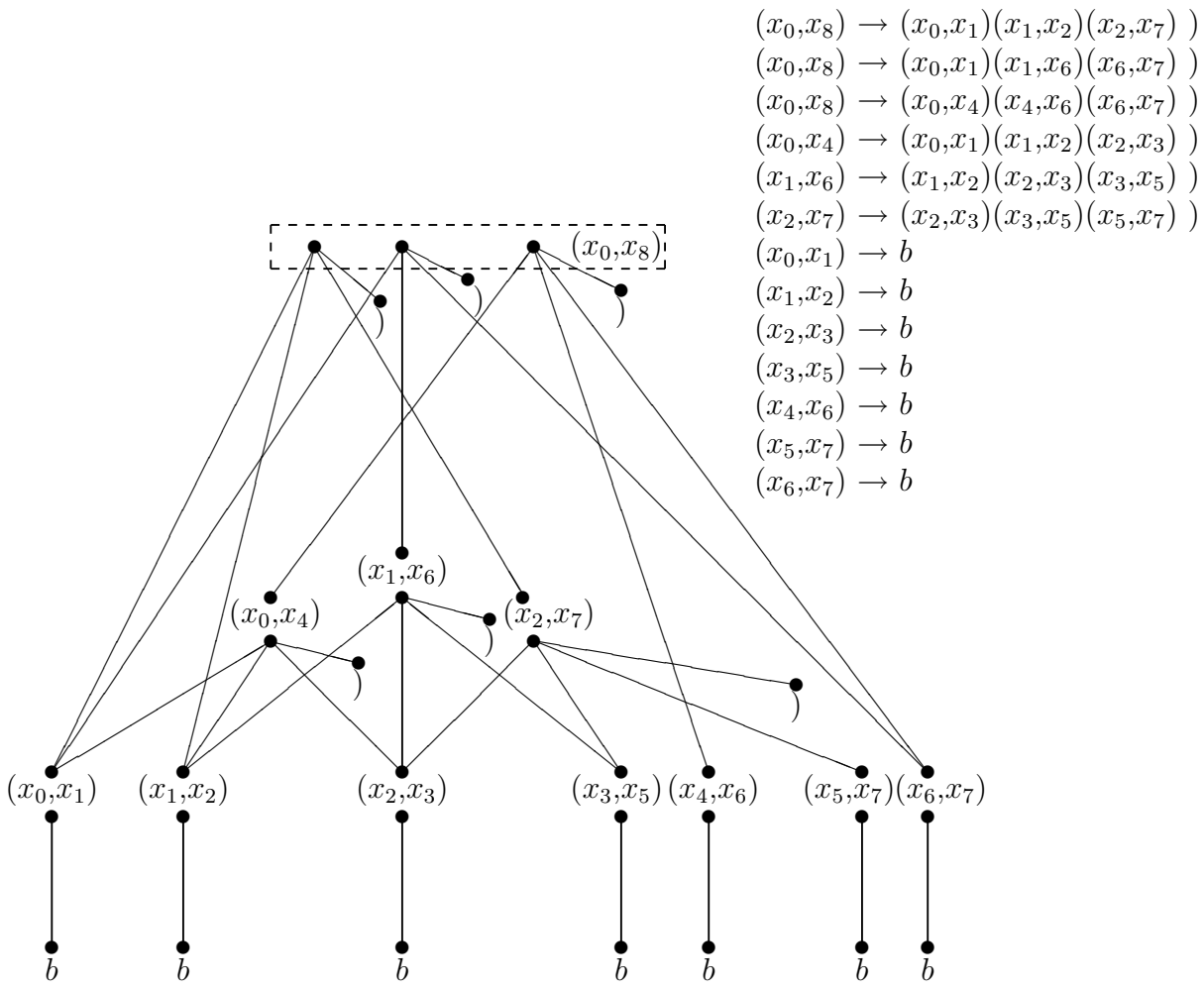


Figure 1.4: A parse forest produced by Algorithm 8 in two forms: as explicit graph and as an output grammar

Tabular execution leads to the graph-structured stack from Figure 1.1. If we use a tuple (y, x) to denote an arrow from node x to node y in the graph-structured stack, then the subforests associated with the arrows in the graph-structured stack can be given as in Figure 1.4. It is interesting to compare this forest with the one from Figure 1.3, which resulted from attaching fragments of parse forest with *nodes* in the graph-structured stack from Figure 1.2, instead of with *arrows* in the graph-structured stack from Figure 1.1.

Observe that the language generated by the grammar in Figure 1.4 is the set of strings $\{bbb)bb), bbbb)b), bbbbb))\}$, which is exactly the set of output strings which result from applying the pushdown transducer to the input *aaaaa* in all possible ways. □

1.2.2.3 Incomplete input

The dynamic programming approach to tabular parsing has been adapted to processing of incomplete input in [Lan88c]. An incomplete input string consists of an input string in which some symbols may be replaced by the special symbol "?", and whole substrings may be replaced by the special symbol "∗". A pushdown automaton must be adapted to

handling such symbols in the input as follows: if it encounters "?" then it may consider this symbol to be any other terminal symbol for the sake of determining the applicability of transitions; if it encounters "$*$" then it may furthermore do this while either shifting or not shifting the input pointer one position to the right.

Tabular realizations of such pushdown automata (or pushdown transducers) are straightforwardly obtained from those of usual pushdown automata. As an example we give the algorithm below. Without loss of generality, we assume that the input does not contain two adjacent occurrences of "$*$", and that the last symbol of the input is not "$*$" (if needed we could add an artificial endmarker).

Algorithm 9 (Graph-structured stack for incomplete input) As Algorithm 8, but the condition $z = \epsilon \vee z = a_{i+1}$ in Step 1, and the determination of j in Step 3, are now taken together in the following:

- Choose j such that

  1. $z = \epsilon \wedge j = i$, or
  2. $z = a \wedge (a_{i+1} = a \vee a_{i+1} = ?) \wedge j = i + 1$, or
  3. $z = a \wedge a_{i+1} = * \wedge j = i$, or
  4. $z = a \wedge a_{i+1} = * \wedge (a_{i+2} = a \vee a_{i+2} = ?) \wedge j = i + 2$.

  (If such a j does not exist, then this iteration of the algorithm fails.)

Since very often the extra nondeterminism involved in processing "?" and "$*$" leads to search trees of infinite size, we may safely say that other realizations of pushdown automata for incomplete input, for example using backtracking, are not useful in practice. Note that tabular parsers are capable of computing an infinitely large set of parse trees by computing a finite representation of such a set of trees in the form of a parse forest with cycles.

We say a pushdown transducer is *canonical* if the transitions are all of the form $X\alpha \overset{a}{\mapsto}_a XY$ or of the form $X\alpha \overset{\epsilon}{\mapsto}_\epsilon XY$; in other words, the output symbols are exactly the input symbols. As has been remarked in [Lan91a], for canonical pushdown transducers, output grammars have a particularly elegant property if used for parsing incomplete input: for an incomplete input string, the language of the output grammar is the set of all correct sentences which match the incomplete string.

For example, if a canonical pushdown transducer has been constructed from a certain grammar G, then parsing the completely unknown input "$*$" results in an output grammar generating the same language as G does; in general, parsing incomplete input results in an output grammar which is a specialization of G to the parts of the input that are known.

Example 1.2.7 Consider the following grammar G, generating the palindromes of even length over $\{a, b\}$.

$$\begin{aligned} S &\rightarrow a\ S\ a \\ S &\rightarrow b\ S\ b \\ S &\rightarrow \epsilon \end{aligned}$$

Using the LR(0) parsing technique, we may obtain the pushdown transducer below.[12] The set of stack symbols is $\{\bot, \top, a, b, \overline{a}, \overline{b}, \overline{\overline{a}}, \overline{\overline{b}}\}$. We assume the transducer is canonical, and may therefore write $\overset{z}{\mapsto}$ in lieu of $\overset{z}{\mapsto}_z$, without loss of information.

$$
\begin{array}{lcl}
\bot & \overset{a}{\mapsto} & \bot\ a \\
\bot & \overset{b}{\mapsto} & \bot\ b \\
a & \overset{a}{\mapsto} & a\ a \\
a & \overset{b}{\mapsto} & a\ b \\
b & \overset{a}{\mapsto} & b\ a \\
b & \overset{b}{\mapsto} & b\ b \\
\overline{a} & \overset{a}{\mapsto} & \overline{a}\ \overline{\overline{a}} \\
\overline{b} & \overset{b}{\mapsto} & \overline{b}\ \overline{\overline{b}}
\end{array}
\qquad
\begin{array}{lcl}
\bot & \overset{\epsilon}{\mapsto} & \bot\top \\
a & \overset{\epsilon}{\mapsto} & a\ \overline{a} \\
b & \overset{\epsilon}{\mapsto} & b\ \overline{b}
\end{array}
\qquad
\begin{array}{lcl}
\bot\ a\ \overline{a}\ \overline{\overline{a}} & \overset{\epsilon}{\mapsto} & \bot\top \\
a\ a\ \overline{a}\ \overline{\overline{a}} & \overset{\epsilon}{\mapsto} & a\ \overline{a} \\
b\ a\ \overline{a}\ \overline{\overline{a}} & \overset{\epsilon}{\mapsto} & b\ \overline{b} \\
\bot\ b\ \overline{b}\ \overline{\overline{b}} & \overset{\epsilon}{\mapsto} & \bot\top \\
a\ b\ \overline{b}\ \overline{\overline{b}} & \overset{\epsilon}{\mapsto} & a\ \overline{a} \\
b\ b\ \overline{b}\ \overline{\overline{b}} & \overset{\epsilon}{\mapsto} & b\ \overline{b}
\end{array}
$$

The graph-structured stack produced by Algorithm 9 on input $aab * a$ is given in Figure 1.5. Note that U_4 does not participate in the graph-structured stack. The arrows labelled $A_1 \ldots A_{15}$ are those which play a role in the construction of the final parse forest. This forest, in the form of an output grammar, is given by:

$$
\begin{array}{lll}
A_1 \to A_2\ A_3\ A_4 & & \\
A_2 \to a & & \\
A_3 \to A_5\ A_6\ A_7 & & \\
A_5 \to a & & \\
A_6 \to A_8\ A_9\ A_{10} & & \\
A_8 \to b & & \\
A_9 \to \epsilon & A_9 \to A_{11}\ A_9\ A_{10} & A_9 \to A_{12}\ A_{13}\ A_7 \\
A_{11} \to b & & \\
A_{12} \to a & & \\
A_{13} \to \epsilon & A_{13} \to A_{14}\ A_{13}\ A_7 & A_{13} \to A_{15}\ A_9\ A_{10} \\
A_{14} \to a & & \\
A_7 \to a & & \\
A_{15} \to b & & \\
A_{10} \to b & & \\
A_4 \to a & &
\end{array}
$$

If we apply substitution for the nonterminals which have only one defining rule, we obtain:

$$
\begin{array}{lll}
A_1 \to a\ a\ b\ A_9\ b\ a\ a & & \\
A_9 \to \epsilon & A_9 \to b\ A_9\ b & A_9 \to a\ A_{13}\ a \\
A_{13} \to \epsilon & A_{13} \to a\ A_{13}\ a & A_{13} \to b\ A_9\ b
\end{array}
$$

which is clearly such that it generates exactly the palindromes of even length, starting with aab and ending with a. This grammar can therefore be seen as a specialization of G. □

[12] We do not explain LR parsing here. To those familiar with LR parsing, we explain that the transitions in the first column represent the shifts, those in the second column reductions with $S \to \epsilon$, and those in the third column reductions with $S \to a\ S\ a$ and $S \to b\ S\ b$.

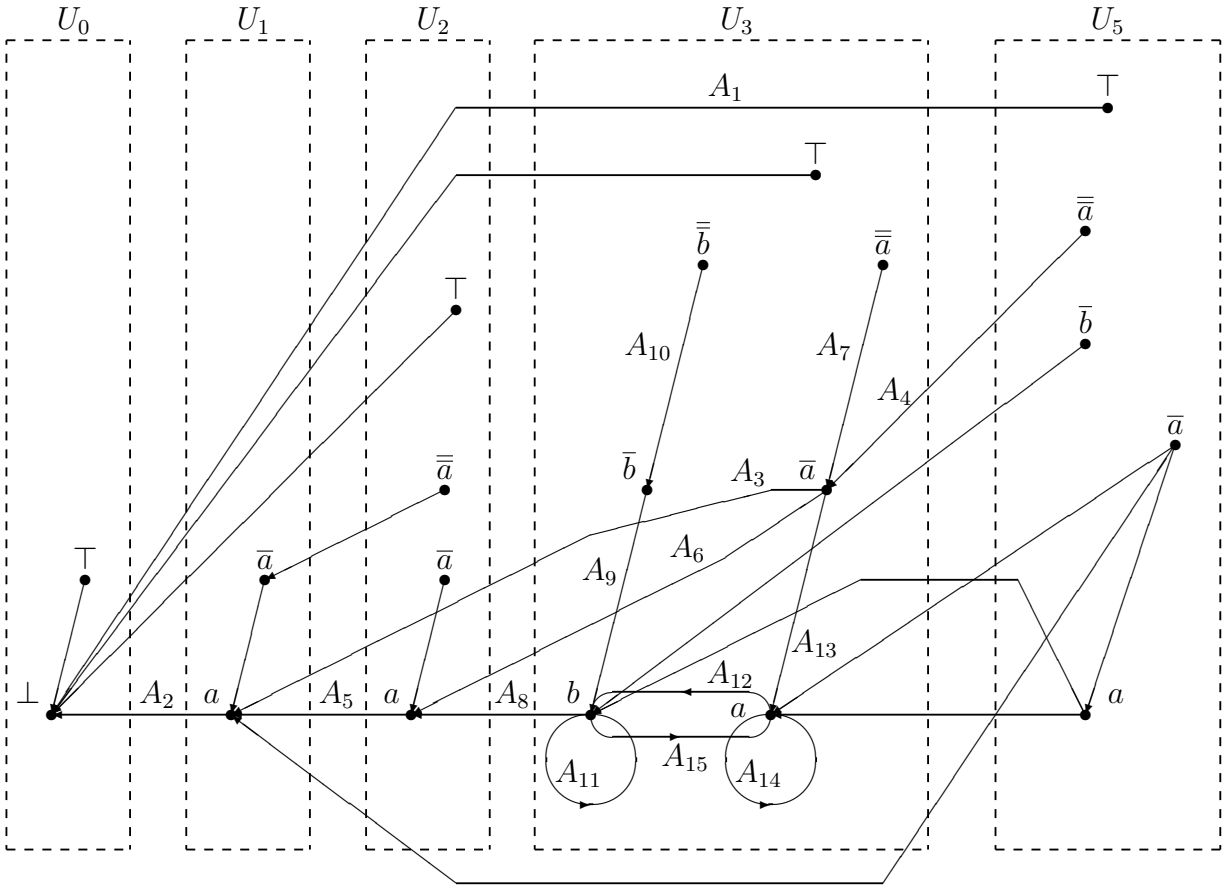


Figure 1.5: A graph-structured stack produced by Algorithm 9

Parsing of incomplete input using dynamic programming may be further generalized to handle word lattices [Lan89]. Word lattices are related to finite automata, and correspondingly, context-free parsing of a word lattice is reminiscent of computing the intersection of a regular language represented as a finite automaton and a context-free language [BHPS64].

1.2.2.4 Ignoring substrings

Processing of incomplete input as explained in the previous section allows automatic insertion of appropriate input symbols in an input string in order to make it correct. The opposite concept is the *elimination* of input symbols from the input string in order to make it correct. This is for example described in [LT93], for tabular LR parsing. For recognition, the general idea can be conveyed by giving a variant of Algorithm 2. This variant results when we replace the reference to U_i in the first step by "U_h, where $h = i$ if $z = \epsilon$ and $h \leq i$ if $z = a_{i+1}$". We then obtain:

1. Choose some i and some $XX_m \ldots X_1 \overset{z}{\mapsto} XY$, with $z = \epsilon \vee z = a_{i+1}$, such that there is a path from some node in U_h, where $h = i$ if $z = \epsilon$ and $h \leq i$ if $z = a_{i+1}$, to some node x in which the nodes are labelled $X_1, \ldots, X_m, X$ in this order. The path must not have been treated before with the transition and with i.

If $z = a_{i+1}$, then this step causes the input between positions h and i to be ignored. The reason that we let $h = i$ if $z = \epsilon$ is to avoid unnecessary nondeterminism; however, if we allow $h \leq i$ if $z = \epsilon$, then the algorithm still functions correctly.

The algorithm suggested above accepts the input if a subsequence of the input is a correct sentence. An extra condition is that the last symbol of this subsequence should be the last symbol of the input. This technical problem can be avoided by introducing an endmarker.

In order to obtain a tractable algorithm for finding the subsequences, it may be necessary to avoid ignoring substrings too often. One solution is to mark those input positions where symbols may or may not be ignored (cf. the special input symbol "$*$" from the previous section). Another solution is to apply some beam search heuristics [LT93, Lav94], in order to compute only a few of the longest subsequences which are correct sentences.

1.2.2.5 Bidirectional parsing

Algorithm 5 (dynamic programming) adds an item (X, i, Y, j), with $X \neq \bot$, to set U if and only if the following two conditions are satisfied for the pushdown automaton and some fixed input:

1. Some configuration can be reached from the initial configuration by reading the input up to position i. In this configuration, an element labelled X is on top of the stack. Formally, $(X_{initial}, 0) \vdash^* (\delta X, i)$.

2. If a stack has an element labelled X on top, then the PDA can, by reading the input between i and j and without ever popping X, obtain a stack with one more element, labelled Y, which is on top of X. Formally, $(X, i) \models^+ (XY, j)$.

The second condition is essential, and justifies taking the condition $(X_{initial}, 0, X_{final}, n) \in U$ as a criterion for recognition of the input. The first condition however only serves to improve the run-time efficiency: it is in general not useful to compute which elements may be pushed on top of some element labelled X starting at some position i of the input, if no stack with X on top does ever occur at i.

How the first condition is satisfied can be explained by investigating Step 1 of Algorithm 5 in the case $m = 0$, which can be simplified to:

> Choose some i and some $X \overset{z}{\mapsto} XY$, with $z = \epsilon \vee z = a_{i+1}$, such that there is $(Z, h, X, i) \in U$ for some Z and h, such that i has not been treated before with the transition.

The condition "there is $(Z, h, X, i) \in U$ for some Z and h" is what causes Condition 1 above to be satisfied. By leaving out this condition on the existence of such an item $(Z, h, X, i) \in U$, only Condition 2 above is sufficient (and required) for an item (X, i, Y, j) to be added to U.

Simplifying the algorithm in this way has two consequences:

- Some items (X, i, Y, j) may be added to U which cannot possibly be useful in computing any sequence of transitions leading from the initial to the final configuration, with respect to the input to the left of position i.

- It is no longer necessary to process the input from left to right; we can start the recognition process by applying a transition $X \overset{z}{\mapsto} XY$ at any position i in the input where $z = \epsilon \vee z = a_{i+1}$ is satisfied.

The first consequence seems only to deteriorate the time-complexity. However, this may be compensated for by the second consequence, which allows *head-driven parsing* [SS89, Kay89, BvN93, SodA93, NS94] or *island-driven parsing* [SS91, SS94]. In some areas of parsing namely, more efficient techniques are obtained if one puts particular emphasis on specific parts of an input string (island-driven parsing) or on specific parts of grammar rules (head-driven parsing). The reason may be that some parts of the input are considered more reliable and therefore more appropriate as a starting-point than other parts of the input, or that in some parts of a grammar rule some values are computed which may lead to more deterministic processing of the other parts.

Parsing algorithms which read the input not strictly from left to right or from right to left are called *bidirectional* parsing algorithms. The first bidirectional tabular parsing algorithm was described in [AHU68] (predating the dynamic programming algorithm from [Lan74]). This algorithm is almost the same as the simplified version of Algorithm 5 suggested above, with two minor differences. The first difference is that the pushdown automata are *two-way*, which means that transitions may also decrease instead of increase the input pointer. This constitutes a second and completely independent source of bidirectionality.

The second difference is that the transitions change the stack in a slightly different way. For those transitions it was more convenient to have items which indicate that "from position i to j an element X may be popped" instead of those in Algorithm 5 which indicate that "from position i to j an element Y may be pushed on top of an element X".[13] This difference seems however to have little theoretical significance.

The method we have outlined of deriving a bidirectional tabular algorithm from a single PDA is conceptually simpler than methods which require specialized automata for both directions [Sai90], since then one has to deal with the complicated issue of how to combine the results from parsing in one direction with those from parsing in the other direction.

1.2.2.6 Conditional parsing systems

A set of items U may be computed using a parallel realization of a dynamic programming algorithm: several processors are used, each of which computes a subset of U. If an unlimited number of processors is available, then the time complexity will however still be at least linear in the length of the input if we use Algorithm 5. This is explained by the fact that there is a left-to-right dependency for different subsets of U: some item $(Y, j, X, i) \in U$, $i > 0$, can only be computed after at least one item $(Y', j', X', i-1) \in U$ has been found.

In the previous section we discussed how this left-to-right dependency may be avoided, and this was therefore the first step towards parallel algorithms with a sublinear time complexity. However, we need one more step to achieve our aim. This will be explained below.

In order to keep the discussion simple, we will only allow transitions of the forms $XY \overset{z}{\mapsto} Z$ and $X \overset{z}{\mapsto} XY$. Applying the simplification for bidirectionality we discussed in the

[13] Yet other kinds of item are used in [Ryt82, Ryt85] and [BKR91, Section 6.3]; see also [KR88] and [BKR91, Exercise 7.28].

previous section, Algorithm 5 can now be rephrased as:

Algorithm 10 (Bidirectional dynamic programming) Assume the input is $a_1 \dots a_n$. Denote the set of all items by

$$\mathcal{I} = \{(Y, j, X, i) \mid 0 \leq j \leq i \leq n \wedge Y \text{ and } X \text{ are stack symbols}\}$$

In the following we do not write empty sequences of items. Sequences of items will be considered to be unordered. Define the binary relation $\gg \subseteq \mathcal{I}^* \times \mathcal{I}$ as the least relation satisfying:

- Let $(W, h, X, i'), (X, i', Y, i) \gg (W, h, Z, j)$
  for all $(W, h, X, i'), (X, i', Y, i), (W, h, Z, j) \in \mathcal{I}$ such that
  - there is a transition $XY \overset{z}{\mapsto} Z$, and
  - if $z = \epsilon$ then $j = i$, else $j = i + 1 \wedge z = a_{i+1}$.
- Let $\gg (X, i, Y, j)$ for each $(X, i, Y, j) \in \mathcal{I}$ such that
  - there is a transition $X \overset{z}{\mapsto} XY$, and
  - if $z = \epsilon$ then $j = i$, else $j = i + 1 \wedge z = a_{i+1}$.

Compute using least fixed-point iteration the set $U \subseteq \mathcal{I}$ defined to be the least set of items satisfying: if there are $I_1, I_2, \dots, I_m \in U$ such that $I_1, I_2, \dots, I_m \gg I$, then $I \in U$.

The input is accepted if $(X_{\mathit{initial}}, 0, X_{\mathit{final}}, n) \in U$.

For some PDAs, this algorithm can process input in less than linear time provided enough processors are used to compute U. (We do not address the issue of exactly how many processors are needed.) However, for some PDAs the set U cannot be computed by this algorithm in less than linear time, regardless of how many processors are available.

In order to explain this, consider some item I which is in U after performing Algorithm 10. We can construct a *composition tree* [dV93] for I as follows: the root is labelled with the item I itself, and each node in the tree labelled with I' has a sequence of sons labelled with $I_1, I_2, \dots, I_m$ such that $I_1, I_2, \dots, I_m \gg I'$. (This means that all leaves of the tree are labelled with some item I' for which $\gg I'$.) Note that for each item in U there may be more than one composition tree.

Using a parallel algorithm to compute U can be seen as computing the subtrees of composition trees in parallel, so that the label at the root of a composition tree is computed after d steps, where d is the depth of the composition tree.

It can be proved that if for each item we consider only the composition trees without cycles (i.e. no node has the same label as one of its descendants), then the sizes of the composition trees (measured in the number of nodes) are linear in the length n of the input. If all composition trees were perfectly balanced, then the depths would be logarithmic in n, and we could recognize all input in logarithmic time. However, in general, composition trees are not perfectly balanced and the depths can therefore be linear in n.

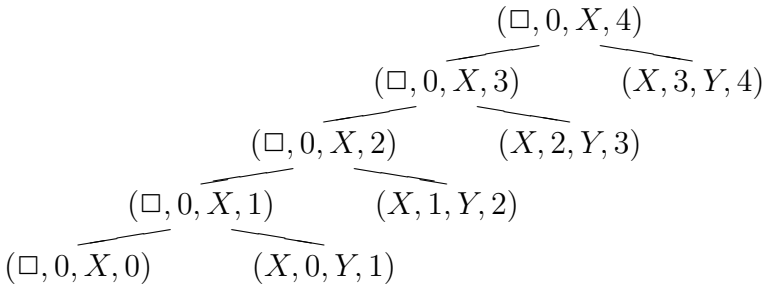


Figure 1.6: A composition tree for $(\Box, 0, X, 4)$

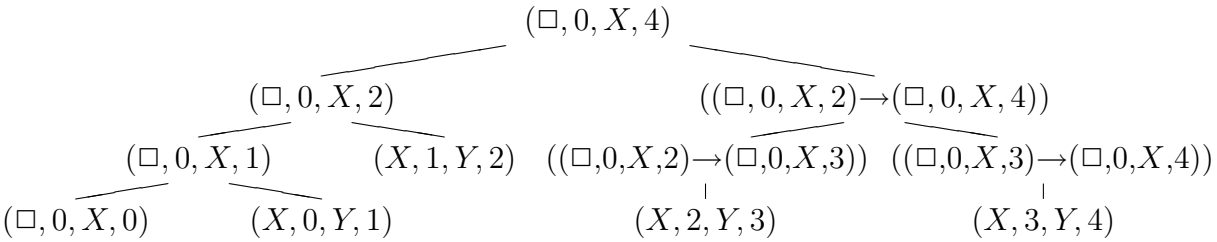


Figure 1.7: A composition tree for $(\Box, 0, X, 4)$ in a conditional parsing system

Example 1.2.8 Consider the PDA with the transitions below. The stack symbols are $X_{initial} = \Box$, $X_{final} = X$, and Y.

$$\begin{array}{lcl} \Box & \overset{\epsilon}{\mapsto} & \Box\ X \\ X & \overset{a}{\mapsto} & XY \\ XY & \overset{\epsilon}{\mapsto} & X \end{array}$$

For input $a_1 \ldots a_n = aa \ldots a$ we have

- $\gg (\Box, i, X, i)$ for $0 \leq i \leq n$
- $\gg (X, i, Y, i+1)$ for $0 \leq i < n$
- $(W, h, X, j), (X, j, Y, i) \gg (W, h, X, i)$ for $W \in \{\Box, X, Y\}$ and $0 \leq h \leq j \leq i \leq n$

The (only) composition tree for $(\Box, 0, X, 4)$ for input *aaaa* is given in Figure 1.6. It has depth 5, which means that if enough processors are available, this item can be computed by Algorithm 10 in 5 steps. In general, the number of steps needed to compute U for input of length n is $n + 1$. □

The idea leading to logarithmic computation of U in spite of the fact that composition trees may not be perfectly balanced, originates in Rytter's algorithm [BKR91, dV93], which can be seen as an adaptation of the CKY algorithm for fast parallel processing. In [Sik93, Section 14.8] this idea is formulated in terms of *conditional (binary) parsing systems.*[14]

[14]As will become clear in the sequel, the computations in conditional parsing systems can be seen as a kind of resolution of program clauses [Llo84].

In order to explain such systems, investigate Figure 1.6. The length of the path leading from $(\Box, 0, X, 0)$ to $(\Box, 0, X, 4)$ corresponds with the length of the input, and this causes the linear behaviour of Algorithm 10. We would achieve a better time complexity if the path from $(\Box, 0, X, 2)$ to $(\Box, 0, X, 4)$ could be computed before the path from $(\Box, 0, X, 0)$ to $(\Box, 0, X, 2)$ is completely computed. We do this by introducing items of the form $(I_1 \rightarrow I_2)$, where $I_1, I_2 \in \mathcal{I}$. Informally, the meaning of such an item is "if ever we succeed in computing $I_1 \in U$, then we may add I_2 to U". In our running example, by using such items $((\Box, 0, X, 2) \rightarrow (\Box, 0, X, 3))$ and $((\Box, 0, X, 2) \rightarrow (\Box, 0, X, 4))$ we may compute the path from $(\Box, 0, X, 2)$ to $(\Box, 0, X, 4)$ in Figure 1.6 before we actually find $(\Box, 0, X, 2) \in U$. When eventually we find $(\Box, 0, X, 2) \in U$ then we combine this with the item $((\Box, 0, X, 2) \rightarrow (\Box, 0, X, 4))$ to conclude that $(\Box, 0, X, 4)$ may be added to U.

This is formalized by the following.

Algorithm 11 (Conditional dynamic programming) Let the binary relation $\gg \subseteq \mathcal{I}^* \times \mathcal{I}$ be as in Algorithm 10. Let $\mathcal{I}'$ be an extended set of items defined by $\mathcal{I}' = \mathcal{I} \cup \{(I_1 \rightarrow I_2) \mid I_1, I_2 \in \mathcal{I}\}$. Define the binary relation $\gg' \subseteq \mathcal{I}'^* \times \mathcal{I}'$ as the least relation satisfying:[15]

- Let $I_1, I_2, \ldots, I_m \gg' I$ for all $I_1, I_2, \ldots, I_m, I \in \mathcal{I}$ such that $I_1, I_2, \ldots, I_m \gg I$
- Let $I_1 \gg' (I_2 \rightarrow I_3)$ for all $I_1, I_2, I_3 \in \mathcal{I}$ such that $I_1, I_2 \gg I_3$
- Let $I_1, (I_1 \rightarrow I_2) \gg' I_2$ for all $I_1, I_2 \in \mathcal{I}$
- Let $(I_1 \rightarrow I_2), (I_2 \rightarrow I_3) \gg' (I_1 \rightarrow I_3)$ for all $I_1, I_2, I_3 \in \mathcal{I}$

Compute using least fixed-point iteration the set $U' \subseteq \mathcal{I}'$ defined to be the least set of items satisfying: if there are $I_1, I_2, \ldots, I_m \in U'$ such that $I_1, I_2, \ldots, I_m \gg' I$, then $I \in U'$.

The input is accepted if $(X_{initial}, 0, X_{final}, n) \in U'$.

For the items in U' we can again give composition trees. The minimum depths of all composition trees for each item in U' now is logarithmic in n, which means that parallel computation of U' can be done in logarithmic time.

Example 1.2.9 Consider the grammar and input from Example 1.2.8. We have

- $\gg' (\Box, i, X, i)$ for $0 \leq i \leq n$
- $\gg' (X, i, Y, i+1)$ for $0 \leq i < n$
- $(W, h, X, j), (X, j, Y, i) \gg' (W, h, X, i)$ for $W \in \{\Box, X, Y\}$ and $0 \leq h \leq j \leq i \leq n$

but also

- $(W, h, X, j) \gg' ((X, j, Y, i) \rightarrow (W, h, X, i))$ for $W \in \{\Box, X, Y\}$ and $0 \leq h \leq j \leq i \leq n$
- $(X, j, Y, i) \gg' ((W, h, X, j) \rightarrow (W, h, X, i))$ for $W \in \{\Box, X, Y\}$ and $0 \leq h \leq j \leq i \leq n$
- $I_1, (I_1 \rightarrow I_2) \gg' I_2$ for all $I_1, I_2 \in \mathcal{I}$

[15] The first clause could be simplified to "Let $\gg' I$ for all $I \in \mathcal{I}$ such that $\gg I$", but we abstain from this simplification for presentational reasons.

- $(I_1 \to I_2), (I_2 \to I_3) \gg' (I_1 \to I_3)$ for all $I_1, I_2, I_3 \in \mathcal{I}$

A composition tree for $(\Box, 0, X, 4)$ is given in Figure 1.7. We see that the right subtree of the root represents computation of the path from $(\Box, 0, X, 2)$ to $(\Box, 0, X, 4)$ in Figure 1.6 under the assumption that $(\Box, 0, X, 2)$ may be computed. Note that the new composition tree has depth 4, whereas the one in Figure 1.6 had depth 5. □

For more details, see [CCMR91, BKR91, dV93, Sik93]. The parallel tabular algorithm in [KR88] and [BKR91, Exercise 7.28] seems to be based on the same ideas, although the exact relationship is not clear.

1.2.2.7 Eliminating the input pointer

Provided the stack elements can be extended to carry input positions with them, the notion of input pointer is redundant. For example, instead of transitions of the form $XX_m \ldots X_1 \overset{z}{\mapsto} XY$ we may have transitions of the form $X(i)X_m(i_m) \ldots X_1(i_1) \overset{\gamma}{\mapsto} X(i)Y(j)$, where γ may be a constraint of the form $j = i_1$, where we used to have $z = \epsilon$, or it may be of the form $a_{i_1+1} = a \wedge j = i_1 + 1$, where we used to have $z = a$.

After we have abandoned the explicit input pointer, we may consider more complicated types of transition. Consider for example the transition $X(i, j) \overset{i<m\leq j \wedge a_m=a}{\longmapsto} X(i, j)Y(m)$, whose application leads to a push of the stack symbol $Y(m)$, where input position m, which is somewhere (!) between positions i and j, is such that the m-th input symbol is a.

Each stack symbol may be attached to a different number of positions. In fact, [NS94] shows that many head-driven algorithms can be specified by stack automata whose stack symbols are attached to 2, 3 or 4 input positions.

The idea of attaching input positions to stack symbols has also been described before in [Bar93, Chapter 10], although that text does not go beyond the realm of traditional parsing from left to right.

How the suggested pushdown automata can be realized using dynamic programming is investigated in the next section.

1.2.2.8 Beyond context-free parsing

Not only input positions may be attached to stack elements; we may allow arguments in any domain. An example is the class of Datalog automata [Lan88b]: each element on the stack consists of a stack symbol associated with a fixed number of argument values; this number is the *arity* of the stack symbol. The argument values range over a finite domain. The transitions may contain variables in place of argument values; the well-known principle of *consistent substitution* is used to such variables to relate argument values in newly created stack elements to those of old stack elements. Instead of an input string we have a *set* of terminal symbols[16] associated with a number of argument values (just as the stack symbols, the terminals each have a certain arity); let us call this the *input set*.

For example, in our notation[17] we may have a transition $X\ Y(i) \overset{T(a,i,j)}{\longmapsto} X\ Z(j, h)$ (we assume i, j and h are variables, and a is an argument value), which states that if the

[16] These are actually called *extensional predicates* in [Lan88b]. We will however keep to the terminology we have used before, as much as possible.

[17] Which was inspired by [VdlC93, page 181] for a similar idea applied to logical pushdown automata.

element $Y(i)$ is on top of the stack, for some argument value i, and if it is on top of X, and if for some argument value j the input set contains $T(a, i, j)$, then $Y(i)$ may be replaced by $Z(j, h)$, for an arbitrary argument value h.

A tabular realization of Datalog automata can be found easily by conceptually considering each transition to be the set of transitions resulting from (consistently) substituting each variable by each possible argument value. The appropriate algorithm can now be defined analogously to, for example, Algorithm 5. Note that we need only 2-tuple items instead of 4-tuple items since there are no input positions. On the other hand, we have to do extra checks on the membership of terminals with argument values in the input set.

For Datalog automata, the assumption that the domains are finite allows tabular realizations which always terminate. This is because the number of stack elements is finite. However, if we consider infinite domains such as term domains, then termination of straightforward tabular realizations is no longer guaranteed, and in fact, no tabular realization is capable of always investigating all search paths in finite time.

However, some work has been done to increase the set of pushdown automata with infinite domains which can be handled in finite time by tabular realizations [Lan88a, BVdlC92, Bar93, VdlC93]. An important notion in this area is *subsumption*, which requires the existence of some partial ordering, usually on the argument domain. Intuitively, this ordering decides which argument value is more "specific" than which other value. A newly derived item is then not merely checked for equality to other items (see Step 3 of Algorithm 5) but it is checked whether it is more or less "specific" than existing items in the table, according to the ordering for the arguments. If it is more specific than some existing item, the new item is not added to the table (*weak subsumption*), and for some tabular algorithms, the existing items which are more specific than the newly derived item are removed from the table (*strong subsumption*).

1.2.3 Memo functions

The previous two methods of constructing tabular parsing algorithms were both based on pushdown automata. In this section, our starting-point will be a set of functions, and a tabular method is then obtained by evaluating these functions in a special way: if a function has been evaluated for certain arguments then the result is stored in a table, and if the function is called a second time with the same arguments, the result is retrieved from the table instead of evaluated a second time. This is called *memoization* or *exact tabulation* [Bir80], and functions which are implemented in this way are called *memo functions*. The realization of memo functions in Lisp is discussed in [Nor91].

There are many ways in which the recognition (or parsing) problem can be expressed using a set of functions. In all cases there is a distinguished function that is called with special arguments (such as the first or last input position). As an example, we reproduce a simple top-down recognizer from [Lee92b] below. There is one function for each position in a rhs of a rule. Such a function is denoted as $[A \rightarrow \alpha \bullet \beta]$, where "$\bullet$" indicates a position in the rule $A \rightarrow \alpha\beta$.

Construction 2 (Functional top-down)

$$[A \rightarrow \alpha \bullet](i) \quad = \quad \{i\}$$

$$\begin{aligned}[A \to \alpha \bullet a\beta](i) &= \{j \mid i < n \wedge a = a_{i+1} \wedge j \in [A \to \alpha a \bullet \beta](i+1)\} \\ [A \to \alpha \bullet B\beta](i) &= \{j \mid \exists h \exists B \to \gamma [h \in [B \to \bullet\, \gamma](i) \wedge j \in [A \to \alpha B \bullet \beta](h)]\}\end{aligned}$$

The recognition process is initiated by the call $[S' \to \bullet\, S](0)$, where we use the augmented grammar (see Example 1.2.5). The input is recognized if $n \in [S' \to \bullet\, S](0)$.

Constructions of functional LR parsers can be found in [Lee92b] and are based on the *recursive-ascent* formulation of LR parsing [KA88, Rob88, Rob89, BC88]. Constructions of Marcus' algorithm and left-corner parsers in a functional framework can be found in [Lee92b] and [Lee92a], respectively. Some of these algorithms are reformulated in the *bunch notation* in [Lee93]. Some constructions of generic functional algorithms of which top-down, left-corner and LR parsers are special cases are discussed in [Aug93, Chapter 3].

A very important aspect of functions are that they lend themselves better to formal derivations of algorithms than, say, pushdown automata. For example, in [Lee92b] the top-down recognizer from Construction 2 is derived by means of transformational development from a specification of the recognition problem stated on a high level: from

$$n \in [S' \to \bullet\, S](0) \quad \text{where} \quad [A \to \alpha \bullet \beta](i) = \{j \mid \beta \to^* a_{i+1} \ldots a_j\},$$

the problem is changed by small steps, until an immediately executable description of the problem is obtained, viz. Construction 2.

Regrettably, functional descriptions of recognizers may be *cyclic*, i.e. some function application may be described in terms of itself. For example, the top-down recognizer from Construction 2 for a grammar containing a (left-recursive) rule $A \to A\alpha$ would contain a function application which is defined in terms of itself: $[A \to \bullet\, A\alpha](i) \supseteq \{j \mid \exists h [h \in [A \to \bullet\, A\alpha](i) \wedge j \in [A \to A \bullet \alpha](h)]\}$, and this obviously leads to non-termination of the parsing process. The top-down algorithm above has this problem for any left-recursive grammar. Other algorithms, such as LR parsing and left-corner parsing, have a similar problem with *hidden* left-recursion [LAKA92, Lee92a] (see also Chapters 2, 4 and 5).

For most kinds of functional recognizers, the tabular realization using memoization is correct if and only if the functional description is not cyclic. One exception seems to be functions which do not yield a set of input positions, but only a boolean value. For example, the following is a reformulation of top-down recognition as presented in [She76].[18]

Construction 3 (Boolean functional top-down)

$$\begin{aligned}[A \to \alpha \bullet](i,i) &= \mathit{TRUE} \\ [A \to \alpha \bullet a\beta](i,j) &= i < j \wedge a = a_{i+1} \wedge [A \to \alpha a \bullet \beta](i+1,j) \\ [A \to \alpha \bullet B\beta](i,j) &= \bigvee_{i \le h \le j, B \to \gamma} [B \to \bullet\, \gamma](i,h) \wedge [A \to \alpha B \bullet \beta](h,j)\end{aligned}$$

The input is recognized if $[S' \to \bullet\, S](0,n)$ evaluates to TRUE.

We may now still have cycles in the functional definition; for example, for $A \to A\alpha$ we have, amongst other things, $[A \to \bullet\, A\alpha](i,j) \leftarrow [A \to \bullet\, A\alpha](i,j) \wedge [A \to A \bullet \alpha](j,j)$. However, we may omit this implication, since nothing useful can be derived from it: it is a tautology.

[18] Similar constructions of left-corner and LR recognizers are presented in [Aug93, Section 3.7].

On this idea, the memo functions from [She76] for top-down parsing are based: if a function $[A \to \alpha \bullet \beta]$ is called with certain arguments (i, j), while this function is already in the process of being evaluated for the same arguments (i.e. there is a cycle) then the cyclic call is immediately terminated giving the result *FALSE*. It is important to note that this approach does not necessarily give correct tabular recognizers if the functions are not boolean-valued.[19]

Formally, if there are cycles in functional definitions, then the functions are not uniquely determined. We can however explicitly state that we want the *least fixed-point* of a set of function definitions, that is, functions which satisfy the equations, but such that they evaluate to the smallest values possible. For Construction 2, this means that the sets $[A \to \alpha \bullet \beta](i)$ should contain as few elements as possible. For Construction 3, the solution we described to enforce termination actually comes down to letting the values $[A \to \alpha \bullet \beta](i, j)$ be *FALSE* unless the definitions require otherwise.

Until now we have assumed that the evaluation of some set of function definitions is initiated by some initial call such as $[S' \to \bullet\ S](0)$. However, we may also start with other calls, with other functions or other arguments, some of which may eventually contribute to computation of $[S' \to \bullet\ S](0)$ whereas others may not be helpful. This is called *overtabulation* [Bir80]. Applied to functional recognizers, this may result in bidirectional algorithms (see Section 1.2.2.5). For example, [Par90, Section 6.6.2] demonstrates how an algorithm related to Construction 3 can be transformed into the CKY algorithm (Section 1.2.6) by applying overtabulation.

1.2.4 Query processing

Finding the smallest solution to a set of logical implications has been studied extensively in the area of query processing. Instead of a set of function definitions, we then have a set of relations defined in the form of Horn clauses. For example, we may specify a top-down recognizer by a set of clauses of the following three forms.

Construction 4 (Horn clause top-down)

$$\begin{array}{rcll} [A \to \alpha \bullet](i,i) & \leftarrow & \mathit{TRUE} & \text{for } A \to \alpha \\ [A \to \alpha \bullet a\beta](i,j) & \leftarrow & i < n \wedge a = a_{i+1} \wedge [A \to \alpha a \bullet \beta](i+1,j) & \text{for } A \to \alpha a \beta \\ [A \to \alpha \bullet B\beta](i,j) & \leftarrow & [B \to \bullet\ \gamma](i,h) \wedge [A \to \alpha B \bullet \beta](h,j) & \text{for } A \to \alpha B \beta, B \to \gamma \end{array}$$

The query $[S' \to \bullet\ S](0, n)$ then expresses the recognition problem.

In the area of deductive databases, queries are traditionally evaluated by an iterative least fixed-point algorithm which is performed bottom-up.[20] More precisely, we first assume that each relation holds for no tuple of arguments. Then by applying the clauses which have no relations in their right-hand sides, we find that the relations hold on some tuples of arguments. For example, for Construction 4 we immediately see that $[A \to \alpha \bullet]$ holds on all pairs of input positions (i, i). Some other clauses may now be used to derive tuples for the

[19] In [Lee91] it was proposed that a call $[A \to \alpha \bullet \beta](i)$ from Construction 2 should yield $\emptyset$ if it is a cyclic call. Other than suggested in that paper, this definitely does not give *complete recognizers*, i.e. not for all correct input will the recognizer give an affirmative answer. A correct solution to top-down parsing for left-recursive grammars different from the solutions presented here is given in Chapter 5.

[20] This is similar to overtabulation, mentioned in Section 1.2.3.

relation in a left-hand side from tuples we already had for the relations in the right-hand side. If we can derive no more new tuples, the parsing process has finished and the query $[S' \rightarrow \bullet\ S](0, n)$ can be answered.[21]

This simple bottom-up algorithm is however very inefficient since it does not take into account the top-down information that we are interested in answering $[S' \rightarrow \bullet\ S](0, n)$ but not, say, $[S' \rightarrow \bullet\ S](i, j)$ for $i \neq 0 \vee j \neq n$. It may therefore be more efficient to first compute (top-down) the argument values which are actually needed to answer the query, so that subsequent bottom-up evaluation can be restricted to finding only the *required* results.

The literature contains a number of realizations of this idea, which differ in a few details. We have for example the Alexander Method [RLK86], generalized to the Alexander Templates [Sek89], OLD resolution with tabulation [TS86] and SLD-AL resolution [Vie89], (Generalized Supplementary) Magic Sets [BR87], generalized to Magic Templates [Ram88] and Magic Filters [LY91], and extension tables [Die87].[22]

For some sets of Horn clauses, query processing with these methods is very similar to what happens when a set of memo functions is evaluated. (Compare e.g. Constructions 2 and 4. See [Gar87] for an example of a translation from sets of Horn clauses to sets of functions.) However, for query processing, termination is guaranteed, at least for finite argument domains. (This difference with memo functions is discussed clearly in [Die87].) It is important to note that some of the above methods can also handle some sets of Horn clauses in which the parameters range over infinite domains, e.g. term domains.

As [PW83] points out, query processing methods may lead to bidirectional algorithms (that paper mentions *head word parsing*) even if a set of Horn clauses seems to suggest that the input is to be processed from left to right (in particular, if we assume that the order of the members in the right-hand sides of the clauses determines the order of processing).

Note that the most commonly used translation of context-free grammars to sets of Horn clauses is the following.

Construction 5 (Simple Horn clause top-down)

$$\begin{array}{rcll} A(i_0, i_m) & \leftarrow & X_1(i_0, i_1) \wedge \ldots \wedge X_m(i_{m-1}, i_m) & \text{for } A \rightarrow X_1 \ldots X_m \\ a(i, i+1) & \leftarrow & i < n \wedge a = a_{i+1} & \text{for } a \in T \end{array}$$

The query $S(0, n)$ then expresses the recognition problem.

This construction can be generalized to translate definite clause grammars to sets of Horn clauses [PW80]. Some variants of this construction are mentioned in Chapter 5.

A construction of sets of Horn clauses from tree adjoining grammars is given in [Lan88d].

1.2.5 Reduction of grammars

We say a grammar is *reduced* if for all A we have $S \rightarrow^* \alpha A \beta \rightarrow^* w$ for some α, β, and w; in other words, each nonterminal can be used in some derivation of some terminal string.

[21] A generalization of this algorithm for a more general class of sets of clauses, with applications in natural language processing, is described in [Joh94]. See [Sei94] for more sophisticated algorithms.

[22] In the context of transformational programming, it is interesting to note that some of these methods are defined by a transformation of a set of Horn clauses. Such a transformation accomplishes top-down evaluation with regard to the original set of clauses although the transformed set of clauses is evaluated bottom-up [War92].

Transforming a grammar such that it becomes reduced is called *reduction*. Reduction can be performed by determining the set of all nonterminals A for which the property $S \rightarrow^* \alpha A \beta \rightarrow^* w$ holds, and then removing all rules containing occurrences of the nonterminals not in that set [AU72].

Tabular parsing can be performed in the following way: first we construct a grammar G_w from a context-free grammar G and input w. Then we reduce the grammar. The input is recognized provided we obtain a non-empty grammar.

For example, if we have some fixed grammar G and a string $w = a_1 \ldots a_n$, then we may construct a context-free grammar G_w by the following, where i, j and h are numbers between 0 and n (and not variables as in Construction 4).

Construction 6 (Context-free top-down)

$$
\begin{array}{rll}
[A \rightarrow \alpha \bullet](i,i) \rightarrow \epsilon & & \text{for } i \leq n, A \rightarrow \alpha \\
[A \rightarrow \alpha \bullet a\beta](i,j) \rightarrow [A \rightarrow \alpha a \bullet \beta](i+1,j) & & \text{for } i < n, j \leq n, \text{ and } A \rightarrow \alpha a \beta \\
& & \quad \text{such that } a = a_{i+1} \\
[A \rightarrow \alpha \bullet B\beta](i,j) \rightarrow [B \rightarrow \bullet \gamma](i,h)\ [A \rightarrow \alpha B \bullet \beta](h,j) & & \text{for } i,j,h \leq n, \\
& & \quad \text{and } A \rightarrow \alpha B \beta, B \rightarrow \gamma
\end{array}
$$

Another way to look at this construction is as the composition of Construction 4 and the transformation which, given an input w, replaces the variables i, j and h by actual input positions in all possible ways, and checks the condition $a = a_{i+1}$.

Another example of a construction of grammars G_w can be derived from Construction 5, in the same way as Construction 6 is derived from Construction 4:

Construction 7 (Simple context-free top-down)

$$
\begin{array}{rcll}
A(i_0,i_m) & \rightarrow & X_1(i_0,i_1)\ \ldots\ X_m(i_{m-1},i_m) & \text{for } i_0,\ldots,i_m \leq n, \text{ and } A \rightarrow X_1 \ldots X_m \\
a(i,i+1) & \rightarrow & \epsilon & \text{for } i < n \text{ such that } a = a_{i+1}
\end{array}
$$

A generalized form of this construction, with states of finite automata in lieu of input positions, has been described in [BHPS64]. One advantage of Construction 7 over Construction 6 is that the rules in the resulting grammar G_w have the same structure as those in the original grammar G.

Subsequent reduction of a grammar G_w may be done in a number of ways. Some may be very similar to how some query processing methods would evaluate a set of clauses such as those produced by Construction 4. But also other reduction algorithms may be used. This adds extra flexibility to this method: not only can we find different constructions of G_w from G and w, we can also find different reduction algorithms for G_w. We may also combine the construction of G_w with its reduction, in order to allow even more flexibility.

A further advantage of this method is that a parse forest is constructed, in the form of the reduced grammar (cf. Section 1.2.2.2).

The method is not restricted to context-free grammars G; for example, [VSW93] describes how tree adjoining grammars G can be transformed into context-free grammars G_w. The method is furthermore not restricted to yield context-free grammars G_w; for example, [VSW93] describes how tree adjoining grammars G can be transformed into linear indexed grammars G_w.[23] In [Lan92] both G and G_w are tree adjoining grammars. Instead of a

[23] Linear indexed grammars require a specialized reduction algorithm [VSW93].

grammar G, we may even take a pushdown automaton M and construct a context-free grammar M_w, as shown in [Lee89].

1.2.6 Covers

A tabular parsing algorithm can be specified as the composition of a grammar transformation and a fixed tabular parsing algorithm for the transformed grammars [Lee89]. An example of such a fixed tabular algorithm is the Cocke-Kasami-Younger (CKY) algorithm [Har78], which requires a grammar to be in Chomsky normal form. For presentational convenience, we will consider an extended Chomsky normal form, which allows rules of the form $A \to a$, of the form $A \to BC$, and of the form $A \to B$.[24] If the CKY algorithm is adapted accordingly, we obtain the following, which is due to [GHR80]:

Algorithm 12 (Cocke-Kasami-Younger) Assume a grammar G in extended Chomsky normal form. For input $a_1 \ldots a_n$, define the sets of nonterminals $U_{i,j}$, with $0 \leq i < j \leq n$, as follows.

- $U_{i-1,i} = \{A \mid A \to^* a_i\}$
- $U_{i,j} = \{D \mid \exists A \to BC \exists h[B \in U_{i,h} \wedge C \in U_{h,j} \wedge D \to^* A]\}$ for $i + 1 < j$.

Note that evaluation of a set $U_{i,j}$ with $i + 1 < j$ requires prior evaluation of all $U_{i,h}$ and $U_{h,j}$ for h with $i < h < j$, but this still leaves much freedom on the order of evaluation.

The input is recognized if $S \in U_{0,n}$.

The following is an example of a construction of a transformed grammar $C(G)$ in extended Chomsky normal form, from an arbitrary grammar G without epsilon rules.

Construction 8 (Extended Chomsky normal form) Assume a grammar $G = (T, N, P, S)$ without epsilon rules. The nonterminals of the new grammar $C(G)$ are of the form $[a]$, for $a \in T$, or of the form $[A \to \alpha \bullet \beta]$, for $A \to \alpha\beta \in P$. The rules of the new grammar $C(G)$ are given by

$$
\begin{array}{rcll}
[a] & \to & a & \text{for } a \in T \\
[A \to a \bullet \alpha] & \to & [a] & \text{for } A \to a\alpha \in P \\
[A \to \alpha a \bullet \beta] & \to & [A \to \alpha \bullet a\beta]\ [a] & \text{for } A \to \alpha a\beta \in P \\
[A \to B \bullet \alpha] & \to & [B \to \beta \bullet] & \text{for } A \to B\alpha, B \to \beta \in P \\
[A \to \alpha B \bullet \beta] & \to & [A \to \alpha \bullet B\beta]\ [B \to \gamma \bullet] & \text{for } A \to \alpha B\beta, B \to \gamma \in P
\end{array}
$$

A transformed grammar $C(G)$, resulting from Construction 8 or a related transformation, is a *cover* for G, which informally means that for each input, the set of parse trees according to G can be effectively recovered from the set of parse trees for the same input according to $C(G)$.

A disadvantage of the original CKY algorithm is that it is completely bottom-up, or in other words, it does not allow filtering of the nonterminals in $U_{i,j}$ based on any of the sets $U_{i',j'}$ with $j' \leq i$ or $j \leq i'$. A way to introduce filtering in the CKY algorithm would be to omit any nonterminal A from $U_{i,j}$ if it cannot be combined directly of indirectly with any

[24] We ignore rules of the form $S \to \epsilon$.

nonterminal B in $U_{i',i}$ for some $i' < i$, or more concretely, if $A \angle^* C$ does not hold for any C such that there is a rule $D \rightarrow BC$, and B is in some set $U_{i',i}$. (This is a weak form of *top-down filtering*, which is further discussed in Chapter 3.)

This leads to the following algorithm.

Algorithm 13 (Cocke-Kasami-Younger with filtering) As Algorithm 12 but here we also compute the sets F_i, with $0 \leq i < n$, with which the sets $U_{i,j}$ are filtered.

- $U_{i-1,i} = \{A \mid A \rightarrow^* a_i\} \cap F_{i-1}$
- $U_{i,j} = \{D \mid \exists A \rightarrow BC \exists h[B \in U_{i,h} \wedge C \in U_{h,j} \wedge D \rightarrow^* A]\} \cap F_i \quad \text{for } i+1 < j.$
- $F_0 = \{A \mid A \angle^* S\}$
- $F_j = \{A \mid \exists D \rightarrow BC \exists i[B \in U_{i,j} \wedge A \angle^* C]\} \quad \text{for } j > 0.$

Note that this set of functions leaves no freedom on the order of evaluation[25].

We may refine this idea even further and specialize each set F_i, with which the sets $U_{i,j}$ are filtered for all j, to different sets $F_{i,j}$ with which the sets $U_{i,j}$ are filtered. In [Lee89], the sets $F_{i,j}$ are computed by evaluating $G_{j-i}(\{A \mid \exists h[A \in U_{h,i}]\})$, where G_1, G_2, ... are fixed functions independent from the input.

A tabular parsing algorithm can now be described by giving a transformation C from arbitrary grammars to grammars in extended Chomsky normal form, and by specifying the family of functions G_1, G_2, ... If however the canonical definition of the F_i is kept as in Algorithm 13, a tabular algorithm is determined only by some transformation C.

The above ideas have been introduced in [Lee89], which generalizes (extended) Chomsky normal form to the *bilinear* form: right-hand sides contain at most two nonterminals, and an arbitrary number of terminals. This requires the CKY algorithm with filtering to be generalized to a kind of parsing which is called *C-parsing*.

That Earley's algorithm minus some filtering can be seen as the composition of a grammar transformation and the CKY algorithm is explained in [GHR80].

In [SS91, SS94] some forms of filtering are discussed which allow *bidirectional* parsing.

Instead of choosing the CKY algorithm as the fixed algorithm, one may also choose Rytter's algorithm ([dV93, Section 3.2]; see also Section 1.2.2.6), possibly adapted to some extended Chomsky normal form. Had this been done in [dV93], then the two new variants of Rytter's algorithm in [dV93, Section 3.3] and [dV93, Section 3.4] *could* have been described in terms of two simple grammar transformations.

1.2.7 Parsing schemata

A very abstract view on parsing is provided by the *parsing schemata* of [Sik93]. Parsing schemata are related to the "primordial soup" algorithm (see also [JPSZ92]), which allows abstract specification of parsing algorithms: starting from the rules and input symbols seen as fragments of parse tree, composition of fragments of parse tree is done until a complete parse tree is obtained. Irrelevant procedural aspects such as the order in which the fragments are composed are avoided.

[25] unless the sets are computed incrementally, using least fixed-point iteration

Parsing schemata allow formal derivation of tabular parsing algorithms. This is roughly explained by means of the following example. Consider the class of context-free grammars in Chomsky normal form (Section 1.2.6). A *tree-based parsing schema* may be specified which allows composition of a fragment of parse tree with root A and yield $a_{i+1} \ldots a_j$ from two fragments with roots B and C, and yields $a_{i+1} \ldots a_h$ and $a_{h+1} \ldots a_j$, respectively, and a rule $A \rightarrow BC$ (we assume, as usual, that the input is $a_1 \ldots a_n$). Such a tree-based parsing schema can be seen as an abstract description of a parsing algorithm. In the running example, the parsing schema formalizes nondeterministic shift-reduce parsing [ASU86]. Note that the objects manipulated are individual fragments of parse tree, both in the case of the tree-based parsing schema, and in the case of a nondeterministic shift-reduce parser.

A tabular algorithm is obtained by considering an equivalence relation on fragments [Sik93, Chapter 4]. This equivalence relation should be such that two fragments can only be in the same equivalence class provided they are indistinguishable with regard to the operation of composing fragments into larger fragments (such an equivalence relation is a *congruence relation*; we abstain from details here). An example is the equivalence relation which puts all fragments with the same root and the same yield in the same class. It is obvious that two fragments in the same class have the same role with regard to composition into larger fragments. If the tree-based parsing schema is transformed in such a way that it manipulates equivalence classes of fragments in lieu of individual fragments of parse tree, then a *quotient parsing schema* results. A quotient parsing schema can be seen as a formal description of a tabular algorithm. In the running example, a formal description of the CKY parsing algorithm [CS70] is obtained.

Note that taking an equivalence class of fragments of parse tree can be seen as a generalization of the notion of "packing" (Section 1.2.1.5). An advantage of parsing schemata with respect to other approaches to tabular parsing is that they allow description of the construction of parse forests without mention of the mechanisms that do the construction, such as graph-structured stacks or memo functions. Parsing schemata are furthermore particularly appropriate for relating different parsing algorithms [Sik93, Chapters 5 and 6].

1.2.8 Chart parsing

As opposed to the methods from the previous sections, *chart parsing* is not a systematic way of deriving tabular algorithms from non-tabular ones; rather, chart parsing consists of notions and terminology which are traditionally used to describe tabular parsing algorithms in the disciplines of artificial intelligence and computational linguistics.

The objective of chart parsing was to find a useful and comprehensive theoretical framework for the description of tabular parsing algorithms. I feel that this attempt has utterly failed. In particular, the parsing schemata from [Sik93] show that there is an alternative framework for the description of tabular parsing algorithms that possesses much more descriptional elegance and mathematical justification. I therefore confine myself to giving the most important references to the chart parsing literature [Kay73, Kap73, Kay86, Var83, Win83, TR84, SDR87].

1.2.9 General issues

1.2.9.1 Complexity issues

A surprising fact about the relationship between pushdown automata and their tabular realizations is that small changes in the pushdown automata which only marginally affect the lengths of the search paths may cause more substantial changes in the complexities of the tabular realizations.

For example, consider Construction 1 from Example 1.2.5, which translates context-free grammars into top-down recognizers. The transitions of the form

$$[A \rightarrow \alpha \bullet B\beta] \quad \stackrel{\epsilon}{\mapsto} \quad [A \rightarrow \alpha \bullet B\beta]\ [B \rightarrow \bullet\, \gamma]$$

give rise in a tabular realization (e.g. Algorithm 7 [26]) to a step such as:

predictor For $(j, [A \rightarrow \alpha \bullet B\beta], i) \in U$ add $(i, [B \rightarrow \bullet\, \gamma], i)$ to U.

If the length of the input is considered bounded, then this step is performed $\mathcal{O}(|P| \times |G|)$ times, where $|P|$ denotes the number of rules, and $|G|$ denotes the sum of the lengths of the right-hand sides of all rules. If a transition of the above kind is split up into two transitions

$$\begin{aligned} [A \rightarrow \alpha \bullet B\beta] \quad &\stackrel{\epsilon}{\mapsto} \quad [A \rightarrow \alpha \bullet B\beta]\ B \\ B \quad &\stackrel{\epsilon}{\mapsto} \quad [B \rightarrow \bullet\, \gamma] \end{aligned}$$

then more steps have to be performed to find a single parse of some input. These two kinds of transitions however also give rise to two different predictor steps in the tabular realization:

predictor 1 For $(j, [A \rightarrow \alpha \bullet B\beta], i) \in U$ add (i, B, i) to U.

predictor 2 For $(i, B, i) \in U$ add $(i, [B \rightarrow \bullet\, \gamma], i)$ to U.

These steps will be applied $\mathcal{O}(|G|)$ and $\mathcal{O}(|P|)$ times, respectively, which together is $\mathcal{O}(|G| + |P|) = \mathcal{O}(|G|)$. The tabular realization therefore now has a lower theoretical time complexity, at least for the predictor step(s).

A similar thing holds for a transition of the form

$$[A \rightarrow \alpha \bullet B\beta]\ [B \rightarrow \gamma \bullet] \quad \stackrel{\epsilon}{\mapsto} \quad [A \rightarrow \alpha B \bullet \beta]$$

which should be turned into two transitions

$$\begin{aligned} [B \rightarrow \gamma \bullet] \quad &\stackrel{\epsilon}{\mapsto} \quad \overline{B} \\ [A \rightarrow \alpha \bullet B\beta]\ \overline{B} \quad &\stackrel{\epsilon}{\mapsto} \quad [A \rightarrow \alpha B \bullet \beta] \end{aligned}$$

to ensure that the completer step(s) of the tabular realization (Algorithm 7) can be performed in linear time, expressed in the size of the grammar.

This phenomenon has also been observed e.g. in [Bar93, page 59] and [VdlC93, page 158]. The ideas leading to a linear time complexity for Earley's algorithm have already been

[26] If 4-tuples items are used instead of 3-tuple items the following discussion is equally relevant.

mentioned in [GHR80]. The version of Earley's algorithm in [KA89] uses two predictor steps as discussed above, but only one completer step.

A related phenomenon is described in [NB94]: if we take a deterministic PDA and use it to parse incomplete input of the form "$* \; w$" by means of Algorithm 9, then the time complexity may be quadratic in the length of the input. By applying a certain transformation to the PDA, which increases the lengths of the search paths, we can achieve that the time complexity of Algorithm 9 becomes linear.

A third example of how a small change in a PDA may have important consequences for its tabular realization again deals with the time complexity of Earley's algorithm. As explained in [Leo91], if Earley's algorithm is applied to LR(k) grammars without using lookahead (i.e. in the form of Algorithm 7) then the time complexity is in the general case quadratic in the length of the input. The quadratic behaviour is caused by some "cascades" of applications of the completer step ($\mathcal{O}(n)$ applications) which may occur a number of times ($\mathcal{O}(n)$ times).

The solution in [Leo91] to avoid this behaviour is to redirect some input pointers in the items when a cascade occurs, so that the next time the algorithm "leaps over" the cascade.[27] The full details as presented in [Leo91] are quite complicated and, I feel, not very intuitive. However, it seems that a linear complexity can also be obtained by the more simple approach of changing the construction of PDAs from which Earley's algorithm is derived, i.e. Construction 1.

In order to explain this approach, we consider a PDA yielded by Construction 1. If this PDA has a stack element $[B \to \gamma \bullet]$ on top of the stack, then a transition of the form

$$[A \to \alpha \bullet B\beta] \; [B \to \gamma \bullet] \quad \stackrel{\epsilon}{\mapsto} \quad [A \to \alpha B \bullet \beta]$$

is applied. If $\beta = \epsilon$ then this results in a stack element $[B' \to \gamma' \bullet] = [A \to \alpha B \bullet]$ of the same form on top of the stack. As next step, therefore, a transition of the form

$$[A' \to \alpha' \bullet B'\beta'] \; [B' \to \gamma' \bullet] \quad \stackrel{\epsilon}{\mapsto} \quad [A' \to \alpha' B' \bullet \beta']$$

is applied, where again β' may be ϵ. This may be repeated a number of times. For the tabular realization (Earley's algorithm) such a sequence occurs as a cascade of applications of the completer step.

We can avoid such sequences by making sure that stack elements cannot occur on top of stack elements of the form $[A \to \alpha \bullet B]$. This is accomplished by changing the second kind of transition of Construction 1 (which corresponds to the predictor step of Earley's algorithm) in such a way that an item $[A \to \alpha \bullet B]$ is removed from the stack when it causes an item $[B \to \bullet \; \gamma]$ to be pushed.[28] The fourth kind of transition of Construction 1 then needs to be refined accordingly. For reasons which have to do with context-independence (Section 1.2.2.1) we incorporate an extra nonterminal in the stack symbols, although this extra nonterminal is redundant for the PDAs themselves.

The result is the construction below. Let a context-free grammar be given. We again augment the grammar, this time by adding the rule $S' \to S\#$, where $\#$ is a fresh symbol. For presentational reasons we assume that the grammar does not contain any epsilon rules. (How epsilon rules may be handled is suggested in [Leo91].)

[27] Cf. *path compression* in the UNION-FIND algorithm [AHU74, Section 4.7].

[28] It is obvious that this idea is related to the *last call optimization* described in [MW88, Section 11.7].

Construction 9 (Top-down without cascades) Construct the PDA with the transitions below. The stack symbols are $X_{initial} = \square$ and symbols of the form $[C, A \rightarrow \alpha \bullet \beta]$, where $A \rightarrow \alpha\beta$ and $C \in N$; as X_{final} we take $[S', S' \rightarrow S \bullet \#]$.

$\square$	$\overset{\epsilon}{\mapsto} \square\ [S', S' \rightarrow \bullet\ S\#]$	
$[C, A \rightarrow \alpha \bullet B\beta]$	$\overset{\epsilon}{\mapsto} [C, A \rightarrow \alpha \bullet B\beta]\ [B, B \rightarrow \bullet\ \gamma]$	for $A \rightarrow \alpha B\beta, B \rightarrow \gamma$ with $\beta \neq \epsilon$
$[C, A \rightarrow \alpha \bullet B]$	$\overset{\epsilon}{\mapsto} [C, B \rightarrow \bullet\ \gamma]$	for $A \rightarrow \alpha B, B \rightarrow \gamma$
$[C, A \rightarrow \alpha \bullet a\beta]$	$\overset{a}{\mapsto} [C, A \rightarrow \alpha a \bullet \beta]$	for $A \rightarrow \alpha a\beta$
$[C, A \rightarrow \alpha \bullet B\beta]\ [B, D \rightarrow \gamma \bullet]$	$\overset{\epsilon}{\mapsto} [C, A \rightarrow \alpha B \bullet \beta]$	for $A \rightarrow \alpha B\beta, D \rightarrow \gamma$ with $\beta \neq \epsilon$

Note that the transitions of the second and third kind ensure that no element is pushed on top of an element of the form $[C, A \rightarrow \alpha \bullet B]$. The result is that if $[C, A \rightarrow \alpha \bullet B\beta]$ and $[B, D \rightarrow \gamma \bullet]$ are at some point the top-most two stack elements, then $\beta \neq \epsilon$ is guaranteed. (The condition $\beta \neq \epsilon$ in the fifth kind of transition in Construction 9 is therefore redundant for the PDAs themselves but nevertheless essential for context-independence.)

Verification of the fact that the above construction yields only context-independent PDAs is straightforward. This allows us to derive a variant of Earley's algorithm as a tabular realization of the PDAs using 3-tuple items (Section 1.2.2.1). The proof that the resulting algorithm has a linear time complexity for LR(k) grammars is beyond the scope of this thesis, but can be easily constructed relying on some results in [Leo91]. We want to stress however that the algorithm we propose has a run-time behaviour which is fundamentally different from the one in [Leo91].

(In [TDL91] an idea similar to the one we described above is applied to bottom-up parsing. This leads to an algorithm such that the time spent at each input position is bounded by a constant, provided parallelism is used to deal with nondeterminism.)

In Chapter 3 we see another remarkable property of tabular realization: if the search trees of a PDA are made smaller by sophisticated techniques multiplying the number of stack symbols, then tabular realization may be less efficient. Some changes to a PDA have no effect at all on the structure of the search trees, but still have consequences for the size of the parser and the efficiency of tabular realizations (see footnote 4 on page 75).

The observations in this section can also be reformulated to apply to e.g. memo functions and query processing.

1.2.9.2 Hybrid algorithms

It is possible to combine tabular parsing algorithms with non-tabular ones. For example, Algorithm 2 ensures that to each set U_i at most one node with the same label is added, which gives the graph-structured stacks properties characteristic to tabular parsing. We could however relax this constraint for some types of stack symbols, such that sometimes more than one node with the same label may be added to some set U_i. The effect is that one part of a PDA (i.e. one subset of the stack symbols) is handled using tabular parsing whereas another part (the other stack symbols) using ordinary breadth-first computation.

For the dynamic programming approach, [Lan74] proposes that a tabular algorithm be used for the parts of the automaton where nondeterminism occurs, and a non-tabular algorithm for the deterministic parts, and [Lan88a] proposes a combination of tabulation with backtracking. For non-context-free parsing, one may further decide not to apply subsumption (Section 1.2.2.8) for one particular subset of the stack symbols [VdlC93, pages 192–193].

There is a similar hybrid approach to functional parsing algorithms: memoization may be restricted to a subset of the functions. In the same way, hybrid approaches are possible for query processing [TS86, Vie89].

The main advantage of using hybrid algorithms instead of full tabular algorithms is the reduction of memory requirements: if parsing tables are implemented naively, then the size of the tables needs to be at least linear in the number of stack symbols. For example, in Section 1.2.9.1 we described a construction of automata whose stack symbols are of the form $[A \rightarrow \alpha \bullet \beta]$, for $A \rightarrow \alpha\beta \in P$, or of the forms A or $\overline{A}$, for $A \in N$. For the first kind of stack symbol the size of the table needs to be linear in the size of the grammar (i.e. $\mathcal{O}(|G|)$), although for the other kinds of stack symbol not more than $\mathcal{O}(|N|)$ space is needed. It is therefore reasonable to apply tabulation only to the stack symbols of the second and third kind. The drawback is a deterioration of the time complexity from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^{p+1})$, where p is the length of the longest rhs. A similar hybrid algorithm is described in Chapter 2.

1.2.9.3 Complementation

Tabular algorithms usually operate by repeatedly adding to a table some element computed from a collection of existing elements in the table. In some cases, however, tabular algorithms are to add an element to the table when it has been determined that some collection of elements can *not* be added to the table.

An application of this idea is described in [HS91]. First, a grammatical formalism is defined called "hierarchical complement intersection grammars". The most interesting extension of this formalism with regard to traditional context-free grammars is the use of complementation in right-hand sides of rules, which allows one to express that some string is *not* generated by some nonterminal. In order to avoid paradoxes[29] this formalism requires that nonterminals are arranged in a hierarchy so that if nonterminal A is defined in terms of the complement of nonterminal B, then A is strictly higher in this hierarchy than B.

Next, the paper gives an adaptation of Earley's algorithm to this new formalism. The most obvious novelty of this algorithm is that it computes elements in the table according to the hierarchy of the nonterminals: for fixed input positions, the elements corresponding with nonterminals low in the hierarchy are computed before those corresponding with nonterminals higher up the hierarchy.

Similar ideas for query processing with negation can be found in [SI88]. Again a hierarchy, this time of the predicates, is required to avoid paradoxes and to determine the order of processing.

[29] Consider e.g. a rule stating that "nonterminal A generates a string v if nonterminal A does not generate v".

1.2.9.4 Full incrementality

Many of the tabular parsing algorithms discussed until now are *left-to-right incremental*, which means that the part of a table pertaining to some prefix of an input string can be computed before any aspect of the remaining part of the input is investigated (this holds in particular for synchronous realizations of the tabular algorithms). Another way of looking at left-to-right incrementality is to regard the part of the table pertaining to some prefix w of some input wv as a complete table U for the (complete) input w; proceeding the parsing process with the next terminal a in the input (say $v = av'$, some v') then corresponds to updating the table U for the modified input wa.

This idea can be generalized to other modifications of the input: suppose we have a table U for input w, now compute the modified table U' for the modified input w_1aw_2, where $w_1w_2 = w$ and a is some terminal, or for the modified input w_1w_2, where $w_1aw_2 = w$ for some a. The modified table U' should, as much as possible, be computed based on the original table U, or in other words, it should not be constructed from scratch. This is called *full incremental parsing* and differs from left-to-right incremental parsing in that some elements from the table may have to be *removed* when the input is modified. Computation of those elements of the table that have to be removed hinges on an additional data structure in the table which records the dependencies between table elements (i.e. information recording which elements in the table justify the presence of which other elements in the table).[30] For more details, see [WR93, Wir93].

1.2.9.5 Parallelism

Tabular algorithms usually allow some parallel execution. For example, the dynamic programming algorithm (Algorithm 5) computes a set of items U in such a way that at each point in time, a number of items may be computed simultaneously. By spreading the computation of such a collection of items over different processors, the computations for different subsets of U may be done in parallel.

In Section 1.2.2.6 we discussed parallelism for the dynamic programming approach to tabular parsing. The combination of parallelism and graph-structured stacks is treated in [TN91a]. Parallelism has been studied in the context of functional and logical programming languages in [PvE93] and [Sha87], respectively. Parallelism for parsing in particular is discussed in [dV93, Sik93, OL93].

1.3 Overview of this thesis

In Chapter 2 we show how graph-structured stacks can be used to define a tabular algorithm based on left-corner parsing. We stress the advantages of such an algorithm over tabular LR parsing, for which graph-structured stacks were originally devised. A more general view of tabular parsing based on a whole family of stack algorithms is given in Chapter 3. We will argue that there is a tabular algorithm which is optimal according to certain requirements.

[30] I conjecture however that the more simple mechanism of *reference counts* [ASU86] is also adequate in many cases.

Chapter 4 investigates variants of LR parsing which possess better properties with regard to termination than traditional LR parsing. These variants are inspired by grammatical transformations.

In Chapter 5 a new parsing algorithm is presented which is top-down in nature but terminates for left-recursive grammars. The parsing algorithms in Chapter 5 are presented as transformations from context-free grammars to definite clause grammars, which is shown to have certain descriptional advantages.

Tabular algorithms were originally devised for context-free grammars. If they are applied to context-free grammars extended with arguments, then some way has to be found to evaluate the arguments. Such a method is presented in Chapter 6. It consists of ordinary construction of a context-free parse forest, followed by an algorithm which decorates the forest with argument values and additionally reduces the forest.

Development of programs is usually supported by automated devices. In Chapter 7 we argue that development of grammars should also be supported by tools. We describe such a tool, and compare it with similar tools described in the existing literature.

Chapter 2

Generalized Left-Corner Parsing

We show how techniques known from generalized LR parsing can be applied to left-corner parsing. The resulting parsing algorithm for context-free grammars has some advantages over generalized LR parsing: the sizes and generation times of the parsers are smaller, the produced output is more compact, and the basic parsing technique can more easily be adapted to arbitrary context-free grammars.

The algorithm can be seen as an optimization of algorithms known from existing literature. A strong advantage of our presentation is that it makes explicit the role of left-corner parsing in these algorithms.

2.1 Introduction

Generalized LR parsing was first described by Tomita [Tom86, Tom87]. It has been regarded as the most efficient parsing technique for context-free grammars for natural languages. The technique has been adapted to other formalisms than context-free grammars in [Tom88].

A favourable property of generalized LR parsing (henceforth abbreviated to *GLR parsing*) is that input is parsed in polynomial time. To be exact, if the length of the rhs of the longest rule is p, and if the length of the input is n, then the time complexity is $\mathcal{O}(n^{p+1})$. Theoretically, this may be worse than the time complexity of Earley's algorithm [Ear70], which is $\mathcal{O}(n^3)$. For practical cases in natural language processing however, GLR parsing seems to give the best results.

The polynomial time complexity is established by using a *graph-structured stack*, which is a generalization of the notion of *parse stack*, in which pointers are used to connect stack elements. If nondeterminism occurs, then the search paths are investigated simultaneously, where the initial part of the parse stack which is common to all search paths is represented only once. If two search paths share the state of the top elements of their imaginary individual parse stacks, then the top element is represented only once, so that any computation which thereupon pushes elements onto the stack is performed only once.

Another useful property of GLR parsing is that the output is a concise representation of all possible parses, the so called *parse forest*, which can be seen as a generalization of the notion of *parse tree*. (By some authors, parse forests are more specifically called *shared, shared-packed*, or *packed shared* (parse) forests.) The parse forests produced by the algorithm can be represented using $\mathcal{O}(n^{p+1})$ space. Efficient decoration of parse forests with

attribute values will be investigated in Chapter 6.

There are however some drawbacks to GLR parsing. In order of decreasing importance, these are:

- The parsing technique is based on the use of LR tables, which may be very large for grammars describing natural languages.[1] Related to this is the large amount of time needed to construct a parser. Incremental construction of parsers may in some cases alleviate this problem [Rek92].

- The parse forests produced by the algorithm are not as compact as they might be. This is because packing of subtrees is guided by the merging of search paths due to equal LR states, instead of by the equality of the derived nonterminals. The solution presented in [Rek92] implies much computational overhead.

- Adapting the technique to arbitrary grammars requires the generalization to *cyclic* graph-structured stacks [NF91], which may complicate the implementation.

- A minor disadvantage is that the theoretical time complexity worsens if p becomes larger. The solution given in [Kip91] to obtain a variant of the parsing technique which has a fixed time complexity of $\mathcal{O}(n^3)$, independent of p, implies an overhead in computation costs which worsens instead of improves the time complexity in practical cases.

These disadvantages of generalized LR parsing are mainly consequences of the LR parsing technique, rather than consequences of the use of graph-structured stacks and parse forests.

Lang [Lan74] gives a general construction of deterministic parsing algorithms from non-deterministic push-down automata. The data structures produced have a strong similarity to parse forests, as argued in [BL89, Lan91a].

The idea of a graph-structured stack, however, does not immediately follow from Lang's construction. Instead, Lang uses the abstract notion of a table to store information, without trying to find the best implementation for this table.[2]

One of the parsing techniques which can with some minor difficulties be derived from the construction of Lang is *generalized left-corner parsing* (henceforth abbreviated to *GLC parsing*).[3] The starting-point is left-corner parsing, which was first formally defined in [RL70]. Generalized left-corner parsing, albeit under a different name, has first been investigated in [Pra75]. (See also [TSM79, Bea83, UOKT84, SodA92].) In [Sha91] it was shown that the parsing technique can be a serious rival to generalized LR parsing with regard to

[1] Purdom [Pur74] has argued that grammars for programming languages require LR tables which have a size which is about linear in the size of the grammar. It is generally considered doubtful that similar observations can be made for grammars for natural languages.

[2] Sikkel [Sik90] has argued that the way in which the table is implemented (using a two-dimensional matrix as in case of Earley's algorithm or using a graph-structured stack) is only of secondary importance to the global behaviour of the parsing algorithm.

[3] The term "generalized left-corner parsing" has been used before by Demers [Dem77] for a different parsing technique. Demers generalizes the left corner of a right-hand side to be a prefix of a rhs which does not necessarily consist of one member, whereas *we* generalize LC parsing with zero lookahead to grammars which are not LC(0).

the time complexities. (Other papers discussing the time complexity of GLC parsing are [Slo81, UOKT84, Wir87, BvN93, Car94].)

A functional variant of GLC parsing for definite clause grammars has been discussed in [MS87]. This algorithm does not achieve a polynomial time complexity however, because no "packing" takes place. (This is reminiscent of some algorithms in [Kay86], which tries to unify Earley's algorithm and GLC parsing.)

A variant of Earley's algorithm discussed in [Lei90] also is very similar to GLC parsing although the top-down nature of Earley's algorithm is preserved. It was inspired by the algorithm in [Kil84], which can be seen as GLC parsing without top-down filtering (see Section 2.6).

GLC parsing has been rediscovered a number of times (e.g. in [Lee89, Lee92a], [Sch91], and [Per91]), but without any mention of the connection with LC parsing, which made the presentations unnecessarily difficult to understand. This also prevented discovery of a number of optimizations which are obvious from the viewpoint of left-corner parsing.

In this paper we reinvestigate GLC parsing in combination with graph-structured stacks and parse forests. It is shown that this parsing technique is not subject to the four disadvantages of the GLR algorithm of Tomita.

The structure of this paper is as follows. In Section 2.2 we explain nondeterministic LC parsing. This parsing algorithm is the starting-point of Section 2.3, which shows how a deterministic algorithm can be defined which uses a graph-structured stack and produces parse forests. Section 2.4 discusses how this generalized LC parsing algorithm can be adapted to arbitrary context-free grammars.

How the algorithm can be improved to operate in cubic time is shown in Section 2.5. The improved algorithm produces parse forests in a non-standard representation, which requires only cubic space.

One more class of optimizations is discussed in Section 2.6. Preliminary results from an implementation of our algorithm are discussed in Section 2.7.

2.2 Left-corner parsing

Before we give an informal introduction to LC parsing, we first define some notions strongly connected with this kind of parsing.

We define a *spine* to be a path in a parse tree which begins at some node which is not the first son of its father (or which does not have a father in the case of the root), then proceeds downwards every time taking the leftmost son, and finally ends in a leaf.

Recall the definition of the relation $\angle^*$ from Section 1.1. Informally, we have that $\mathsf{B}\ \angle^*\ \mathsf{A}$ if and only if it is possible to have a spine in some parse tree in which B occurs below A (or $\mathsf{B} = \mathsf{A}$).

We define the set *GOAL* to be the set consisting of S, the start symbol, and of all nonterminals A which occur in a rule of the form $\mathsf{B} \rightarrow \alpha\ \mathsf{A}\ \beta$ where α is not ϵ. Informally, a nonterminal is in *GOAL* if and only if it may occur at the first node of some spine.

We explain LC parsing by means of the small context-free grammar below. No claims are made about the linguistic relevance of this grammar. Note that we have transformed lexical ambiguity into grammatical ambiguity by introducing the nonterminals VorN and VorP.

S → NP VP
S → S PP
NP → "time"
NP → "an" "arrow"
NP → NP NP
NP → VorN
VP → VorN
VP → VorP NP
PP → VorP NP
VorN → "flies"
VorP → "like"

The algorithm reads the input from left to right. The elements on the parse stack are either nonterminals from *GOAL* (the *goal elements*), items (the *item elements*), or nonterminals between brackets (the *corners*). *Items* consist of a rule in which a dot has been inserted somewhere in the rhs to separate the members which have been recognized from those which have not. The corners represent completed subparses, the goals represent as yet unknown subparses which are needed by the context.

Initially, the parse stack consists only of the start symbol, which is the first goal, as indicated in Figure 2.1. The indicated parse corresponds with one of the two possible readings of "time flies like an arrow" according to the grammar above.

In the first step of this parse, it is determined that the first symbol of the input "time" is a left corner of the goal S. Because the symbol on the lhs of the rule NP → "time" is also a left corner of S, the algorithm determines that this rule may be used to construct the lowest part of a spine from S. Consequently [NP → "time" •] is pushed onto the stack. The dot indicates that the preceding part has been read from the input.

In the second step, the algorithm detects that the dot in the item on top of the stack is at the end of the rhs, which indicates that all members in the rhs have been recognized, and that therefore an occurrence of NP has been found. In the third step, a rule is sought which has NP as first member in the rhs. Furthermore, the lhs symbol of that rule must be a left corner of the goal S. These requirements are satisfied by NP → NP NP and consequently [NP → NP • NP] is pushed onto the stack, where the dot indicates that the first member has already been recognized. Subsequently, NP is pushed onto the stack to indicate that the first member after the dot is our new goal.

Steps 4 up to 7 are straightforward. In Step 8 two elements are popped from the stack. This is done because the goal NP was fulfilled by recognition of an occurrence of NP. The dot in [NP → NP • NP] is shifted one position to indicate that also the second NP has been recognized. The remaining steps are again straightforward.

Formally, we define a *nondeterministic LC parser* by the parsing steps which are possible according to the following clauses:[4]

1a. If the element on top of the stack is the nonterminal A and if the first symbol of the remaining input is t, then we may remove t from the input and push an item of the form [B → t • α] onto the stack, provided B $\angle^*$ A.

[4]The construction of parse trees is not explicitly given until Section 2.3.

Step	Parse stack	Input read
	S	
1	S [NP → "time" •]	"time"
2	S (NP)	
3	S [NP → NP • NP] NP	
4	S [NP → NP • NP] NP [VorN → "flies" •]	"flies"
5	S [NP → NP • NP] NP (VorN)	
6	S [NP → NP • NP] NP [NP → VorN •]	
7	S [NP → NP • NP] NP (NP)	
8	S [NP → NP NP •]	
9	S (NP)	
10	S [S → NP • VP] VP	
11	S [S → NP • VP] VP [VorP → "like" •]	"like"
12	S [S → NP • VP] VP (VorP)	
13	S [S → NP • VP] VP [VP → VorP • NP] NP	
14	S [S → NP • VP] VP [VP → VorP • NP] NP [NP → "an" • "arrow"]	"an"
15	S [S → NP • VP] VP [VP → VorP • NP] NP [NP → "an" "arrow" •]	"arrow"
16	S [S → NP • VP] VP [VP → VorP • NP] NP (NP)	
17	S [S → NP • VP] VP [VP → VorP NP •]	
18	S [S → NP • VP] VP (VP)	
19	S [S → NP VP •]	
20	S (S)	

Figure 2.1: One possible sequence of parsing steps while reading "time flies like an arrow"

1b. If the element on top of the stack is the nonterminal A, then we may push an item of the form [B → •] onto the stack, provided B $\angle^*$ A. (The item [B → •] is derived from an epsilon rule B → ϵ.)

2. If the element on top of the stack is the item [A → α • t β] and if the first symbol of the remaining input is t, then we remove t from the input and replace the item by the item [A → α t • β].

3. If the element on top of the stack is the item [A → α •], then we replace the item by the corner (A).

4. If the top-most two elements on the stack are B (A), then we may replace the corner by an item of the form [C → A • β], provided C $\angle^*$ B.

5. If the top-most three elements on the stack are [B → β • A γ] A (A), then we may replace these three elements by the item [B → β A • γ].

6. If a step according to one of the previous clauses ends with an item [A → α • B β] on top of the stack, where B is a nonterminal, then we subsequently push B onto the stack.

7. If the stack consists only of the two elements S (S) and if the input has been completely read, then we may successfully terminate the parsing process.

The nondeterministic LC parsing algorithm defined above uses one symbol of lookahead in case of terminal left corners. The algorithm is therefore deterministic for the LC(0) grammars, according to the definition of LC(k) grammars in [SSU79]. (This definition is incompatible with that of [RL70].)

The exact formulation of the algorithm above is chosen so as to simplify the treatment of generalized LC parsing in the next section. (We explained in Section 1.2.9.1 that in the case of Earley's algorithm strict separation between three categories of stack elements (cf. goal elements, item elements and corners) is necessary to obtain a linear time complexity, expressed in the size of the grammar. For GLC parsing however, a linear time complexity cannot be achieved when using top-down filtering (see Section 2.6). The gain of having the three categories of stack elements is here therefore no more than a constant factor.)

2.3 Generalizing left-corner parsing

The construction of Lang can be used to form deterministic table-driven parsing algorithms from nondeterministic push-down automata. Because left-corner parsers are also push-down automata, Lang's construction can also be applied to formulate a table-driven parsing algorithm based on LC parsing.

The parsing algorithm we propose in this paper does however not follow straightforwardly from Lang's construction. If we applied the construction directly, then not as much sharing would be provided as we would like. This is caused by the fact that sharing of computation of different search paths is interrupted if different elements occur on top of the stack (or just beneath the top if elements below the top are investigated).

To explain this more carefully we focus on Clause 4 of the nondeterministic LC parser. Assume the following situation. Two different search paths have at the same time the same corner (A) on top of the stack. The goal elements (say B' and B'') below that corner however are different in both search paths.

This means that the step which replaces (A) by [C $\rightarrow$ A $\bullet$ β], which is done for both search paths (provided both C $\angle^*$ B' and C $\angle^*$ B''), is done separately because B' and B'' differ. This is unfortunate because sharing of computation in this case is desirable both for efficiency reasons but also because it would simplify the construction of a most-compact parse forest.

Related to the fact that we propose to implement the parse table by means of a graph-structured stack, our solution to this problem lies in the introduction of goal elements consisting of *sets* of nonterminals from $GOAL$, instead of single nonterminals from $GOAL$.

As an example, Figure 2.2 shows the state of the graph-structured stack for the situation just after reading "time flies". Note that this state represents the states of two different search paths of a nondeterministic LC parser after reading "time flies", one of which is the state after Step 4 in Figure 2.1. We see that the goals NP and VP are merged in one goal element so that there is only one edge from the item element labelled with [VorN $\rightarrow$ "flies" $\bullet$] to those goals.

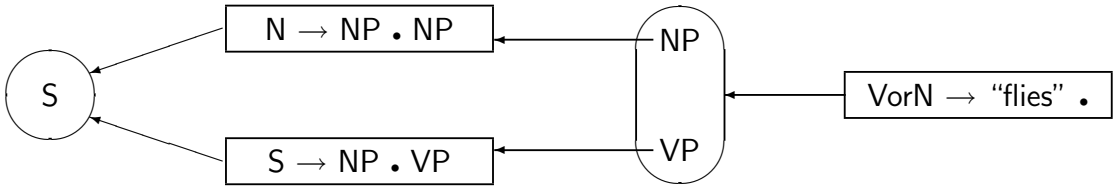


Figure 2.2: The graph-structured stack after reading "time flies"

Merging goals in one stack element is of course only useful if those goals have at least one left corner in common. For the simplicity of the algorithm, we even allow merging of two goals in one goal element if these goals have anything to do with each other with respect to the left-corner relation $\angle^*$.

Formally, we define an equivalence relation $\sim$ on nonterminals, which is the reflexive, transitive, and symmetric closure of $\angle$. An equivalence class of this relation which includes nonterminal A will be denoted by [A]. Each goal element will now consist of a subset of some equivalence class of $\sim$.

In the running example, the goal elements consist of subsets of $\{\mathsf{S}, \mathsf{NP}, \mathsf{VP}, \mathsf{PP}\}$, which is the only equivalence class in this example.

Figures 2.3 and 2.4 give the complete generalized LC parsing algorithm. At this stage we do not want to complicate the algorithm by allowing epsilon rules in the grammar. Consequently, Clause 1b of the nondeterministic LC parser will have no corresponding piece of code in the GLC parsing algorithm. For the other clauses, we will indicate where they can be retraced in the new algorithm. In Section 2.4 we explain how our algorithm can be extended so that also grammars with epsilon rules can be handled.

The nodes and arrows in the parse forest are constructed by means of two functions:

MAKE_NODE (X) constructs a node with label X, which is a terminal or nonterminal. It returns (the address of) that node.

A node is associated with a collection of lists of *sons*, which are other nodes in the forest. Each list represents an alternative derivation of the nonterminal with which the node is labelled. Initially, a node is associated with an empty collection of lists of sons. ADD_SUBNODE (m, l) adds a list of sons l to the node m.

In the algorithm, an item element el labelled with $[\mathsf{A} \rightarrow \mathsf{X}_1 \ldots \mathsf{X}_m \bullet \alpha]$ is associated with a list of nodes deriving $\mathsf{X}_1, \ldots, \mathsf{X}_m$. This list is accessed by SONS (el). A list consisting of exactly one node m is denoted by $<m>$, and list concatenation is denoted by the operator $+$.

A goal element g contains for every nonterminal A such that $\mathsf{A} \angle^* \mathsf{P}$ for some P in g a value NODE (g, A), which is the node representing the derivations of A found at the current input position, provided any such derivations exist, and NODE (g, A) is NIL otherwise. Note that this obviates the need for separate stack elements for the corners.

In the graph-structured stack there may be an edge from an item element to a unique goal element, and from a goal in a goal element to a number of item elements. For item element el, SUCCESSOR (el) yields the unique goal element to which there is an edge from el. For goal element g and goal P in g, SUCCESSORS (g, P) yields the zero or more item elements to which there is an edge from P in g.

PARSE:

- $r \Leftarrow$ NIL
- Create goal element g consisting of S, the start symbol
- $\Gamma \Leftarrow \{g\}$
- $\mathrm{I} \Leftarrow \emptyset$
- $\mathrm{F} \Leftarrow \emptyset$
- **for** $i \Leftarrow 0$ **to** n **do** PARSE_WORD
- **return** r, as the root of the parse forest

PARSE_WORD:

- $\Gamma_{next} \Leftarrow \emptyset$
- $\mathrm{I}_{next} \Leftarrow \emptyset$
- **for all** pairs $(g, \mathsf{A}) \in \mathrm{F}$ **do**
  - NODE $(g, \mathsf{A}) \Leftarrow$ NIL
- $\mathrm{F} \Leftarrow \emptyset$
- $t \Leftarrow$ MAKE_NODE (a_i)
- FIND_CORNERS (t)
- SHIFT (t)
- $\Gamma \Leftarrow \Gamma_{next}$
- $\mathrm{I} \Leftarrow \mathrm{I}_{next}$

FIND_CORNERS (t): /* cf. Clause 1a of the nondeterministic LC parser */

- **for all** goal elements g in Γ containing goals in class $[\mathsf{B}]$ **do**
  - **for all** rules $\mathsf{A} \rightarrow \mathsf{a}_i\ \alpha$ such that $\mathsf{A} \in [\mathsf{B}]$ **do**
    - **if** $\mathsf{A} \angle^* \mathsf{P}$ for some goal P in g /* top-down filtering */
      then
      - MAKE_ITEM_ELEM $([\mathsf{A} \rightarrow \mathsf{a}_i \bullet \alpha], <t>, g)$

SHIFT (t): /* cf. Clause 2 */

- **for all** item elements el in I labelled with $[\mathsf{A} \rightarrow \alpha \bullet \mathsf{a}_i\ \beta]$ **do**
  - MAKE_ITEM_ELEM $([\mathsf{A} \rightarrow \alpha\ \mathsf{a}_i \bullet \beta]$, SONS $(el) + <t>$, SUCCESSOR $(el))$

MAKE_ITEM_ELEM $([\mathsf{A} \rightarrow \alpha \bullet \beta], l, g)$:

- Create item element el labelled with $[\mathsf{A} \rightarrow \alpha \bullet \beta]$
- SONS $(el) \Leftarrow l$
- Create an edge from el to g
- **if** $\beta = \epsilon$
  then
  - REDUCE (el)
  elseif $\beta = \mathsf{t}\gamma$, where t is a terminal
  then
  - $\mathrm{I}_{next} \Leftarrow \mathrm{I}_{next} \cup \{el\}$
  elseif $\beta = \mathsf{B}\gamma$, where B is a nonterminal /* cf. Clause 6 */
  then
  - MAKE_GOAL (B, el)

Figure 2.3: The generalized LC parsing algorithm

MAKE_GOAL (A, *el*):

- **if** there is a goal element g in Γ_{next} containing goals in class [A]
  then
  - Add goal A to g (provided it is not already there)

  else
  - Create goal element g consisting of A
  - Add g to Γ_{next}
- Create an edge from A in g to *el*

REDUCE (*el*):

- Assume the label of *el* is $[A \rightarrow \alpha \bullet]$
- Assume SUCCESSOR (*el*) is g
- **if** NODE (g, A) = NIL /* cf. Clause 3 */
  then
  - $m \Leftarrow$ MAKE_NODE (A)
  - NODE (g, A) $\Leftarrow m$
  - F $\Leftarrow$ F $\cup$ {(g, A)}
  - **for all** rules $B \rightarrow A\ \beta$ **do** /* cf. Clause 4 */
    - **if** $B \angle^* P$ for some goal P in g /* top-down filtering */
      then
      - MAKE_ITEM_ELEM ($[B \rightarrow A \bullet \beta]$, <$m$>, g)
  - **if** A is a goal in g
    then
    - **if** SUCCESSORS (g, A) $\neq \emptyset$
      then

      /* cf. Clause 5 */
      - **for all** *el'* $\in$ SUCCESSORS (g, A) labelled with $[B \rightarrow \beta \bullet A\ \gamma]$ **do**
        - MAKE_ITEM_ELEM ($[B \rightarrow \beta\ A \bullet \gamma]$,
          SONS (*el'*) + <m>, SUCCESSOR (*el'*))

      elseif $i = n$ /* cf. Clause 7 */
      then
      - $r \Leftarrow m$
- ADD_SUBNODE (NODE (g, A), SONS (*el*))

Figure 2.4: The generalized LC parsing algorithm (continued)

The global variables used by the algorithm are the following.

$\mathsf{a}_0\ \mathsf{a}_1\ \ldots\ \mathsf{a}_n$ The symbols in the input string.

i The current input position.

r The root of the parse forest. It has the value NIL at the end of the algorithm if no parse has been found.

Γ **and** Γ_{next} The sets of goal elements containing goals to be fulfilled from the current and next input position on, respectively.

I **and** I_{next} The sets of item elements labelled with $[\mathsf{A} \rightarrow \alpha \bullet \mathsf{t}\ \beta]$ such that a shift may be performed through t at the current and next input position, respectively.

F The set of pairs (g, A) such that a derivation from A has been found for g at the current input position. In other words, F is the set of all pairs (g, A) such that NODE (g, A) $\neq$ NIL.

The graph-structured stack (which is initially empty) and the rules of the grammar are implicit global data structures.

In a straightforward implementation, the relation $\angle^*$ is recorded by means of one large $s' \times s$ boolean matrix, where s is the number of nonterminals in the grammar, and s' is the number of elements in *GOAL*. We can do better however by using the fact that $\mathsf{A}\ \angle^*\ \mathsf{B}$ is never true if $\mathsf{A} \not\sim \mathsf{B}$. We propose the storage of $\angle^*$ for every equivalence class of $\sim$ separately, i.e. we store one $t' \times t$ boolean matrix for every class of $\sim$ with t members, t' of which are in *GOAL*.

We furthermore need a list of all rules $\mathsf{A} \rightarrow \mathsf{X}\ \alpha$ for each terminal and nonterminal X. A small optimization of top-town filtering (see also Section 2.6) can be achieved by grouping the rules in these lists according to the left-hand sides A.

Note that the storage of the relation $\angle^*$ is the main obstacle to a linear-sized parser.

The time needed to generate a parser is determined by the time needed to compute $\angle^*$ and the classes of $\sim$, which is quadratic in the size of the grammar.

2.4 An algorithm for arbitrary context-free grammars

The generalized LC parsing algorithm from the previous section is only specified for grammars without epsilon rules. Allowing epsilon rules would not only complicate the algorithm but would for some grammars also introduce the danger of non-termination of the parsing process.

There are two sources of non-termination for nondeterministic LC and LR parsing: cyclicity and hidden left-recursion (both concepts were defined in Section 1.1). Both sources of non-termination are studied extensively in Chapters 4 and 5.

An obvious way to avoid non-termination for nondeterministic LC parsers in case of hidden left-recursive grammars is the following. We generalize the relation $\angle$ so that $\mathsf{B} \angle \mathsf{A}$ if and only if there is a rule $\mathsf{A} \rightarrow \mu\ \mathsf{B}\ \beta$, where μ is a (possibly empty) sequence of grammar symbols such that $\mu \rightarrow^* \epsilon$. Clause 1b is eliminated and to compensate this, Clauses 1a and

4 are modified so that they take into account prefixes of right-hand sides which generate the empty string:

1a. If the element on top of the stack is the nonterminal A and if the first symbol of the remaining input is t, then we may remove t from the input and push an item of the form $[\mathsf{B} \rightarrow \mu\ \mathsf{t} \bullet \alpha]$ onto the stack, provided $\mathsf{B} \angle^* \mathsf{A}$ and $\mu \rightarrow^* \epsilon$.

4. If the top-most two elements on the stack are B (A), then we may replace the corner by an item of the form $[\mathsf{C} \rightarrow \mu\ \mathsf{A} \bullet \beta]$, provided $\mathsf{C} \angle^* \mathsf{B}$ and $\mu \rightarrow^* \epsilon$.

These clauses now allow for nullable members at the beginning of right-hand sides. To allow for other nullable members we need an extra eighth clause: [5]

8. If the element on top of the stack is the item $[\mathsf{A} \rightarrow \alpha \bullet \mathsf{B}\ \beta]$, then we may replace this item by the item $[\mathsf{A} \rightarrow \alpha\ \mathsf{B} \bullet \beta]$, provided $\mathsf{B} \rightarrow^* \epsilon$.

The same idea can be used in a straightforward way to make generalized LC parsing suitable for hidden left-recursive grammars, similar to the way this is handled in [Sch91] and [Lee92a]. The only technical problem is that, in order to be able to construct a complete parse forest, we need precomputed subforests which derive the empty string in every way from nullable nonterminals. This precomputation consists of performing $m_{\mathsf{A}} \Leftarrow$ MAKE_NODE (A) for each nullable nonterminal A, (where m_{A} are specific variables, one for each nonterminal A) and subsequently performing ADD_SUBNODE (m_{A}, $<m_{\mathsf{B}_1}, \ldots, m_{\mathsf{B}_k}>$) for each rule $\mathsf{A} \rightarrow \mathsf{B}_1 \ \ldots \ \mathsf{B}_k$ consisting only of nullable nonterminals. The variables m_{A} now contain pointers to the required subforests.

GLC parsing is guaranteed to terminate also for cyclic grammars, in which case the infinite number of parses is reflected by cyclic forests, which are also discussed in [NF91, Lan88c].

2.5 Parsing in cubic time

The size of parse forests, even of those which are optimally dense, can be more than cubic in the length of the input. More precisely, the number of nodes in a parse forest is $\mathcal{O}(n^{p+1})$, where p is the length of the rhs of the longest rule.

Using the normal representation of parse forests does therefore not allow cubic parsing algorithms for arbitrary grammars. There is however a kind of shorthand for parse forests which allows a representation which only requires a cubic amount of space.

For example, suppose that of some rule $\mathsf{A} \rightarrow \alpha\ \beta$, the prefix α of the rhs derives the same part of the input in more than one way, then these derivations may be combined in a new kind of packed node. Instead of the multiple derivations from α, this packed node is then combined with the derivations from β deriving subsequent input. We call packing of derivations from prefixes (or suffixes) of right-hand sides *subpacking* to distinguish this from normal packing of derivations from one nonterminal. Subpacking has been discussed in [BL89, Lei90, LAKA92]; see also [She76].

[5] Actually, a ninth clause is necessary to handle the special case where S, the start symbol, is nullable, and the input is empty. We omit this clause for the sake of clarity.

Example 2.5.1 Assume the following situation. We have a rule $\mathsf{A} \rightarrow \mathsf{B}\ \mathsf{C}\ \mathsf{D}\ \mathsf{E}$. A substring of the input is $\mathsf{a}_1 \ \ldots \ \mathsf{a}_v \ \mathsf{a}_{v+1} \ \ldots \ \mathsf{a}_w$. There are k positions i in the first half ($0 \leq i \leq v$) such that there is a subparse from B of $\mathsf{a}_1 \ \ldots \ \mathsf{a}_i$ and a subparse from C of $\mathsf{a}_{i+1} \ \ldots \ \mathsf{a}_v$. In the same way, there are m positions j in the second half ($v \leq j \leq w$) such that there is a subparse from D of $\mathsf{a}_{v+1} \ \ldots \ \mathsf{a}_j$ and a subparse from E of $\mathsf{a}_{j+1} \ \ldots \ \mathsf{a}_w$. The situation for $k = 3$ and $m = 3$ is sketched in Figure 2.5.

If only normal packing for nonterminals is used, a parse forest requires $\mathcal{O}(k * m)$ space here. With subpacking however only $\mathcal{O}(k+m)$ space is required, as Figure 2.6 demonstrates. □

Figure 2.5: From B and C there are k ($k = 3$) pairs of connecting subparses, together covering $\mathsf{a}_1 \ \ldots \ \mathsf{a}_v$. Likewise, from D and E there are m ($m = 3$) pairs of connecting subparses, together covering $\mathsf{a}_{v+1} \ \ldots \ \mathsf{a}_w$.

Figure 2.6: Packing and subpacking

Connected with cubic representation of parse forests is cubic parsing. The GLC parsing algorithm in Section 2.3 has a time complexity of $\mathcal{O}(n^{p+1})$. The algorithm can be easily changed so that, with little overhead, the time complexity is reduced to $\mathcal{O}(n^3)$, similar to the algorithms in [Per91], [Sch91], and [Lee92a], and the algorithm produces parse forests with subpacking, which require only $\mathcal{O}(n^3)$ space for storage.

We consider how this can be accomplished. First we define the *origin* of an item element labelled with $[\mathsf{A} \rightarrow \alpha \bullet \beta]$ to be the rule $\mathsf{A} \rightarrow \alpha\ \beta$. Now suppose that two item elements el_1 and el_2 with the same origin, with the dot at the same position and with the same successor are created at the same input position, then we may perform subpacking for the prefix of the rhs before the dot. From then on, we only need one of the item elements el_1 and el_2 for continuing the parsing process.

Identification of those item elements which have one and the same goal element as successors and which have the same labels cannot be done in an efficient way. Therefore we propose to introduce a new kind of stack element which takes over the role of all former item elements whose successors are one and the same goal element and which have the same origin.

We leave the details to the imagination of the reader.

2.6 Optimization of top-down filtering

One of the most time-costly activities of generalized LC parsing is the check whether for a goal element g and a nonterminal A there is some goal P in g such that $\mathsf{A} \angle^* \mathsf{P}$. This check, which is sometimes called *top-down filtering*, occurs in the routines FIND_CORNERS and REDUCE. We propose some optimizations to reduce the number of goals P in g for which $\mathsf{A} \angle^* \mathsf{P}$ has to be checked.

The most straightforward optimization consists of annotating every edge from an item element labelled with $[\mathsf{A} \rightarrow \alpha \bullet \beta]$ to a goal element g with the subset of goals in g which does not include those goals P for which $\mathsf{A} \angle^* \mathsf{P}$ has already been found to be false. This is the set of goals in g which are actually useful in top-down filtering when a new item element labelled with $[\mathsf{B} \rightarrow \mathsf{A} \bullet \gamma]$ is created during a REDUCE (see the piece of code in REDUCE corresponding with Clause 4 of the nondeterministic LC parser). The idea is that if $\mathsf{A} \angle^* \mathsf{P}$ does not hold for goal P in g, then neither does $\mathsf{B} \angle^* \mathsf{P}$ if $\mathsf{A} \angle \mathsf{B}$. This optimization can be realized very easily if sets of goals are implemented as lists.

A second optimization is useful if $\angle$ is such that there are many nonterminals A such that there is only one B with $\mathsf{A} \angle \mathsf{B}$. In case we have such a nonterminal A which is not a goal, then no top-down filtering needs to be performed when a new item element labelled with $[\mathsf{B} \rightarrow \mathsf{A} \bullet \alpha]$ is created during a REDUCE. This can be explained by the fact that if for some goal P we have $\mathsf{A} \angle^* \mathsf{P}$, and if $\mathsf{A} \neq \mathsf{P}$, and if there is only one B such that $\mathsf{A} \angle \mathsf{B}$, then we already know that $\mathsf{B} \angle^* \mathsf{P}$.

There are many more of these optimizations but not all of these give better performance in all cases. It depends heavily on the properties of $\angle$ whether the gain in time while performing the actual top-down filtering (i.e. performing the tests $\mathsf{A} \angle^* \mathsf{P}$ for some P in a particular subset of the goals in a goal element g) outweighs the time needed to set up extra administration for the purpose of reducing those subsets of the goals.

2.7 Preliminary results

The author has implemented a GLC parser. The algorithm as presented in this paper has been implemented almost literally, with the treatment of epsilon rules as suggested in Section 2.4. A small adaptation has been made in order to deal with terminals of different lengths.

The author has also followed with great interest the development of a GLR parser by colleagues in the same department. Because both systems have been implemented using different programming languages, fair comparison of the two systems is difficult. Specific problems which occurred concerning the efficient calculation of LR tables and the correct treatment of epsilon rules for GLR parsing suggest that GLR parsing requires more effort to implement than GLC parsing.

Preliminary tests show that the division of nonterminals into equivalence classes yields disappointing results. In all tested cases, one large class contained most of the nonterminals.

The first optimization discussed in Section 2.6 proved to be very useful. The number of goals which had to be considered could in some cases be reduced to one third.

2.8 Conclusions

We have discussed a parsing algorithm for context-free grammars called *generalized LC parsing*. This parsing algorithm has the following advantages over generalized LR parsing (in order of decreasing importance).

- The size of a parser is much smaller; if we neglect the storage of the relation $\angle^*$, the size is even linear in the size of the grammar. Related to this, only a small amount of time is needed to generate a parser.
- The generated parse forests are as compact as possible.
- Cyclic and hidden left-recursive grammars can be handled more easily and more efficiently (Section 2.4).
- As Section 2.5 shows, GLC parsing can more easily be made to run in cubic time for arbitrary context-free grammars. Furthermore, this can be done without much loss of efficiency in practical cases.

Because LR parsing is a more refined form of parsing than LC parsing, generalized LR parsing may at least for some grammars be more efficient than generalized LC parsing.[6] However, we feel that this does not outweigh the disadvantages of the large sizes and generation times of LR parsers in general, which renders GLR parsing unfeasible in some natural language applications.

GLC parsing does not suffer from these defects. We therefore propose this parsing algorithm as a reasonable alternative to GLR parsing. Because of the small generation time of GLC parsers, we expect this kind of parsing to be particularly appropriate during

[6] The ratio between the time complexities of GLC parsing and GLR parsing is smaller than some constant, which is dependent on the grammar.

the development of grammars, when grammars change often and consequently new parsers have to be generated many times.

As we have shown in this paper, the implementation of GLC parsing using a graph-structured stack allows many optimizations. These optimizations would be less straightforward and possibly less effective if a two-dimensional matrix was used for the implementation of the parse table. Furthermore, matrices require a large amount of space, especially for long input, causing overhead for initialization (at least if no optimizations are used).

In contrast, the time and space requirements of GLC parsing using a graph-structured stack are only a negligible quantity above that of nondeterministic LC parsing if no nondeterminism occurs (e.g. if the grammar is LC(0)). Only in the worst-case does a graph-structured stack require the same amount of space as a matrix.

In this paper we have not considered GLC parsing with more lookahead than one symbol for terminal left corners. The reason for this is that we feel that one of the main advantages of our parsing algorithm over GLR parsing is the small sizes of the parsers. Adding more lookahead requires larger tables and may therefore reduce the advantage of generalized LC parsing over its LR counterpart.

On the other hand, the phenomenon reported in [BL89] and [Lan91b] that the time complexity of GLR parsing sometimes worsens if more lookahead is used, does possibly not apply to GLC parsing. For GLR parsing, more lookahead may result in more LR states, which may result in less sharing of computation. For GLC parsing there is however no relation between the amount of lookahead and the amount of sharing of computation.

Therefore, a judicious use of extra lookahead may on the whole be advantageous to the usefulness of GLC parsing.

Chapter 3

An Optimal Tabular Parsing Algorithm

In this chapter we relate a number of parsing algorithms which have been developed in very different areas of parsing theory. We show that these algorithms are based on the same underlying ideas. The different parsing algorithms which are related include deterministic algorithms, tabular algorithms, and a parallel algorithm.

By relating existing ideas, we hope to provide an opportunity to improve some algorithms based on features of others. A second purpose of this chapter is to answer a question from the area of tabular parsing, namely how to obtain a parsing algorithm with the property that the table will contain as little entries as possible, but without the possibility that two entries represent the same subderivation.

3.1 Introduction

Left-corner (LC) parsing is a parsing technique which has been used in different guises in various areas of computer science. Deterministic LC parsing with k symbols of lookahead can handle the class of LC(k) grammars. Since LC parsing is a very simple parsing technique and at the same time is able to deal with left recursion, it is often used as an alternative to top-down (TD) parsing, which cannot handle left recursion and is generally less efficient.

Non-deterministic LC parsing is the foundation of a very efficient parsing algorithm (Chapter 2). It has one disadvantage however, which becomes noticeable when the grammar contains many rules whose right-hand sides begin with the same few grammars symbols, e.g.

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots$$

where α is not the empty string. After an LC parser has recognized the first symbol X of such an α, it will as next step predict all aforementioned rules (we abstain from discussing lookahead). This amounts to much nondeterminism, which is detrimental both to the time-complexity and the space-complexity.

The much older predictive LR (PLR) parsing technique has been

The predictive LR (PLR) parsing technique has been introduced in [SSU79] as a way of constructing small deterministic parsers with k symbols of lookahead for a proper subset

of the LR(k) grammars [SSS90]. These PLR(k) grammars properly contain the LC(k) grammars.

A PLR parser can be seen as an LC parser for a transformed grammar in which some common prefixes have been eliminated and in this way PLR parsing is a partial solution to our problem. However, PLR parsing only solves the problem of common prefixes when the left-hand sides of the rules are the same. In case we have e.g. the rules $A \rightarrow \alpha\beta_1$ and $B \rightarrow \alpha\beta_2$, where again α is not the empty string but now $A \neq B$, then even PLR parsing will in some cases not be able to avoid nondeterminism after the first symbol of α is recognized. We therefore go one step further and discuss extended LR (ELR) and common-prefix (CP) parsing, which are algorithms capable of dealing with all kinds of common-prefixes without having to resort to unnecessary nondeterminism. ELR and CP parsing are the foundations of tabular parsing algorithms and a parallel parsing algorithm from the existing literature, but they have not been described in their own right.

To the best of the author's knowledge, the various parsing algorithms mentioned above have not been discussed together in the existing literature. The main purpose of this chapter is to make explicit the connections between these algorithms. In this respect, we consider this chapter to be a continuation of Chapter 2, which discussed algorithms that are all derived from left-corner parsing.

A second purpose of this chapter is to show that CP and ELR parsing are obvious solutions to a problem of tabular parsing which can be described as follows. For each parsing algorithm working on a stack there is a realisation using a parse table, where the parse table allows sharing of computation between different search paths. For example, we have Tomita's algorithm [Tom86], which can be seen as a tabular realisation of nondeterministic LR parsing for grammars which do not satisfy an LR property.

At this point we introduce the term *state* to indicate the symbols occurring on the stack of the original algorithm, which also occur as entries in the parse table of its tabular realisation.

In general, powerful algorithms working on a stack correspond to efficient tabular parsing algorithms, provided the grammar can be handled almost deterministically. In case the original algorithm is very nondeterministic for a certain grammar however, sophistication of this algorithm which increases the number of states may lead to an increasing number of entries in the parse table of its tabular counterpart. This can be informally explained by the fact that each state represents the computation of a number of subderivations. If the number of states is increased then it is inevitable that at some point some states represent an overlapping collection of subderivations, which may lead to work being repeated during parsing. Furthermore, the parse forest (a compact representation of all parse trees) which is output by a tabular algorithm may in this case not be optimally dense.

We conclude that we have a tradeoff between the case that the grammar allows almost deterministic parsing and the case that the original algorithm is very nondeterministic for a certain grammar. In the former case, sophistication leads to *less* entries in the table, and in the latter case, sophistication leads to *more* entries, provided this sophistication is realised by an increase in the number of states. This is corroborated by empirical data from [BL89, Lan91b], which deal with tabular LR parsing.

As we will explain, CP and ELR parsing have the nice property that they are more deterministic than most other parsing algorithms for many grammars, but their tabular

realizations can never compute the same subderivation twice. This represents an optimum in a range of possible parsing algorithms.

This chapter is organized as follows. Section 3.2 discusses nondeterministic left-corner parsing, and demonstrates how common prefixes in a grammar may be a source of bad performance for this technique.

A multitude of parsing techniques which exhibit better treatment of common prefixes are discussed in Section 3.3. These techniques, including nondeterministic PLR, ELR, and CP parsing, have their origins in theory of deterministic, parallel, and tabular parsing. The application to parallel and tabular parsing is investigated more closely in Section 3.4. Data structures needed by the parsing techniques are discussed in Section 3.5.

If a grammar contains epsilon rules, then this may complicate the parsing process. How we can deal with these rules is discussed in Section 3.6.

Section 3.7 relates LR parsing to the ideas described in the preceding sections. In Section 3.8 a two-dimensional system is discussed in which some of the parsing techniques mentioned in this paper can be arranged.

The ideas described in this chapter can be generalized to head-driven parsing, as argued in [NS94].

We will take some liberty in describing algorithms from the existing literature, since using the original descriptions would blur the similarities between the algorithms. In particular, we will not treat the use of lookahead, and we will consider all algorithms working on a stack to be nondeterministic. We will only describe *recognition* algorithms. Each of the algorithms can however be easily extended to yield parse trees as a side-effect of recognition.

Most of the notation and terminology in this chapter is explained in Section 1.1.

The notation of rules $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \ldots$ with the same lhs is often simplified to $A \rightarrow \alpha_1 | \alpha_2 | \ldots$

Until Section 3.6 we will tacitly assume that grammars do not contain any epsilon rules.

We say two rules $A \rightarrow \alpha_1$ and $B \rightarrow \alpha_2$ have a *common prefix* β if $\alpha_1 = \beta\gamma_1$ and $\alpha_2 = \beta\gamma_2$, for some γ_1 and γ_2, where $\beta \neq \epsilon$.

A recognition algorithm can be specified by means of a push-down automaton $A = (T, \mathit{Alph}, \mathit{Init}, \vdash, \mathit{Fin})$, which manipulates configurations of the form (Γ, v), where $\Gamma \in \mathit{Alph}^*$ is the stack, and $v \in T^*$ is the remaining input. The stack is constructed from left to right.

The initial configuration is (Init, w), where $\mathit{Init} \in \mathit{Alph}$ is a distinguished stack symbol, and w is the input. The steps of an automaton are specified by means of the relation $\vdash$. Thus, $(\Gamma, v) \vdash (\Gamma', v')$ denotes that (Γ', v') is obtainable from (Γ, v) by one step of the automaton. The reflexive and transitive closure of $\vdash$ is denoted by $\vdash^*$. The input w is accepted if $(\mathit{Init}, w) \vdash^* (\mathit{Fin}, \epsilon)$, where $\mathit{Fin} \in \mathit{Alph}$ is a distinguished stack symbol.

Note that from a recognition algorithm for some fixed grammar which does not use lookahead, we can easily obtain one that does: suppose the algorithm with transition relation $\vdash$ does not use lookahead. We can now define the transition relation $\vdash_k$ of a new algorithm as follows

$$(\Gamma, vw) \vdash_k (\Gamma', w)$$
$$\text{if and only if}$$
$$(\Gamma, vw) \vdash (\Gamma', w) \text{ and } \exists_{w'}[k : vw = k : vw' \wedge (\Gamma', w') \vdash^* (\mathit{Fin}, \epsilon)]$$

where $k : w$ denotes the first k symbols of w if w is longer than k symbols, and w otherwise. In words, a transition using k symbols of lookahead is the same as a usual transition, except that it also checks whether the transition may lead to acceptance of the input as far as only the first k symbols of the remaining input are concerned. Note that the *lookahead check* $\exists_{w'}[k : vw = k : vw' \wedge (\Gamma', w') \vdash^* (Fin, \epsilon)]$ is often not done for all types of steps (for example, for the steps of a top-down recognizer which read input symbols). For practical algorithms the lookahead check is compiled into tables, so that it can be performed in constant time.

Many algorithms such as LL(k), LC(k), and LR(k) recognition can be specified by $\vdash_k$ in this way by taking $\vdash$ to be the straightforward transition relations for LL(0), LC(0), and LR(0) recognition, respectively. For strong LL(k), strong LC(k), and LALR(k) recognition we need a different lookahead check which takes only the top-most symbol of the stack Γ into account. We do not give this form of lookahead check since it is a little technical.

3.2 LC parsing

For the definition of left-corner (LC) recognition we need stack symbols of the form $[A \rightarrow \alpha \bullet \beta]$, where $A \rightarrow \alpha\beta$ is a rule, and $\alpha \neq \epsilon$. Such a symbol is called an *item*.[1] The informal meaning of an item is "The part before the dot has just been recognized, the first symbol after the dot is to be recognized next". For technical reasons we also need the items $[S' \rightarrow \bullet\ S]$ and $[S' \rightarrow S\ \bullet]$, where S' is a fresh symbol. We formally define the set of all items as

$$I^{LC} = \{[A \rightarrow \alpha \bullet \beta] \mid A \rightarrow \alpha\beta \in P^\dagger \wedge (\alpha \neq \epsilon \vee A = S')\}$$

where $P^\dagger$ represents the *augmented* set of rules, consisting of the rules in P plus the extra rule $S' \rightarrow S$.

We now have

Algorithm 14 (Left-corner) $A^{LC} = (T, Alph, Init, \vdash, Fin)$, where $Alph = I^{LC}$, $Init = [S' \rightarrow \bullet\ S]$, $Fin = [S' \rightarrow S\ \bullet]$, and transitions are allowed according to the following clauses.

1. $(\Gamma[B \rightarrow \beta \bullet C\gamma], av) \vdash (\Gamma[B \rightarrow \beta \bullet C\gamma][A \rightarrow a \bullet \alpha], v)$
   where there is $A \rightarrow a\alpha \in P^\dagger$ such that $A \angle^* C$

2. $(\Gamma[A \rightarrow \alpha \bullet a\beta], av) \vdash (\Gamma[A \rightarrow \alpha a \bullet \beta], v)$

3. $(\Gamma[B \rightarrow \beta \bullet C\gamma][A \rightarrow \alpha \bullet], v) \vdash (\Gamma[B \rightarrow \beta \bullet C\gamma][D \rightarrow A \bullet \delta], v)$
   where there is $D \rightarrow A\delta \in P^\dagger$ such that $D \angle^* C$

4. $(\Gamma[B \rightarrow \beta \bullet A\gamma][A \rightarrow \alpha \bullet], v) \vdash (\Gamma[B \rightarrow \beta A \bullet \gamma], v)$

The transition steps can be explained informally as follows. The automaton recognizes derivations in a bottom-up order. The first type of transition states that if symbol C is to be recognized next, we start by recognizing the production in the leftmost "corner" of a derivation from C. The second type of transition reads an input symbol if that symbol is to be recognized next.

[1] Remember that we do not allow epsilon rules until Section 3.6.

The third and fourth clauses both deal with the situation that the item on top of stack indicates that a complete rule has been recognized. The third clause predicts another rule in a bottom-up manner, so that eventually a larger subderivation may be recognized. The fourth rule represents the case that the subderivation obtained so far is already the one needed by the item just underneath the top of stack.

The conditions using the left-corner relation $\angle^*$ in the first and third clauses together form a feature which is called *top-down (TD) filtering*. TD filtering makes sure that subderivations that are being computed bottom-up may eventually grow into subderivations with the required root. TD filtering is not necessary for a correct algorithm, but it reduces nondeterminism, and guarantees the correct-prefix property (see also Section 3.3.3).[2]

Example 3.2.1 Consider the grammar with the following rules:

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \uparrow E \mid T \\ T & \rightarrow & T * F \mid T * * F \mid F \\ F & \rightarrow & a \end{array}$$

It is easy to see that $E \angle E$, $T \angle E$, $T \angle T$, $F \angle T$. The relation $\angle^*$ contains $\angle$ but from the reflexive closure it also contains $F \angle^* F$ and from the transitive closure it also contains $F \angle^* E$.

The recognition of $a * a$ is realised by the following sequence of configurations:

	$[E' \rightarrow \bullet\ E]$	$a * a$
1	$[E' \rightarrow \bullet\ E][F \rightarrow a\ \bullet]$	$* a$
2	$[E' \rightarrow \bullet\ E][T \rightarrow F\ \bullet]$	$* a$
3	$[E' \rightarrow \bullet\ E][T \rightarrow T\ \bullet * F]$	$* a$
4	$[E' \rightarrow \bullet\ E][T \rightarrow T * \bullet\ F]$	a
5	$[E' \rightarrow \bullet\ E][T \rightarrow T * \bullet\ F][F \rightarrow a\ \bullet]$	
6	$[E' \rightarrow \bullet\ E][T \rightarrow T * F\ \bullet]$	
7	$[E' \rightarrow \bullet\ E][E \rightarrow T\ \bullet]$	
8	$[E' \rightarrow E\ \bullet]$	

Note that since the automaton does not use any lookahead, Step 3 may also have replaced $[T \rightarrow F\ \bullet]$ by any other item besides $[T \rightarrow T\ \bullet * F]$ whose rhs starts with T and whose lhs satisfies the condition of top-down filtering with regard to E, i.e. by $[T \rightarrow T\ \bullet * * F]$, $[E \rightarrow T\ \bullet \uparrow E]$, or $[E \rightarrow T\ \bullet]$. □

LC parsing with k symbols of lookahead can handle deterministically the so called LC(k) grammars. This class of grammars is formalized in [RL70].[3] How LC parsing can be improved to handle common *suffixes* efficiently is discussed in [Lee92a].[4] In this chapter we restrict our attention to common *prefixes*.

[2] As argued in [DMAM94], processing of *incorrect* input is made possible by omitting TD filtering for at least a subset of the nonterminals.

[3] In [SSU79] a different definition of the LC(k) grammars may be found, which is not completely equivalent.

[4] Shared computation of common suffixes as discussed in [Lee92a] (see also [Aug93, Definition 3.29 on page 59], [Lee93, Exercise 2 of Chapter 6], and [Bea83, DMAM94]) has the advantages that the size of the parser is smaller, and that tabular realizations are more efficient. For the run-time behaviour of stack algorithms, however, this optimization has no significance.

3.3 PLR, ELR, and CP parsing

In this section we investigate a number of algorithms which exhibit a better treatment of common prefixes.

3.3.1 Predictive LR parsing

Predictive LR (PLR) parsing with k symbols of lookahead was introduced in [SSU79] as an algorithm which yields efficient parsers for a subset of the LR(k) grammars [SSS90] and a superset of the LC(k) grammars. How deterministic PLR parsing succeeds in handling a larger class of grammars (the PLR(k) grammars) than the LC(k) grammars can be explained by identifying PLR parsing for some grammar G with LC parsing for some grammar G' which results after applying a transformation called *left-factoring*.

Left-factoring consists of replacing two or more rules $A \to \alpha\beta_1|\alpha\beta_2|\ldots$ with a common prefix α by the rules $A \to \alpha A'$ and $A' \to \beta_1|\beta_2|\ldots$, where A' is a fresh nonterminal. The effect on LC parsing is that a choice between rules is postponed until after all symbols of α are completely recognized. Investigation of the next k symbols of the remaining input may then allow a choice between the rules to be made deterministically.

The PLR algorithm is formalised in [SSU79] by transforming a PLR(k) grammar into an LL(k) grammar and then assuming the standard realisation of LL(k) parsing. When we consider nondeterministic top-down parsing instead of LL(k) parsing, then we obtain the new formulation of nondeterministic PLR(0) parsing below.

We first need to define another kind of item, viz. of the form $[A \to \alpha]$ such that there is at least one rule of the form $A \to \alpha\beta$ for some β. The set of all such items is formally defined by

$$I^{PLR} = \{[A \to \alpha] \mid A \to \alpha\beta \in P^\dagger \wedge (\alpha \neq \epsilon \vee A = S')\}$$

Informally, an item $[A \to \alpha] \in I^{PLR}$ represents one or more items $[A \to \alpha \bullet \beta] \in I^{LC}$.

We now have

Algorithm 15 (Predictive LR) $A^{PLR} = (T, Alph, Init, \vdash, Fin)$, where $Alph = I^{PLR}$, $Init = [S' \to\ ]$, $Fin = [S' \to S]$, and transitions are allowed according to the following clauses.

1. $(\Gamma[B \to \beta], av) \vdash (\Gamma[B \to \beta][A \to a], v)$
   where there are $A \to a\alpha, B \to \beta C\gamma \in P^\dagger$ such that $A \angle^* C$

2. $(\Gamma[A \to \alpha], av) \vdash (\Gamma[A \to \alpha a], v)$
   where there is $A \to \alpha a\beta \in P^\dagger$

3. $(\Gamma[B \to \beta][A \to \alpha], v) \vdash (\Gamma[B \to \beta][D \to A], v)$
   where $A \to \alpha \in P^\dagger$ and where there are $D \to A\delta, B \to \beta C\gamma \in P^\dagger$ such that $D \angle^* C$

4. $(\Gamma[B \to \beta][A \to \alpha], v) \vdash (\Gamma[B \to \beta A], v)$
   where $A \to \alpha \in P^\dagger$ and where there is $B \to \beta A\gamma \in P^\dagger$

Example 3.3.1 Consider the grammar from Example 3.2.1. Using Predictive LR, recognition of $a * a$ is realised by the following sequence of configurations:

	$[E' \rightarrow \ ]$	$a * a$
1	$[E' \rightarrow \ ][F \rightarrow a]$	$* a$
2	$[E' \rightarrow \ ][T \rightarrow F]$	$* a$
3	$[E' \rightarrow \ ][T \rightarrow T]$	$* a$
4	$[E' \rightarrow \ ][T \rightarrow T *]$	a
5	$[E' \rightarrow \ ][T \rightarrow T *][F \rightarrow a]$	
6	$[E' \rightarrow \ ][T \rightarrow T * F]$	
7	$[E' \rightarrow \ ][E \rightarrow T]$	
8	$[E' \rightarrow E]$	

If we compare these configurations with those reached by the LC recognizer, then we see that here after Step 3 the stack element $[T \rightarrow T]$ represents both $[T \rightarrow T \bullet * F]$ and $[T \rightarrow T \bullet ** F]$, so that nondeterminism is reduced. In this step still some nondeterminism remains, since Step 3 could also have replaced $[T \rightarrow F]$ by $[E \rightarrow T]$, which represents both $[E \rightarrow T \bullet \uparrow E]$ and $[E \rightarrow T \bullet]$. □

3.3.2 Extended LR parsing

An *extended* context-free grammar has right-hand sides consisting of arbitrary regular expressions over V. This requires an LR parser for an extended grammar (an *ELR* parser) to behave differently from normal LR parsers.

The behaviour of a normal LR parser upon a reduction with some rule $A \rightarrow \alpha$ is very simple: it pops $|\alpha|$ states from the stack, revealing, say, state Q; it then pushes state $goto(Q, A)$. (We identify a state with its corresponding set of items.)

For extended grammars the behaviour upon a reduction cannot be realised in this way since the regular expression of which the rhs is composed may describe strings of various lengths, so that it is unknown how many states need to be popped.

In [PB81] this problem is solved by forcing the parser to decide at each call $goto(Q, X)$ whether

a) X is one more symbol of an item in Q of which some symbols have already been recognized, or whether

b) X is the first symbol of an item which has been introduced in Q by means of the closure function.

In the second case, a state which is a variant of $goto(Q, X)$ is *pushed* on top of state Q as usual. In the first case, however, state Q on top of the stack is *replaced* by a variant of $goto(Q, X)$. This is safe since we will never need to return to Q if after some more steps we succeed in recognizing some rule corresponding with one of the items in Q.

A consequence of the action in the first case above is that upon reduction we need to pop only one state off the stack. A further consequence of this scheme is that deterministic parsing only works if a choice between case **a)** and case **b)** can be uniquely made. Further work in this area is reported in [Lee89], which treats nondeterministic ELR parsing and therefore does not regard it as an obstacle to a working parser if the above-mentioned choice cannot be uniquely made.

We are not concerned with extended context-free grammars in this chapter. However, a very interesting algorithm results from ELR parsing if we restrict its application to ordinary context-free grammars. (We will maintain the name "*extended* LR" to stress the origin of the algorithm.) This results in the new nondeterministic ELR(0) algorithm that we describe below, derived from the formulation of ELR parsing in [Lee89].

First, we define a set of items as

$$I = \{[A \to \alpha \bullet \beta] \mid A \to \alpha\beta \in P^\dagger\}$$

Note that $I^{LC} \subset I$. If we define the *closure* function on subsets of I by

$$closure(Q) = Q \cup \{[A \to \bullet\, \alpha] \mid [B \to \beta \bullet C\gamma] \in Q \wedge C \to^* A\delta\}$$

then the usual *goto* function for normal LR(0) parsing is defined by

$$goto(Q, X) = closure(\{[A \to \alpha X \bullet \beta] \mid [A \to \alpha \bullet X\beta] \in Q\})$$

For ELR parsing however, we need two *goto* functions, $goto_1$ and $goto_2$, one for kernel items (i.e. those in I^{LC}) and one for nonkernel items (the others, i.e. those of the form $[A \to \bullet\, \beta]$, where $A \neq S'$). These are defined by

$$\begin{aligned} goto_1(Q, X) &= closure(\{[A \to \alpha X \bullet \beta] \mid [A \to \alpha \bullet X\beta] \in Q \wedge (\alpha \neq \epsilon \vee A = S')\}) \\ goto_2(Q, X) &= closure(\{[A \to X \bullet \beta] \mid [A \to \bullet\, X\beta] \in Q \wedge A \neq S'\}) \end{aligned}$$

At each shift (where X is some terminal) and each reduce with some rule $A \to \alpha$ (where X is A) we may nondeterministically apply $goto_1$, which corresponds with case **a)**, or $goto_2$, which corresponds with case **b)**. Of course, one or both may not be defined on Q and X, because $goto_i(Q, X)$ may be $\emptyset$, for $i \in \{1, 2\}$.

We shortly present the ELR algorithm analogously to the LC and PLR algorithms. First, we remark that when using $goto_1$ and $goto_2$, each reachable set of items contains only items of the form $[A \to \alpha \bullet \beta]$, for some fixed string α, plus some nonkernel items. We will ignore the nonkernel items since they can be derived from the kernel items by means of the closure function.

This suggests representing each set of items by a new kind of item of the form $[\{A_1, A_2, \ldots, A_n\} \to \alpha]$, which represents all items $[A \to \alpha \bullet \beta]$ for some β and $A \in \{A_1, A_2, \ldots, A_n\}$. The set of all such items is formally given by

$$I^{ELR} = \{[\Delta \to \alpha] \mid \emptyset \subset \Delta \subseteq \{A \mid A \to \alpha\beta \in P^\dagger\} \wedge (\alpha \neq \epsilon \vee \Delta = \{S'\})\}$$

where we use the symbol Δ to range over sets of nonterminals.

Not all items in I^{ELR} may actually be used in the recognition process.

We now have

Algorithm 16 (Extended LR) $A^{ELR} = (T, Alph, Init, \vdash, Fin)$, where $Alph = I^{ELR}$, $Init = [\{S'\} \to\ ]$, $Fin = [\{S'\} \to S]$, and transitions are allowed according to the following clauses.

1. $(\Gamma[\Delta \to \beta], av) \vdash (\Gamma[\Delta \to \beta][\Delta' \to a], v)$
   where $\Delta' = \{A \mid \exists A \to a\alpha, B \to \beta C\gamma \in P^\dagger [B \in \Delta \wedge A \measuredangle^* C]\}$ is non-empty

2. $(\Gamma[\Delta \to \alpha], av) \vdash (\Gamma[\Delta' \to \alpha a], v)$
   where $\Delta' = \{A \in \Delta \mid A \to \alpha a \beta \in P^\dagger\}$ is non-empty

3. $(\Gamma[\Delta \to \beta][\Delta' \to \alpha], v) \vdash (\Gamma[\Delta \to \beta][\Delta'' \to A], v)$
   where there is $A \to \alpha \in P^\dagger$ with $A \in \Delta'$, and $\Delta'' = \{D \mid \exists D \to A\delta, B \to \beta C\gamma \in P^\dagger[B \in \Delta \wedge D \mathrel{\angle^*} C]\}$ is non-empty

4. $(\Gamma[\Delta \to \beta][\Delta' \to \alpha], v) \vdash (\Gamma[\Delta'' \to \beta A], v)$
   where there is $A \to \alpha \in P^\dagger$ with $A \in \Delta'$, and $\Delta'' = \{B \in \Delta \mid B \to \beta A \gamma \in P^\dagger\}$ is non-empty

Note that Clauses 1 and 3 correspond with the application of $goto_2$ (on a terminal or nonterminal, respectively) and that Clauses 2 and 4 correspond with the application of $goto_1$.

Example 3.3.2 Consider again the grammar from Example 3.2.1. Using the ELR algorithm, recognition of $a * a$ is realised by the following sequence of configurations:

	$[\{E'\} \to]$	$a * a$
1	$[\{E'\} \to][\{F\} \to a]$	$* a$
2	$[\{E'\} \to][\{T\} \to F]$	$* a$
3	$[\{E'\} \to][\{T, E\} \to T]$	$* a$
4	$[\{E'\} \to][\{T\} \to T *]$	a
5	$[\{E'\} \to][\{T\} \to T *][\{F\} \to a]$	
6	$[\{E'\} \to][\{T\} \to T * F]$	
7	$[\{E'\} \to][\{T, E\} \to T]$	
8	$[\{E'\} \to E]$	

If we compare these configurations with those reached by the PLR recognizer, then we see that here after Step 3 the stack element $[\{T, E\} \to T]$ represents both $[T \to T \bullet * F]$ and $[T \to T \bullet * * F]$, but also $[E \to T \bullet]$ and $[E \to T \bullet \uparrow E]$, so that nondeterminism is even further reduced. □

A simplified ELR algorithm, which we call the *pseudo* ELR algorithm, results from avoiding reference to Δ in Clauses 1 and 3. In Clause 1 we then have a simplified definition of Δ', viz. $\Delta' = \{A \mid \exists A \to a\alpha, B \to \beta C\gamma \in P^\dagger[A \mathrel{\angle^*} C]\}$, and in the same way we have in Clause 3 the new definition $\Delta'' = \{D \mid \exists D \to A\delta, B \to \beta C\gamma \in P^\dagger[D \mathrel{\angle^*} C]\}$.

Pseudo ELR parsing can be more easily realised than full ELR parsing, but the correct-prefix property can no longer be guaranteed (see also Section 3.3.3). Pseudo ELR parsing is the foundation of a tabular algorithm in [Voi88].

3.3.3 Common-prefix parsing

One of the more complicated aspects of the ELR algorithm is the treatment of the sets of nonterminals in the left-hand sides of items. A drastically simplified version of this algorithm is the basis of a tabular algorithm in [VR90]. Since in [VR90] the algorithm itself is not described but only its tabular realisation,[5] we take the liberty of giving this

[5] An attempt has been made in [Voi86] but this paper does not describe the algorithm in its full generality.

algorithm our own name: *common-prefix (CP) parsing*, since parsing of all rules with a common prefix is done simultaneously.[6]

The simplification of this algorithm with regard to the ELR algorithm consists of omitting the sets of nonterminals in the left-hand sides of items. This results in

$$I^{CP} = \{[\rightarrow \alpha] \mid A \rightarrow \alpha\beta \in P^\dagger\}$$

We now have

Algorithm 17 (Common-prefix) $A^{CP} = (T, Alph, Init, \vdash, Fin)$, where $Alph = I^{CP}$, $Init = [\rightarrow]$, $Fin = [\rightarrow S]$, and transitions are allowed according to the following clauses.

1. $(\Gamma[\rightarrow \beta], av) \vdash (\Gamma[\rightarrow \beta][\rightarrow a], v)$
   where there are $A \rightarrow a\alpha, B \rightarrow \beta C\gamma \in P^\dagger$ such that $A \angle^* C$

2. $(\Gamma[\rightarrow \alpha], av) \vdash (\Gamma[\rightarrow \alpha a], v)$
   where there is $A \rightarrow \alpha a\beta \in P^\dagger$

3. $(\Gamma[\rightarrow \beta][\rightarrow \alpha], v) \vdash (\Gamma[\rightarrow \beta][\rightarrow A], v)$
   where there are $A \rightarrow \alpha, D \rightarrow A\delta, B \rightarrow \beta C\gamma \in P^\dagger$ such that $D \angle^* C$

4. $(\Gamma[\rightarrow \beta][\rightarrow \alpha], v) \vdash (\Gamma[\rightarrow \beta A], v)$
   where there are $A \rightarrow \alpha, B \rightarrow \beta A\gamma \in P^\dagger$

The algorithms presented in the previous sections all share the *correct-prefix property*, which means that in case of incorrect input the parser does not read past the first incorrect character. The simplification which leads to the CP algorithm inevitably causes the correct-prefix property to be lost.

Example 3.3.3 Consider again the grammar from Example 3.2.1. It is clear that $a+a \uparrow a$ is not a correct string according to this grammar. The CP algorithm may go through the following sequence of configurations:

	$[\rightarrow]$	$a + a \uparrow a$
1	$[\rightarrow][\rightarrow a]$	$+ a \uparrow a$
2	$[\rightarrow][\rightarrow F]$	$+ a \uparrow a$
3	$[\rightarrow][\rightarrow T]$	$+ a \uparrow a$
4	$[\rightarrow][\rightarrow E]$	$+ a \uparrow a$
5	$[\rightarrow][\rightarrow E +]$	$a \uparrow a$
6	$[\rightarrow][\rightarrow E +][\rightarrow a]$	$\uparrow a$
7	$[\rightarrow][\rightarrow E +][\rightarrow F]$	$\uparrow a$
8	$[\rightarrow][\rightarrow E +][\rightarrow T]$	$\uparrow a$
9	$[\rightarrow][\rightarrow E +][\rightarrow T \uparrow]$	a
10	$[\rightarrow][\rightarrow E +][\rightarrow T \uparrow][\rightarrow a]$	

[6]The original algorithm in [VR90] applies an optimization concerning unit rules (rules of the form $A \rightarrow B$) following [GHR80]. This is irrelevant to our discussion however.

We see that in Step 9 the first incorrect symbol ↑ is read, but recognition then continues. Eventually, the recognition process is blocked in some unsuccessful configuration, which is guaranteed to happen for any incorrect input[7]. In general however, after reading the first incorrect symbol, the algorithm may perform an unbounded number of steps before it halts. (Imagine what happens for input of the form $a + a \uparrow a + a + a + \ldots + a$.)

Note that in this case, the recognizer might have detected the error at Step 9 by investigating the item $[\rightarrow E +]$. In general however, items unboundedly deep in the stack need to be investigated in order to regain the correct-prefix property. □

3.4 Tabular parsing

Non-deterministic push-down automata can be realised efficiently using parse tables [BL89]. A parse table consists of sets $T_{i,j}$ of items, for $0 \leq i \leq j \leq n$, where $a_1 \ldots a_n$ represents the input. The idea is that an item is only stored in a set $T_{i,j}$ if the item represents recognition of the part of the input $a_{i+1} \ldots a_j$. The table is gradually filled, first with items which can be added irrespective of other items, then with items whose insertion into some set $T_{i,j}$ is justified by other items in other sets in the table.

We will first discuss a tabular form of CP parsing, since this is the most simple parsing technique from Section 3.3. We will then move on to the more difficult but also more interesting ELR technique, and apply an optimization which gives the tabular realisation properties not shared by nondeterministic ELR parsing itself.[8] Tabular PLR parsing is fairly straightforward and will not be discussed in this chapter.

3.4.1 Tabular CP parsing

For CP parsing we can give the following tabular version:

Algorithm 18 (Tabular common-prefix) Sets $T_{i,j}$ of the table are to be subsets of I^{CP}. Start with an empty table. Add $[\rightarrow]$ to $T_{0,0}$. Perform one of the following steps until no more items can be added.

1. Add $[\rightarrow a]$ to $T_{i-1,i}$ for $a = a_i$ and $[\rightarrow \beta] \in T_{j,i-1}$
   where there are $A \rightarrow a\alpha, B \rightarrow \beta C\gamma \in P^\dagger$ such that $A \angle^* C$

2. Add $[\rightarrow \alpha a]$ to $T_{j,i}$ for $a = a_i$ and $[\rightarrow \alpha] \in T_{j,i-1}$
   where there is $A \rightarrow \alpha a \beta \in P^\dagger$

3. Add $[\rightarrow A]$ to $T_{j,i}$ for $[\rightarrow \alpha] \in T_{j,i}$ and $[\rightarrow \beta] \in T_{h,j}$
   where there are $A \rightarrow \alpha, D \rightarrow A\delta, B \rightarrow \beta C\gamma \in P^\dagger$ such that $D \angle^* C$

4. Add $[\rightarrow \beta A]$ to $T_{h,i}$ for $[\rightarrow \alpha] \in T_{j,i}$ and $[\rightarrow \beta] \in T_{h,j}$
   where there are $A \rightarrow \alpha, B \rightarrow \beta A\gamma \in P^\dagger$

[7] unless the grammar is cyclic, in which case the parser may not terminate, both on correct and on incorrect input

[8] This is reminiscent of the *admissibility tests* [Lan88a], which are applicable to *tabular* realisations of logical push-down automata, but not to these automata themselves.

Report recognition of the input if $[\rightarrow S] \in T_{0,n}$.

Example 3.4.1 Consider again the grammar from Example 3.2.1 and the (incorrect) input $a + a \uparrow a$. After execution of the tabular common-prefix algorithm, the table is as given by the following.

	0	1	2	3	4	5
0	$[\rightarrow]$ (0)	$[\rightarrow a]$ (1) $[\rightarrow F]$ (2) $[\rightarrow T]$ (3) $[\rightarrow E]$ (4)	$[\rightarrow E +]$ (5)	$[\rightarrow E + T]$ $[\rightarrow E]$	$\emptyset$	$\emptyset$
1			$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2				$[\rightarrow a]$ (6) $[\rightarrow F]$ (7) $[\rightarrow T]$ (8)	$[\rightarrow T \uparrow]$ (9)	$[\rightarrow T \uparrow E]$
3					$\emptyset$	$\emptyset$
4						$[\rightarrow a]$ (10) $[\rightarrow F]$ $[\rightarrow T]$ $[\rightarrow E]$

The items which correspond with those from Example 3.3.3 are labelled with $(0), (1), \ldots$ These labels also indicate the order in which these items are added to the table. □

Tabular CP parsing is related to a variant of CKY parsing with TD filtering in [Lee89]. A form of tabular CP parsing without top-down filtering (i.e. without the checks concerning the left-corner relation $\angle^*$) is the main algorithm in [VR90].

Without the use of top-down filtering, the references to $[\rightarrow \beta]$ in Clauses 1 and 3 are clearly not of much use any more.[9] When we also remove the use of these items, then we obtain the following new versions of these clauses:

1. Add $[\rightarrow a]$ to $T_{i-1,i}$ for $a = a_i$
   where there is $A \rightarrow a\alpha \in P^\dagger$

3. Add $[\rightarrow A]$ to $T_{j,i}$ for $[\rightarrow \alpha] \in T_{j,i}$
   where there are $A \rightarrow \alpha, D \rightarrow A\delta \in P^\dagger$

In the resulting algorithm, no set $T_{i,j}$ depends on any set $T_{g,h}$ with $g < i$. In [SL92] this fact is used to construct a parallel parser with n processors $P_0, \ldots, P_{n-1}$, with each P_i processing the sets $T_{i,j}$ for all $j > i$. The flow of data is strictly from right to left, i.e. items computed by P_i are only passed on to $P_0, \ldots, P_{i-1}$.

[9] This demonstrates that the omission of TD filtering causes a weak form of bidirectionality for tabular realizations (Section 1.2.2.5).

3.4.2 Tabular ELR parsing

The tabular form of ELR parsing allows an optimization which constitutes an interesting example of how a tabular algorithm can have a property not shared by its nondeterministic origin.

First note that we can compute the columns of a parse table strictly from left to right, that is, for fixed i we can compute all sets $T_{j,i}$ before we compute the sets $T_{j,i+1}$.

If we formulate a tabular ELR algorithm in a naive way analogously to Algorithm 18, as is done in [Lee89], then for example the first clause is given by:

1. Add $[\Delta' \to a]$ to $T_{i-1,i}$ for $a = a_i$ and $[\Delta \to \beta] \in T_{j,i-1}$
   where $\Delta' = \{A \mid \exists A \to a\alpha, B \to \beta C\gamma \in P^\dagger[B \in \Delta \wedge A \angle^* C]\}$ is non-empty

However, for certain i there may be many $[\Delta \to \beta] \in T_{j,i-1}$, for some j, and each may give rise to a different Δ' which is non-empty. In this way, Clause 1 may add several items $[\Delta' \to a]$ to $T_{i-1,i}$, some possibly with overlapping sets Δ'. Since items represent computation of subderivations, the algorithm may therefore compute the same subderivation several times.

We propose an optimization which makes use of the fact that all possible items $[\Delta \to \beta] \in T_{j,i-1}$ are already present when we compute items in $T_{i-1,i}$: we compute one single item $[\Delta' \to a]$, where Δ' is a large set computed using all $[\Delta \to \beta] \in T_{j,i-1}$, for any j. A similar optimization can be made for the third clause.

We now have:

Algorithm 19 (Tabular extended LR) Sets $T_{i,j}$ of the table are to be subsets of I^{ELR}. Start with an empty table. Add $[\{S'\} \to]$ to $T_{0,0}$. For $i = 1, \ldots, n$, in this order, perform one of the following steps until no more items can be added.

1. Add $[\Delta' \to a]$ to $T_{i-1,i}$ for $a = a_i$
   where $\Delta' = \{A \mid \exists j \exists [\Delta \to \beta] \in T_{j,i-1} \exists A \to a\alpha, B \to \beta C\gamma \in P^\dagger[B \in \Delta \wedge A \angle^* C]\}$ is non-empty

2. Add $[\Delta' \to \alpha a]$ to $T_{j,i}$ for $a = a_i$ and $[\Delta \to \alpha] \in T_{j,i-1}$
   where $\Delta' = \{A \in \Delta \mid A \to \alpha a \beta \in P^\dagger\}$ is non-empty

3. Add $[\Delta'' \to A]$ to $T_{j,i}$ for $[\Delta' \to \alpha] \in T_{j,i}$
   where there is $A \to \alpha \in P^\dagger$ with $A \in \Delta'$, and $\Delta'' = \{D \mid \exists h \exists [\Delta \to \beta] \in T_{h,j} \exists D \to A\delta, B \to \beta C\gamma \in P^\dagger[B \in \Delta \wedge D \angle^* C]\}$ is non-empty

4. Add $[\Delta'' \to \beta A]$ to $T_{h,i}$for $[\Delta' \to \alpha] \in T_{j,i}$ and $[\Delta \to \beta] \in T_{h,j}$
   where there is $A \to \alpha \in P^\dagger$ with $A \in \Delta'$, and $\Delta'' = \{B \in \Delta \mid B \to \beta A\gamma \in P^\dagger\}$ is non-empty

Report recognition of the input if $[\{S'\} \to S] \in T_{0,n}$.

Informally, the top-down filtering in the first and third clauses is realised by investigating all left corners D of nonterminals C (i.e. $D \angle^* C$) which are expected from a certain input position. For input position i these nonterminals D are given by

$$S_i = \{D \mid \exists j \exists [\Delta \to \beta] \in T_{j,i} \exists B \to \beta C\gamma \in P^\dagger[B \in \Delta \wedge D \angle^* C]\}$$

Provided each set S_i is computed just after completion of the i-th column of the table, the first and third clauses can be simplified to:

1. Add $[\Delta' \rightarrow a]$ to $T_{i-1,i}$ for $a = a_i$
   where $\Delta' = \{A \mid A \rightarrow a\alpha \in P^\dagger\} \cap S_{i-1}$ is non-empty

3. Add $[\Delta'' \rightarrow A]$ to $T_{j,i}$ for $[\Delta' \rightarrow \alpha] \in T_{j,i}$
   where there is $A \rightarrow \alpha \in P^\dagger$ with $A \in \Delta'$, and $\Delta'' = \{D \mid D \rightarrow A\delta \in P^\dagger\} \cap S_j$ is non-empty

which may lead to more practical implementations.

Note that we may have that the tabular ELR algorithm manipulates items of the form $[\Delta \rightarrow \alpha]$ which would not occur in any search path of the nondeterministic ELR algorithm, because in general such a Δ is the union of many sets Δ' of items $[\Delta' \rightarrow \alpha]$ which would be manipulated at the same input position by the nondeterministic algorithm in *different* search paths.

With minor differences, the above tabular ELR algorithm is described in [VR90]. A tabular version of *pseudo* ELR parsing is presented in [Voi88].

3.4.3 Finding an optimal tabular algorithm

In [Sch91] Schabes derives the LC algorithm from LR parsing similar to the way that ELR parsing can be derived from LR parsing (Section 3.3.2). The LC algorithm is obtained by not only splitting up the goto function into $goto_1$ and $goto_2$ but also splitting up $goto_2$ even further, so that it nondeterministically yields the closure of one single kernel item. (This idea was described earlier in [Lee89], and more recently in [OLS93].)

Schabes then argues that the LC algorithm can be determinized (i.e. made more deterministic) by manipulating the goto functions. One application of this idea is to take a fixed grammar and choose different goto functions for different parts of the grammar, in order to tune the parser to the grammar.

In this section we discuss a different application of this idea: we consider various goto functions which are *global*, i.e. which are the same for all parts of a grammar. One example is ELR parsing, as its $goto_2$ function can be seen as a determinized version of the $goto_2$ function of LC parsing. In a similar way we obtain PLR parsing. Traditional LR parsing is obtained by taking the full determinization, i.e. by taking the normal goto function which is not split up.[10]

We conclude that we have a family consisting of LC, PLR, ELR, and LR parsing, which are increasingly deterministic. In general, the more deterministic an algorithm is, the more parser states it requires. For example, the LC algorithm requires a number of states (the items in I^{LC}) which is linear in the size of the grammar. By contrast, the LR algorithm requires a number of states (the sets of items) which is *exponential* in the size of the grammar [Joh91].

The differences in the number of states complicate the choice of a tabular algorithm as the one giving optimal behaviour for all grammars. If a grammar is very simple, then a sophisticated algorithm such as LR may allow completely deterministic parsing, which requires a linear number of entries to be added to the parse table, measured in the size of the grammar.

[10] Schabes more or less also argues that LC itself can be obtained by determinizing TD parsing. (In lieu of TD parsing he mentions Earley's algorithm, which is its tabular realisation.)

If, on the other hand, the grammar is very ambiguous such that even LR parsing is very nondeterministic, then the tabular realisation may at worst add each state to each set $T_{i,j}$, so that the more states there are, the more work the parser needs to do. This favours simple algorithms such as LC over more sophisticated ones such as LR. Furthermore, if more than one state represents the same subderivation, then computation of that subderivation may be done more than once, which leads to parse forests (compact representations of collections of parse trees) which are not optimally dense [BL89, Rek92] (see also Chapter 2).

Schabes proposes to tune a parser to a grammar, or in other words, to use a *combination* of parsing techniques in order to find an optimal parser for a certain grammar.[11] This idea has until now not been realised. However, when we try to find a single parsing algorithm which performs well for all grammars, then the tabular ELR algorithm from Section 3.4.2 may be a serious candidate, for the following reasons:

- For all i, j, and α at most one item of the form $[\Delta \rightarrow \alpha]$ is added to $T_{i,j}$. Therefore, identical subderivations are not computed more than once. (Note that this is a consequence of the optimization of top-down filtering at the beginning of Section 3.4.2.) Note that this also holds for the tabular CP algorithm.

- ELR parsing guarantees the correct-prefix property, contrary to the CP algorithm. This prevents computation of all subderivations which are useless with regard to the already processed input.

- ELR parsing is more deterministic than LC and PLR parsing, because it allows processing of all common prefixes to be shared. It is hard to imagine a practical parsing technique more deterministic than ELR parsing which also satisfies the previous two properties (see also Section 3.7).

3.5 Data structures

The most straightforward data structure for handling items in I^{CP} is a trie. The vertices of the trie represent the items in I^{CP}. From a vertex representing $[\rightarrow \alpha]$ to a vertex representing $[\rightarrow \alpha X]$ there is an edge labelled X. In order to be able to detect when a complete rhs has been recognized, each vertex representing $[\rightarrow \alpha]$ should have a label $COMPLETE(\alpha)$, which is a set of all nonterminals A such that $A \rightarrow \alpha \in P^{\dagger}$.

For the purpose of top-down filtering we also need to label each vertex representing $[\rightarrow \alpha]$ with $LHS(\alpha)$, which is a set of all A such that $A \rightarrow \alpha\beta \in P^{\dagger}$, for some β. Note that $COMPLETE(\alpha) \subseteq LHS(\alpha)$, for all items $[\rightarrow \alpha]$. Figure 3.1 gives an example.

The data structure used in [SL92] for CP parsing without top-down filtering is inspired by LR tables. First we take an initial set of items to be the set of all nonkernel items, and then we generate new sets of items using a version of the goto function which is stripped of the application of the closure function. Each set of items then corresponds with one LR state, and a table representing the goto function and a table containing reduce entries are constructed, analogously to those normally used for LR(0) parsing. It is easy to see that this structure is equivalent to a trie.

[11] This is reminiscent of the idea of "optimal cover" from [Lee89].

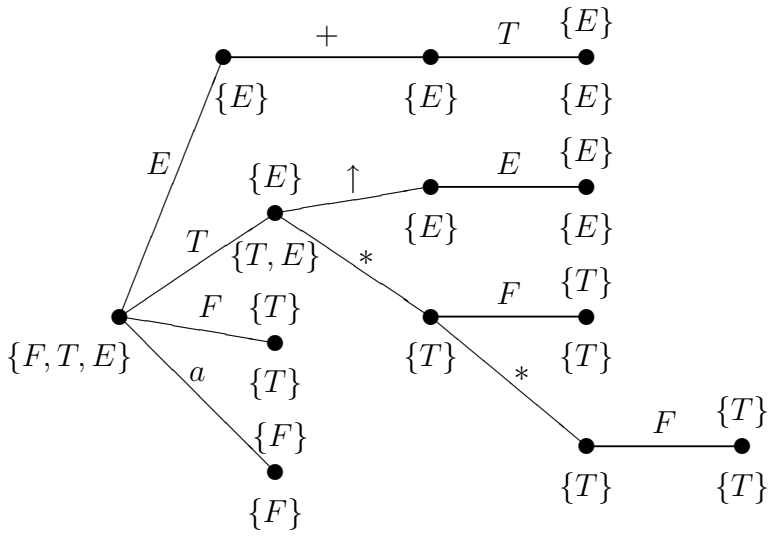


Figure 3.1: The items of I^{CP} in a trie, according to the grammar from Example 3.2.1. The sets $LHS(\alpha)$ are given beneath the vertices, the non-empty $COMPLETE(\alpha)$ are given above the vertices.

Essentially the same ideas may be used for the representation of items in I^{PLR} and I^{ELR}, except that we should then partition the items into a *collection* of tries. For PLR parsing, each trie represents the items $[A \to \alpha]$ for some fixed A.

For ELR parsing, the root of each trie represents an item $[\Delta \to X]$ such that for some other item $[\Delta' \to \beta]$ we have $\Delta = \{A \mid \exists A \to X\alpha, B \to \beta C\gamma \in P^\dagger [B \in \Delta' \wedge A \mathrel{\angle^*} C]\}$. There is an edge labelled X from a vertex representing $[\Delta \to \alpha]$ to a vertex representing $[\Delta' \to \alpha X]$ if $\Delta' = \{A \in \Delta \mid A \to \alpha X\beta \in P^\dagger\}$. Note that the tries for I^{ELR} may have subtries in common.

For tabular ELR parsing we may need more tries. The root of each trie represents an item $[\Delta \to X]$ such that for some S_i which may be calculated during parsing

$$\begin{aligned} \Delta &= \{A \mid A \to X\alpha \in P^\dagger\} \cap S_i \\ &= \bigcup_{[\Delta' \to \beta] \in T_{j,i}} \{A \mid \exists A \to X\alpha, B \to \beta C\gamma \in P^\dagger [B \in \Delta' \wedge A \mathrel{\angle^*} C]\} \end{aligned}$$

Since for all X the number of subsets of $\{A \mid A \to X\alpha \in P^\dagger\}$ is generally small, the calculation of $\{A \mid \exists A \to X\alpha, B \to \beta C\gamma \in P^\dagger [B \in \Delta' \wedge A \mathrel{\angle^*} C]\}$ for X and $[\Delta' \to \beta]$ may be computed statically. Also the unions of these sets for different $[\Delta' \to \beta]$ may be computed statically. These results may be stored into tables which can be used during parsing. In addition, the outcome of the complete calculation may be memoized for X and i.

This implementation of tabular ELR parsing is in contrast to the algorithm in [VR90], which makes use of a single trie for I^{CP}, combined with (elementwise) calculation of the S_i and of the sets of nonterminals in the left-hand sides of items, which obviously does not give optimal time complexities.

3.6 Epsilon rules

There are two ways in which epsilon rules may form an obstacle to bottom-up parsing. The first is non-termination for simple realisations of nondeterminism (such as backtrack parsing) caused by hidden left recursion.

Secondly, we explain informally how for tabular parsing, epsilon rules may interfere with optimization of top-down filtering. In Section 3.4.2 we made use of the fact that before we calculate the sets $T_{j,i}$ for some fixed i, we have already calculated the sets $T_{j,i-1}$. We can then prepare for top-down filtering by combining all $[\Delta \to \beta] \in T_{j,i-1}$ in S_{i-1} as explained.

However, suppose that we allow epsilon rules and that for some $[\Delta' \to \beta'] \in T_{j',i-1}$, for some j', we have a rule $A \to \beta' BC$, where $A \in \Delta'$, and $B \to^* \epsilon$. Then for finding a derivation $B \to^* \epsilon$ we need top-down filtering, for which we have combined in S_{i-1} all $[\Delta \to \beta] \in T_{j,i-1}$, for any j. After we have found this derivation however, we find that some $[\Delta'' \to \beta' B]$ should be in $T_{j',i-1}$. But this is in conflict with our assumption that we had already found all $[\Delta \to \beta] \in T_{j,i-1}$, for any j.

We propose treatment of epsilon rules analogously to Chapters 2 and 4, which amounts to merging epsilon-rule elimination with the parser construction, without actually transforming the grammar. We omit details.

3.7 Beyond ELR parsing

We have conjectured in Section 3.4.3 that there is no parsing algorithm more deterministic than ELR parsing which allows a *practical* tabular algorithm with the property that subderivations may never be computed more than once. Below we give an informal explanation of why we feel this conjecture to be true.

For tabular ELR parsing, the above property was ensured in Section 3.4.2 by applying an optimization of top-down filtering. The result of this optimization is that, for example, Clause 1 adds a single item $[\Delta \to \alpha]$ to $T_{i-1,i}$ where a naive tabular ELR parser would add several items $[\Delta' \to \alpha]$ to $T_{i-1,i}$ for different Δ', such that Δ is the union of all such Δ'. A similar fact holds for Clause 3.

Let us now consider how this idea can be translated into the realm of LR parsing. For LR parsing, the stack symbols (the states) represent *sets* of items from I. If two states representing say J and H are both added to some set $T_{i,j}$ of the table of a tabular LR parser, then we have that a subderivation is being computed twice if some kernel item belongs to both J and H. We can avoid this by adding a single state to $T_{i,j}$ representing $J \cup H$, by analogy of the above-mentioned optimization for tabular ELR parsing.

However, the number of sets of items that we will then need may become prohibitively large, even larger than the number of sets we need for traditional LR parsing. (In case we do not compile the operations on sets of items into an LR table, but compute the sets during parsing, then the algorithm actually comes down to a tabular LC algorithm.) Furthermore, merging two states leads to new problems later on when a reduction has to be performed which is only applicable for one of the two original states.

We conclude that there is no practical tabular LR parsing algorithm which has the property that subderivations cannot be computed more than once. The main obstacle is the number of sets of items which would be required to apply the above optimization.

However, mixing LR parsing with simpler kinds of parsing, as suggested by Schabes, may lead to feasible tabular algorithms, since in that case the number of states which are required may be smaller than that for full LR parsing.

How the top-down prediction of rules or nonterminals may be incorporated in LR parsing is discussed in a number of papers, which only take deterministic parsing into account. For example, [Ham74] describes how top-down prediction of nonterminals may be mixed with LR parsing. LR parsing is performed until a certain nonterminal A can be *predicted with certainty*, with which we mean that it is certain that this nonterminal will occur left-most in the derivation under construction. Then a specialised LR automaton is activated for the recognition of A. After this has been completed, normal LR parsing continues.

A similar idea is described in [IF83]. LR parsing is performed until a unique item $[A \rightarrow \alpha \bullet \beta]$ can be predicted with certainty. TD parsing is then activated consecutively for the symbols in β, until TD parsing cannot be done deterministically, and then LR parsing takes over recursively. After recognition of the symbols in β, normal LR parsing continues.

The generalized LC algorithm in [Dem77] makes use of an annotated grammar, where each rhs is divided into two parts, the first being the (generalized) left corner. LR parsing is performed until we reach a state of which the set of items contains some item $[A \rightarrow \alpha \bullet \beta]$ whose left corner is α. It should then be possible to predict this item by certainty (this is a constraint on the grammar), and specialised LR automata are then activated for the consecutive recognition of the symbols in β. After this has been completed, normal LR parsing continues.

For general grammars, in particular for grammars which are very ambiguous, TD prediction with certainty of nonterminals or rules may not be possible very often. We may however generalize the above ideas by allowing nondeterministic prediction of nonterminals or rules, even if these cannot be predicted with certainty.[12] The resulting algorithms may be used as starting-points for tabular algorithms. An example is the head-driven chart parser from [BvN93], which is essentially a tabular realisation of nondeterministic generalized LC parsing without LR states. Also the tabular algorithm in [JM92, JM94] seems to be based on generalized LC parsing.

Algorithms which allow more deterministic parsing than LR parsing does, such as Marcus' algorithm [Lee92b], fall outside the scope of this chapter.

3.8 A two-dimensional system of parsing techniques

In the previous section we mentioned the generalized LC algorithm from [Dem77], which makes use of an annotated grammar. The annotations determine when a rule is to be predicted. A refinement of this idea is described in [Nij80, Chapter 12]: there are two annotations for each rule, the first determines when the lhs of the rule is to be predicted and the second determines when the rhs is to be predicted. By annotating all rules in the

[12] There is a complication with generalizing the algorithm from [Ham74] in this way, since nondeterministic prediction of different nonterminals may lead to different computations of the same subderivation. In other words, different search paths may lead to recognition of the same derivation, which is an objectionable property for a parsing algorithm. These problems do not occur for [Dem77] or [IF83], provided no nonkernel items are predicted.

same way, a parsing technique can be specified, as we explain below. (This idea from [Nij80, Chapter 12], which treats deterministic parsing, will be generalized in a straightforward way to nondeterministic parsing.)

Consider pairs of the form (i, j), where $i, j \in \{0, 1, \infty\}$. The first variable, i, indicates when left-hand sides are predicted, and the second variable, j, indicates when right-hand sides are predicted. The value 0 means "at the beginning of a rhs", 1 means "after having recognized the first member", and ∞ means "at the end of a rhs".

So for example, the pair $(0, 0)$ indicates top-down parsing: both the lhs and the rhs of each rule are predicted before any of the members have been recognized. The pair $(1, 1)$ indicates LC parsing: a rule is predicted after the first member has been recognized. The pair $(1, \infty)$ indicates PLR parsing: although we predict the lhs of a rule after having recognized the first member, the rhs is only predicted after all of its members have been recognized.

In [Nij80] it is initially argued that the pair (∞, ∞) should indicate LR parsing. However, it may be more accurate to have it denote ELR parsing, since LR parsing allows states which each represent a set of kernel items $[A \rightarrow \alpha \bullet \beta]$ where α does not have to be fixed, and this additional power does not fit into the two-dimensional system. In [Nij80, Section 12.4] this problem is identified and *weak PLR* parsing is introduced, which in fact corresponds to ELR parsing.

Two more parsing techniques in the system are explicitly mentioned in [Nij80]: *PLC* parsing[13], indicated by $(0, 1)$, and *LP* parsing, indicated by $(0, \infty)$. It is interesting to note that LP parsing relates to TD parsing as PLR parsing relates to LC parsing: PLR parsing can be seen as LC parsing for a grammar which has been left-factored, and in the same way LP parsing can be seen as TD parsing for a grammar which has been left-factored.

3.9 Conclusions

We have discussed a range of different parsing algorithms, which have their roots in compiler construction, expression parsing, and natural language processing. We have shown how these algorithms can be described in a common framework.

We further discussed tabular realisations of these algorithms, and concluded that we have found an optimal algorithm, which in most cases leads to parse tables containing fewer entries than for other algorithms, but which avoids computing identical subderivations more than once.

[13] A head-driven variant is the TD algorithm in [NS94].

Chapter 4

Increasing the Applicability of LR Parsing

In this chapter we discuss a phenomenon present in some context-free grammars, called *hidden left recursion*. We show that ordinary LR parsing according to hidden left-recursive grammars is not possible and we indicate a range of solutions to this problem. One of these solutions is a new parsing technique, which is a variant of traditional LR parsing. This new parsing technique can be used both with and without lookahead and the nondeterminism can be realized using backtracking or using a graph-structured stack.

4.1 Introduction

The class of LR parsing strategies [SSS90] constitutes one of the strongest and most efficient classes of parsing strategies for context-free grammars. LR parsing is commonly used in compilers as well as in systems for the processing of natural language.

Deterministic LR parsing with lookahead of k symbols is possible for LR(k) grammars. Deterministic parsing according to grammars which are not LR(k) can in some cases be achieved with some disambiguating techniques ([AJU75, Ear75]; important progress in this field has been reported in [Tho94]). However, these techniques are not powerful enough to handle practical grammars for e.g. natural languages.

If we consider LR parsing tables in which an entry may contain multiple actions, then we obtain nondeterministic LR parsing. We will refer to realizations of nondeterministic LR parsing as *generalized* LR parsing. The most straightforward way to obtain generalized LR parsing is by using backtracking [Nil86].

A more efficient kind of generalized LR parsing has been proposed in [Tom86]. The essence of this approach is that multiple parses are processed simultaneously. Where possible, the computation processing two or more parses is shared. This is accomplished by using a *graph-structured stack*. (See also Chapter 2.)

Although generalized LR parsing can handle a large class of grammars, there is one phenomenon which it cannot deal with, viz. *hidden left recursion*. Hidden left recursion, defined in Section 1.1, occurs very often in grammars for natural languages.

A solution for handling hidden left-recursive grammars using Tomita's algorithm was proposed in [NF91]. In that paper, the ordinary acyclic graph-structured stack is generalized to allow cycles. The resulting parsing technique is largely equivalent to a parsing technique which follows from a construction defined earlier in [Lan74], which makes use of a parse matrix. As a consequence, termination of the parsing process is always guaranteed. This means that hidden left-recursive grammars and even cyclic grammars can be handled.

However, cyclic graph-structured stacks may complicate garbage collection and cannot be realized using memo-functions [LAKA92]. Tomita's algorithm furthermore becomes very complicated in the case of augmented context-free grammars (e.g. attribute grammar, affix grammar, definite clause grammar, etc.). In this case, different subparses almost always have different attribute values (or affix values, variable instantiations, etc.) and therefore sharing of the computation of context-free parsing would obstruct the correct computation of these values (see Chapter 6).

In this chapter we discuss an alternative approach to adapting the (generalized) LR parsing technique to hidden left-recursive grammars.

Our approach can be roughly described as follows. Reductions with epsilon rules are no longer performed. Instead, a reduction with some non-epsilon rule does not have to pop all the members in the right-hand side off the stack; only those which do not derive the empty string must be popped, for others it is optional. The definition of the closure function for sets of items is changed accordingly. Our approach requires the inspection of the parse stack upon each reduction in order to avoid incorrect parses.

The structure of this chapter is as follows. In the next section we give an introduction to the problem of LR parsing according to hidden left-recursive grammars. We give two naive ways of solving this problem by first transforming the grammar before constructing the (nondeterministic) LR automaton. (These methods are naive because the transformations lead to larger grammars and therefore to much larger LR automata.) We then show how the first of these transformations can be incorporated into the construction of LR automata, which results in parsers with a fewer number of states. We also outline an approach of adapting the LR technique to cyclic grammars.

In Section 4.3 we prove the correctness of our new parsing technique, called ϵ-LR parsing. Efficient generation of ϵ-LR parsers is discussed in Section 4.4. We conclude in Section 4.5 by giving some results on the comparison between the number of states of various LR and ϵ-LR parsers.

We would like to stress beforehand that grammars with nontrivial hidden left recursion can never be handled using deterministic LR parsing (Section 4.2.5), so that most of the discussion in this chapter is not applicable to deterministic LR parsing. We therefore, contrary to custom, use the term "LR parsing" for *generalized* LR parsing, which can at will be realized using backtracking (possibly in combination with memo-functions) or using *acyclic* graph-structured stacks. Wherever we deal with *deterministic* LR parsing, this will be indicated explicitly.

Most of the notation and terminology in this chapter is explained in Section 1.1.

We call a nonterminal A *reachable* if $S \rightarrow^* \alpha A \beta$ for some α and β. We call a grammar *reduced* if every nonterminal is reachable and derives some terminal string. Where we give a transformation between context-free grammars, we tacitly assume that the input grammars are reduced and for these grammars the output grammars are guaranteed also to be reduced.

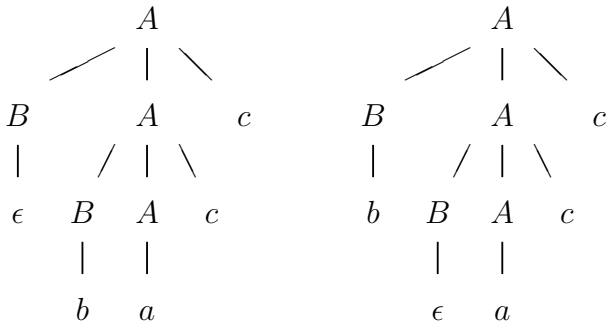


Figure 4.1: Two parse trees with the same yield, showing ambiguity of G_1.

4.2 Hidden left recursion and LR parsing

The simplest nontrivial case of hidden left recursion is the grammar G_1 given by the following rules.

$$\begin{array}{lcl} A & \rightarrow & BAc \\ A & \rightarrow & a \\ B & \rightarrow & b \\ B & \rightarrow & \epsilon \end{array}$$

In this grammar, nonterminal A is left-recursive. This fact is hidden by the presence of a nullable nonterminal B in the rule $A \rightarrow BAc$. Note that this grammar is ambiguous, as illustrated in Figure 4.1. This is typically so in the case where the one or more nullable nonterminals which hide the left recursion are not all predicates.

4.2.1 Generalized LR parsing and hidden left recursion

We now discuss informally how (generalized) LR parsing fails to terminate for the above grammar. We assume that the reader is familiar with the construction of (nondeterministic) LR(0) automata. Our terminology is taken from [ASU86].

A pictorial representation of the LR(0) parsing table for G_1 is given in Figure 4.2. LR parsing of any input w may result in many sequences of parsing steps, one of which is illustrated by the following sequence of configurations.

Stack contents	Input	Action
Q_0	w	reduce($B \rightarrow \epsilon$)
$Q_0\ B\ Q_1$	w	reduce($B \rightarrow \epsilon$)
$Q_0\ B\ Q_1\ B\ Q_1$	w	reduce($B \rightarrow \epsilon$)
$Q_0\ B\ Q_1\ B\ Q_1\ B\ Q_1$	w	reduce($B \rightarrow \epsilon$)
$\vdots$	$\vdots$	$\vdots$

The sequence of parsing steps illustrated above does not terminate. We can find a non-terminating sequence of parsing steps for the LR(0) automaton for every hidden left-recursive grammar. In fact, this is even so for the LR(k), LALR(k), and SLR(k) automata,

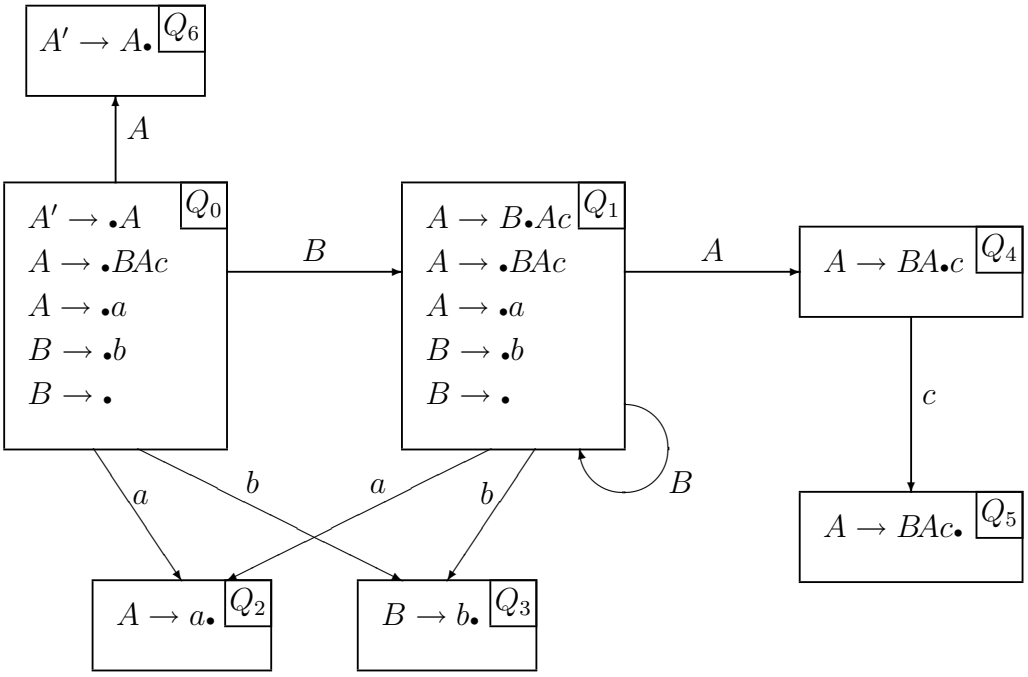


Figure 4.2: The LR(0) automaton for G_1.

for any k. Hidden left recursion has been identified in [SST88] as one of two sources, together with cyclicity, of the looping of LR parsers.

Various other parsing techniques, such as left-corner parsing (Chapter 2) and cancellation parsing (Chapter 5), also suffer from this deficiency.

4.2.2 Eliminating epsilon rules

We first discuss a method to allow LR parsing for hidden left-recursive grammars by simply performing a source to source transformation on grammars to eliminate the rules of which the right-hand sides only derive the empty string. To preserve the language, for each rule containing an occurrence of a nullable nonterminal a copy must be added without that occurrence. Following [AU72], this transformation, called ϵ-*elim*, is described below. The input grammar is called G.

1. Let G_0 be G.

2. Remove from G_0 all rules defining predicates in G and remove all occurrences of these predicates from the rules in G_0.

3. Replace every rule of the form $A \to \alpha_0 B_1 \alpha_1 B_2 \ldots B_m \alpha_m$ in G_0, $m \geq 0$, where the members which are nullable in G are exactly $B_1, \ldots, B_m$, by the set of rules of the form $A \to \alpha_0 \beta_1 \alpha_1 \beta_2 \ldots \beta_m \alpha_m$, where β_i is either B_i or ϵ and $\alpha_0 \beta_1 \alpha_1 \beta_2 \ldots \beta_m \alpha_m \neq \epsilon$. Note that this set of rules is empty if $m = 0$ and $\alpha_0 = \epsilon$, in which case the original rule is just eliminated from G_0.

4. If S is nullable in G, then add the rules $S^\dagger \to S$ and $S^\dagger \to \epsilon$ to G_0 and make $S^\dagger$ the new start symbol of G_0. (In the pathological case that S is a predicate in G, $S^\dagger \to S$ should of course not be added to G_0.)

5. Let ϵ-*elim*(G) be G_0.

Note that for every rule $A \to \alpha$ such that α contains k occurrences of nullable non-predicates, the transformed grammar may contain 2^k rules.

In this chapter, an expression of the form $[B]$ in a rhs indicates that the member B has been eliminated by the transformation. It is for reasons of clarity that we write this expression instead of just leaving B out.

An item of the form $A \to [\alpha_0]X_1[\alpha_1]\ldots[\alpha_{i-1}]\bullet X_i \ldots X_m[\alpha_m]$ is said to be *derived* from the *basic* item $A \to \alpha_0 X_1 \alpha_1 \ldots \alpha_{i-1}\bullet X_i \ldots X_m \alpha_m$.[1] According to the convention mentioned above, $A \to \alpha_0 X_1 \alpha_1 \ldots X_m \alpha_m$ is a rule in G, and $A \to X_1 \ldots X_m$ is a rule in ϵ-*elim*(G). The item of the form $S^\dagger \to \bullet$ which may be introduced by ϵ-*elim* will be regarded as the derived item $S^\dagger \to [S]\bullet$.

Example 4.2.1 Let the grammar G_2 be defined by the rules

$$\begin{array}{lcl} A & \to & BCD \\ B & \to & \epsilon \\ B & \to & b \\ C & \to & \epsilon \\ D & \to & \epsilon \\ D & \to & d \end{array}$$

Step 2 of ϵ-*elim* removes the rule $C \to \epsilon$ defining the only predicate C. Also the occurrence of C in $A \to BCD$ is removed, i.e. this rule is replaced by $A \to B[C]D$.

Step 3 removes all rules with an empty rhs, viz. $B \to \epsilon$ and $D \to \epsilon$, and replaces $A \to B[C]D$ by the set of all rules which result from either eliminating or retaining the nullable members, viz. B and D (C is not a member anymore!), such that the rhs of the resulting rule is not empty. This yields the set of rules

$$\begin{array}{lcl} A & \to & B[C]D \\ A & \to & B[CD] \\ A & \to & [BC]D \end{array}$$

Step 4 adds the rules $A^\dagger \to A$ and $A^\dagger \to \epsilon$. The new start symbol is $A^\dagger$.

We have now obtained ϵ-*elim*(G_2), which is defined by

$$\begin{array}{lcl} A^\dagger & \to & A \\ A^\dagger & \to & \epsilon \\ A & \to & B[C]D \\ A & \to & B[CD] \\ A & \to & [BC]D \\ B & \to & b \\ D & \to & d \end{array}$$

□

[1] We avoid writing dots in dotted items immediately to the left of eliminated members.

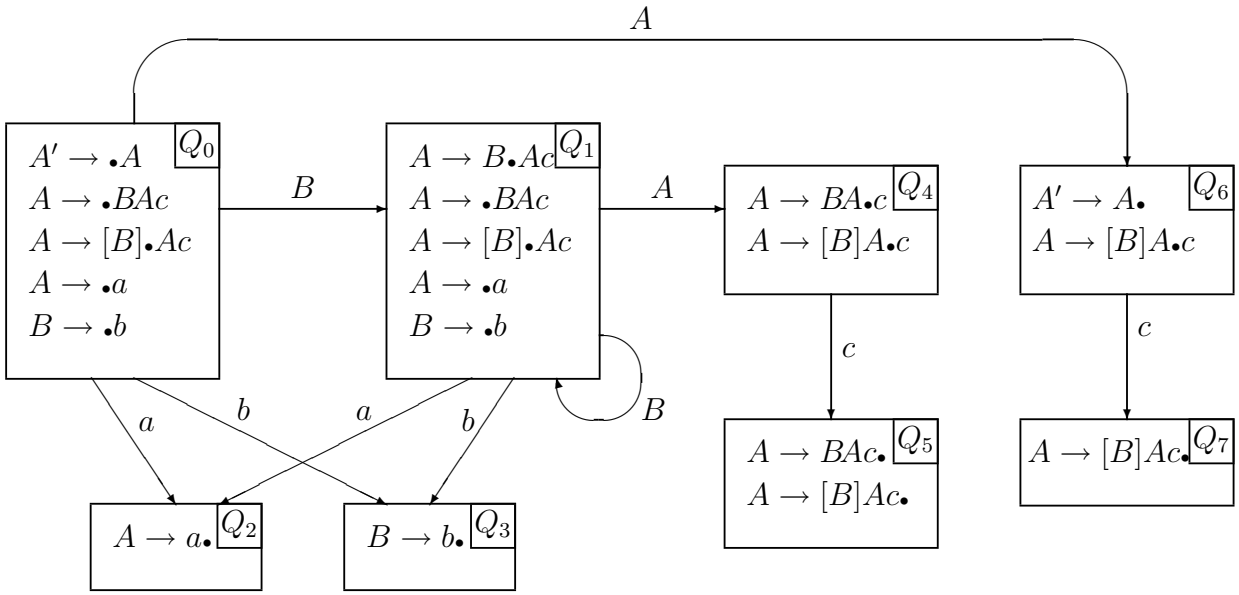


Figure 4.3: The LR(0) automaton for ϵ-*elim*(G_1).

Note that in the case that ϵ-*elim* introduces a new start symbol $S^\dagger$, there is no need to augment the grammar (i.e. add the rule $S' \to S^\dagger$ and make S' the new start symbol) for the purpose of constructing the LR automaton. Augmentation is in this case superfluous because the start symbol $S^\dagger$ is not recursive.

In the case of G_1, the transformation yields the following grammar.

$$\begin{array}{lcl} A & \to & BAc \\ A & \to & [B]Ac \\ A & \to & a \\ B & \to & b \end{array}$$

The LR(0) table for this grammar is represented in Figure 4.3.

Together with the growing number of rules, the above transformation may also give rise to a growing number of states in the LR(0) automaton. In the above case, the number of states increases from 7 to 8, as indicated by Figures 4.2 and 4.3. As G_1 is only a trivial grammar, we may expect that the increase of the number of states for practical grammars is much larger. Tangible results are discussed in Section 4.5.

4.2.3 A new parsing algorithm

To reduce the number of states needed for an LR automaton for ϵ-*elim*(G), we incorporate the transformation in the closure function. This requires changing the behaviour of the LR automaton upon reduction.

This approach can in a different way be explained as follows. Items derived from the same basic item by ϵ-*elim* are considered the same. For instance, the items $A \to BAc\bullet$ and

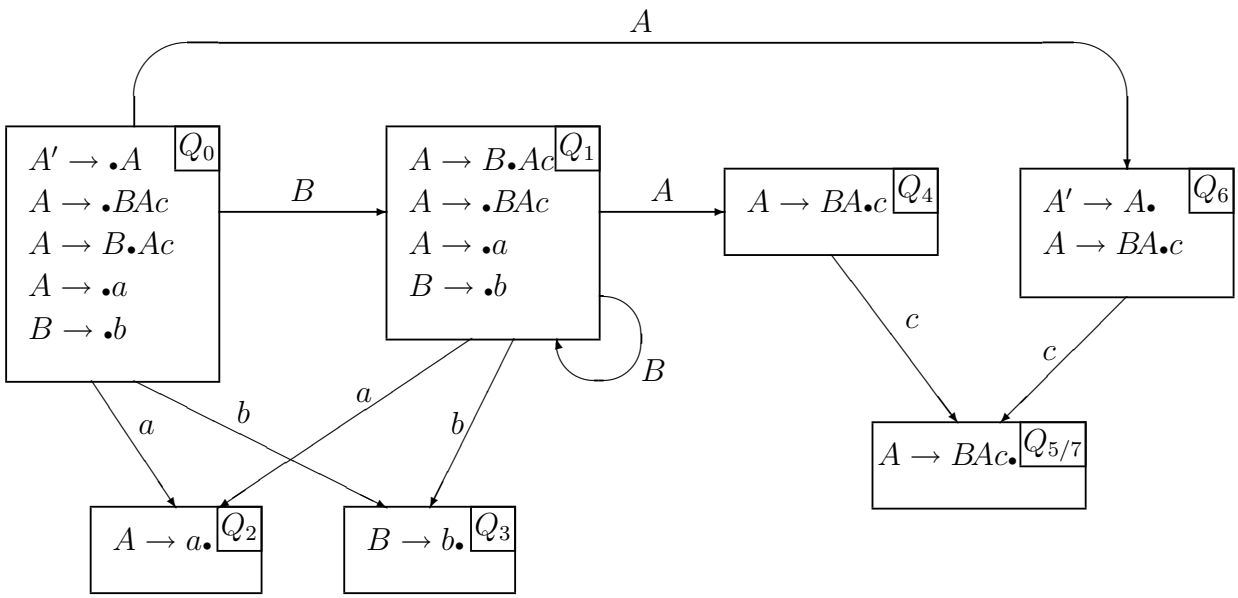


Figure 4.4: The optimised LR(0) automaton for ϵ-$elim(G_1)$ with merged states.

$A \to [B]Ac\bullet$ in Figure 4.3 are considered the same because they are derived from the same basic item $A \to BAc\bullet$.

All items are now represented by the basic item from which they are derived. For instance, both items in Q_5 in Figure 4.3 are henceforth represented by the single basic item $A \to BAc\bullet$. The item $A \to [B]Ac\bullet$ in state Q_7 is now represented by $A \to BAc\bullet$.

As a result, some pairs of states now consist of identical sets of items and may therefore be merged. For the example in Figure 4.3, the new collection of states is given in Figure 4.4. It can be seen that states Q_5 and Q_7 are merged into state $Q_{5/7}$.

In the resulting LR table, it is no longer indicated which derived items are actually represented. Correspondingly, the behaviour of the new automaton is such that upon reduction all possibilities of derived items are to be tried nondeterministically.

For instance, consider the parsing of *bacc* using the LR(0) automaton in Figure 4.4. The first sequence of parsing steps is without complications:

Stack contents	Input	Action
Q_0	*bacc*	shift
$Q_0\ b\ Q_3$	*acc*	reduce($B \to b$)
$Q_0\ B\ Q_1$	*acc*	shift
$Q_0\ B\ Q_1\ a\ Q_2$	*cc*	reduce($A \to a$)
$Q_0\ B\ Q_1\ A\ Q_4$	*cc*	shift
$Q_0\ B\ Q_1\ A\ Q_4\ c\ Q_{5/7}$	*c*	reduce(?)

Now there are two ways to perform a reduction with the item $A \to BAc\bullet$. One way is to assume that B has been eliminated from this rule. In other words, we are dealing with the

derived item $A \to [B]Ac\bullet$. In this case we remove two states and grammar symbols from the stack. The sequence of configurations from here on now begins with

$Q_0\ B\ Q_1\ A\ Q_4\ c\ Q_{5/7}$	c	reduce($A \to [B]Ac$)
$Q_0\ B\ Q_1\ A\ Q_4$	c	
$\vdots$	$\vdots$	$\vdots$

The other way to perform reduction is by taking off the stack all the members in the rule $A \to BAc$ and the same number of states, and we obtain

$Q_0\ B\ Q_1\ A\ Q_4\ c\ Q_{5/7}$	c	reduce($A \to BAc$)
$Q_0\ A\ Q_6$	c	shift
$Q_0\ A\ Q_6\ c\ Q_{5/7}$		reduce(?)

We are now at an interesting configuration. We have again the choice between reducing with $A \to [B]Ac$ or with the unaffected rule $A \to BAc$. However, it can be established that reduction with $A \to BAc$ is not possible, because there is no B on the stack to be popped.

At this point, the main difference between traditional LR parsing and the new parsing technique we are developing becomes clear. Whereas for traditional LR parsing, the grammar symbols on the stack have no other purpose except an explanatory one, for our new parsing technique, the investigation of the grammar symbols on the stack is essential for guiding correct parsing steps.

In general, what happens upon reduction is this. Suppose the state on top of the stack contains an item of the form $A \to \alpha\bullet$, then reduction with this item is performed in the following steps.

1. The parser nondeterministically looks for some sequence of grammar symbols $X_1, \ldots, X_m$ such that there are $\alpha_0, \ldots, \alpha_m$ with

   - $\alpha = \alpha_0 X_1 \alpha_1 \ldots X_m \alpha_m$
   - $\alpha_0 \to^* \epsilon \wedge \ldots \wedge \alpha_m \to^* \epsilon$
   - The top-most m grammar symbols on the stack are $X_1, \ldots, X_m$ in that order, i.e. X_1 is deepest in the stack and X_m is on top of the stack.
   - $m = 0 \Rightarrow A = S'$

   In words, α is divided into a part which is on the stack and a part which consists only of nullable nonterminals. The part on the stack should not be empty with the exception of the case where $A \to \alpha$ is the rule $S' \to S$.

2. The top-most m symbols and states are popped off the stack.

3. Suppose that the state on top of the stack is Q, then

   - if $A = S'$, then the input is accepted, provided Q is the initial state and the end of the input has been reached; and
   - if $A \neq S'$, then A and subsequently *goto* (Q, A) are pushed onto the stack, provided *goto* (Q, A) is defined (otherwise this step fails).

The way the reduction is handled above corresponds with the reduction with the rule $A \to [\alpha_0]X_1[\alpha_1]\ldots X_m[\alpha_m]$ in the original LR(0) parser for ϵ-*elim*(G).

Incorporating the transformation ϵ-*elim* into the construction of the LR table can be seen as a restatement of the usual closure function, as follows.

$$\begin{aligned} \mathit{closure}\ (q) \quad = \quad & \{B \to \delta\bullet\theta \mid A \to \alpha\bullet\beta \in q \wedge \beta \to^* B\gamma \wedge B \to \delta\theta \wedge \\ & \qquad\qquad \exists v[v \neq \epsilon \wedge \delta\theta \to^* v] \wedge \delta \to^* \epsilon\} \\ & \cup \\ & \{A \to \alpha\delta\bullet\beta \mid A \to \alpha\bullet\delta\beta \in q \wedge \delta \to^* \epsilon\} \end{aligned}$$

Note that the expression $\beta \to^* B\gamma$ allows nonterminals to be rewritten to the empty string. Also note that $\exists v[v \neq \epsilon \wedge \delta\theta \to^* v]$ excludes rules of which the rhs can only derive ϵ. Efficient calculation of the closure function is investigated in Section 4.4.1.

[Lee92a] proposes similar changes to some functions in the recursive ascent Earley parser in order to allow hidden left recursion. Similar changes were made in [GHR80, Lei90] in order to improve the efficiency of Earley parsing. A parsing technique very similar to ours is suggested in [Lee93, Section 9.4.1]. A similar idea is also suggested in [Sik90], but here details are lacking.

The investigation of the grammar symbols on the stack for the purpose of guiding correct parsing steps is reminiscent of [Pag70], which proposed a general method for the compression of parsing tables by merging states. If the full stack may be investigated upon reduction, then the need for states in the traditional sense is even completely removed, as shown in [FG92].[2]

In Section 4.3 we formalise the new parsing technique, which we call ϵ-LR parsing, and prove its correctness.

4.2.4 Dealing with cyclic grammars

If needed, ϵ-LR parsing can be further refined to handle cyclic grammars.

The starting-point is again a transformation on grammars, called *C-elim*, which eliminates all *unit rules*, i.e. all rules of the form $A \to B$. This transformation consists of the following steps.

1. Let G_0 be G.
2. Replace every non-unit rule $A \to \alpha$ in G_0 by the set of rules of the form $B \to \alpha$ such that $B \xrightarrow{G}{}^* A$ and either $B = S$ or B has an occurrence in the rhs of some non-unit rule.
3. Remove all unit rules from G_0.
4. Let *C-elim*(G) be G_0.

[2] It is interesting to note that various degrees of simplification of the collection of sets of items are possible. For example, one could imagine an approach half-way between our approach and the one in [FG92], according to which items consist only of the parts after the dots of ordinary items. This leads to even more merging of states but requires more effort upon reductions.

Termination of LR parsing according to *C-elim*(ϵ-*elim*(G)) is guaranteed for any G.

If we incorporate *C-elim* into our ϵ-LR parsers, then reduction with $A \rightarrow \alpha$ is performed by the following steps.

1. The parser nondeterministically looks for some sequence of grammar symbols $X_1, \ldots, X_m$ such that there are $\alpha_0, \ldots, \alpha_m$ with
   - $\alpha = \alpha_0 X_1 \alpha_1 \ldots X_m \alpha_m$
   - $\alpha_0 \rightarrow^* \epsilon \wedge \ldots \wedge \alpha_m \rightarrow^* \epsilon$
   - The top-most m grammar symbols on the stack are $X_1, \ldots, X_m$.
   - $m = 0 \Rightarrow A = S'$
   - $m = 1 \Rightarrow (X_1 \in T \vee A = S')$
2. The top-most m symbols and states are popped off the stack.
3. Suppose that the state on top of the stack is Q, then
   - if $A = S'$, then the input is accepted, provided Q is the initial state and the end of the input has been reached; and
   - if $A \neq S'$, then the parser nondeterministically looks for some nonterminal B such that $B \rightarrow^* A$ and *goto* (Q, B) is defined, and then B and subsequently *goto* (Q, B) are pushed onto the stack.

Note that the parser which performs reduction in this way, using the parse tables from the ϵ-LR parser, may go into unnecessary dead alleys of length one. The reason is that the goto function is defined in cases where is it obvious that the next step must be a reduction with a rule with a single non-predicate in its rhs, which is of course blocked by the above formulation of reduction. This may be avoided by reformulating the closure function such that rules containing a single non-predicate in their right-hand sides are left out.

How to avoid reductions with unit rules (*unit reductions*) in the case of deterministic LR parsing has been investigated in a number of papers (e.g. [Hei85]). Our particular technique of avoiding unit reductions is reminiscent of an optimization of Earley's algorithm [GHR80, Lei90].

In the remaining part of this chapter, the term "ϵ-LR parsing" will not include the extra extension to ϵ-LR parsing described in this section.

4.2.5 Applicability of ϵ-LR parsing

In the same way as LR(0) parsing can be refined to SLR(k), LALR(k), and LR(k) parsing ($k > 0$) we can also refine ϵ-LR(0) parsing to ϵ-SLR(k), ϵ-LALR(k), and ϵ-LR(k) parsing. The construction of ϵ-LR tables for these parsing strategies can be adopted from the construction of their LR counterparts in a reasonably straightforward way.

We have shown that ϵ-LR parsing can be used for hidden left-recursive grammars, which cannot be handled using ordinary LR parsing. The variants of ϵ-LR parsing which apply lookahead are useful for making the parsing process more deterministic, i.e. to reduce the number of entries in the parsing table that contain multiple actions.

However, adding lookahead cannot yield completely deterministic parsers in the case of hidden left recursion where at least one of the hiding nonterminals is not a predicate. This is because such a grammar is ambiguous, as discussed earlier. (If all hiding nonterminals are predicates, then we are dealing with a trivial form of hidden left recursion, which can easily be eliminated by eliminating the hiding nonterminals.)

Also in the case of grammars without hidden left recursion, ϵ-LR parsing may have an advantage over ordinary (generalized) LR parsing: the parsing actions corresponding with subtrees of the parse tree which have empty yields are avoided. For these grammars, the application of lookahead may serve to construct deterministic ϵ-LR parsers.

Chapter 2 describes how subtrees which have empty yields can be attached to the complete parse tree without actually parsing the empty string.

4.2.6 Specific elimination of hidden left recursion

For the sake of completeness, we describe a way of removing hidden left recursion without using epsilon rule elimination. The idea is that we selectively remove occurrences of nullable nonterminals which hide left recursion. In case of a nullable non-predicate A, we replace the occurrence of A by an occurrence of a new nonterminal A'. This A' is constructed so as to derive the same set of strings as A does, with the exception of ϵ.

The transformation, constructing the grammar *HLR-elim*(G) from grammar G, consists of the following steps.

1. Let G_0 be G.

2. For every rule $A \rightarrow B\alpha$ in G_0 which leads to a hidden left-recursive call (i.e. $\alpha \stackrel{G}{\rightarrow}{}^* A\beta$ for some β, and $B \stackrel{G}{\rightarrow}{}^* \epsilon$), replace the rule by $A \rightarrow \alpha$, and also add $A \rightarrow B'\alpha$ to G_0 provided B is not a predicate in G. Repeat this step until it can no longer be applied.

3. For every new nonterminal A' introduced in G_0 in step 2, or by an earlier iteration of step 3, and for every rule $A \rightarrow \alpha$ in G_0, add to G_0 the rule

   - $A' \rightarrow \alpha$ if not $\alpha \stackrel{G}{\rightarrow}{}^* \epsilon$, or rules of the form
   - $A' \rightarrow X_i' X_{i+1} \ldots X_n$ if $\alpha \stackrel{G}{\rightarrow}{}^* \epsilon$, where $\alpha = X_1 \ldots X_n$, and X_i is not a predicate.

4. Remove from G_0 all rules $A \rightarrow \alpha$ such that A was rendered unreachable by the elimination of rules in step 2.

5. Let *HLR-elim*(G) be G_0.

Example 4.2.2 Let the grammar G_3 be defined by

$$\begin{array}{lcl} A & \rightarrow & ABAa \\ A & \rightarrow & AAB \\ A & \rightarrow & \epsilon \\ B & \rightarrow & \epsilon \end{array}$$

Application of step 2 to the rule $A \rightarrow ABAa$ replaces it by $A \rightarrow BAa$, and also adds $A \rightarrow A'BAa$ because A is not a predicate in G_3. Application of step 2 to the rule $A \rightarrow BAa$, which resulted from the previous iteration, replaces it by $A \rightarrow Aa$. (Note that the rule $A \rightarrow B'Aa$ is not added because B is a predicate.) Application of step 2 to the rule $A \rightarrow AAB$ replaces it by $A \rightarrow AB$ and also adds $A \rightarrow A'AB$.

We now have

$$\begin{array}{lcl} A & \rightarrow & Aa \\ A & \rightarrow & A'BAa \\ A & \rightarrow & AB \\ A & \rightarrow & A'AB \\ A & \rightarrow & \epsilon \\ B & \rightarrow & \epsilon \end{array}$$

In step 3, a definition for A' is constructed. The rules defining A which do not derive the empty string are copied and their left hand sides are replaced by A'. The rule $A \rightarrow \epsilon$ is not used to construct the definition of A' since its rhs can only derive the empty string. The rule $A \rightarrow AB$ can derive the empty string and therefore its rhs is included in the definition of A' only after changing it to make the rule $A' \rightarrow A'B$.

The grammar *HLR-elim*(G_3) now consists of the rules

$$\begin{array}{lcl} A & \rightarrow & Aa \\ A & \rightarrow & A'BAa \\ A & \rightarrow & AB \\ A & \rightarrow & A'AB \\ A & \rightarrow & \epsilon \\ A' & \rightarrow & Aa \\ A' & \rightarrow & A'BAa \\ A' & \rightarrow & A'B \\ A' & \rightarrow & A'AB \\ B & \rightarrow & \epsilon \end{array}$$

□

The size of the grammar resulting from the application of this transformation is much smaller than that in the case of ϵ-*elim*. In fact it is only quadratic in the size of the original grammar.

The transformation *HLR-elim* is very often incorporated in the construction of parsers which can deal with hidden left recursion. An example is the variant of backtrack left-corner parsing as applied in Programmar [Meij86]. See also Chapter 2. The transformation in [GHR80, page 449] is related to *HLR-elim*, but also transforms the grammar into an extended Chomsky normal form (see Section 1.2.6).

4.3 Correctness of ϵ-LR parsing

The correctness of ϵ-LR parsing can be proved in two ways: by means of assuming correctness of LR parsing according to ϵ-*elim*(G), or by means of a derivation of ϵ-LR parsing assuming only the most elementary properties of context-free grammars.

In Section 4.3.1 we give a short proof based on the correctness of LR parsing and the fact that ϵ-*elim* preserves the language. In Section 4.3.2 we give a derivation of the ϵ-LR(0) parsing strategy. We will not make a distinction between kernel and nonkernel items. The reason for this will be discussed in Section 4.4.

4.3.1 An easy way to prove the correctness of ϵ-LR parsing

In Section 4.2.3 we derived the new parsing technique of ϵ-LR parsing. We showed that this kind of parsing is based on traditional LR parsing, with the following differences:

- Instead of using the original grammar G, the transformed grammar ϵ-*elim*(G) is used.
- No distinction is made between items derived from the same basic item. This can be seen as merging states of the LR automaton of ϵ-*elim*(G).
- Because considering derived items as the same leads to a loss of information, a new mechanisms is introduced, which checks upon reduction whether the members of the applied rule are actually on the stack and whether the goto function is defined for the lhs and the state which is on top of the stack after the members are popped.

Because the transformation ϵ-*elim* preserves the language and because the correctness of LR parsing has already been proved in literature, the correctness of ϵ-LR parsing can be proved by mentioning two points:

- The symbols on the stack and the remaining input together derive the original input, which can be proved by induction on the length of a sequence of parsing steps. This argument shows that no incorrect derivations can be found.
- For every sequence of parsing steps performed by an LR parser (LR(k), SLR(k), etc.) for ϵ-*elim*(G) there is a corresponding sequence of parsing steps performed by the corresponding type of ϵ-LR parser (ϵ-LR(k), ϵ-SLR(k), etc.) for G.

  This proves that ϵ-LR parsing cannot fail to find correct derivations by the assumption that LR parsing according to ϵ-*elim*(G) does not fail to find correct derivations.

In case of ϵ-LR(0) and ϵ-SLR parsing it can also be shown that the set of sequences of parsing steps is isomorphic with the set of sequences of the LR(0) or SLR parsers for ϵ-*elim*(G), and that the corresponding sequences are equally long. It is sufficient to prove that if a reduction can be successfully performed in an ϵ-LR parser, then it can be performed in an LR parser in the corresponding configuration.

For this purpose, suppose that in an ϵ-LR parser some reduction is possible with the item $A \to \alpha_0 A_1 \alpha_1 \dots A_m \alpha_{m\bullet} \in Q_m$ such that

- $\alpha_i \to^* \epsilon$ for $0 \leq i \leq m$,
- the topmost $2m+1$ elements of the stack are $Q_0 A_1 Q_1 \dots A_m Q_m$,
- the goto function for Q_0 and A is defined,

- in the corresponding configuration in the LR parser, the states corresponding with Q_i are called Q'_i.

From the fact that the goto function is defined for Q_0 and A we know that it is also defined for Q'_0 and A and that the item $A \to [\alpha_0]_\bullet A_1[\alpha_1] \dots A_m[\alpha_m]$ is in Q'_0. This implies that $A \to [\alpha_0]A_1[\alpha_1] \dots A_i[\alpha_i]_\bullet \dots A_m[\alpha_m]$ is in Q'_i because Q'_i is *goto* (Q'_{i-1}, A_i), for $1 \leq i \leq m$.

Therefore, in the corresponding LR parser a reduction would also take place according to the item $A \to [\alpha_0]A_1[\alpha_1] \dots A_m[\alpha_m]_\bullet$.

Regrettably, an isomorphism between sequences of parsing steps of ϵ-LR parsers and the corresponding LR parsers is not possible for ϵ-LR(k) and ϵ-LALR(k) parsing, where $k > 0$. This is because merging derived items causes loss of information on the lookahead of items. This causes the parser to be sent up blind alleys which are not considered by the corresponding LR parser.

Because ϵ-LR parsing retains the prefix-correctness of traditional LR parsing (that is, upon incorrect input the parser does not move its input pointer across the first invalid symbol), the blind alleys considered by an ϵ-LR parser but not the corresponding LR parser are of limited length, and therefore unimportant in practical cases.

Theoretically however, the extra blind alleys may be avoided by attaching the lookahead information not to the state on top of the stack before reduction but to the state on top after popping m states and grammar symbols off the stack (m as in Section 4.2.3). This means that we have lookahead (a set of strings, each of which not longer than k symbols) for each state q and nonterminal A for which *goto* (q, A) is defined.

In the cases we have examined, the number of pairs (q, A) for which *goto* (q, A) is defined is larger than the total number of items $A \to \alpha_\bullet$ in all states (about 4 to 25 times as large), so this idea is not beneficial to the memory requirements of storing lookahead information. In the case of ϵ-LR(k) parsing ($k > 0$), this idea may however lead to a small reduction of the number of states, since some states may become identical after the lookahead information has been moved to other states.

4.3.2 A derivation of ϵ-LR(0) parsing

To supply deeper insight into ϵ-LR parsing we show a second proof of the correctness of ϵ-LR(0) parsing by deriving the ϵ-LR(0) recognition algorithm starting from a simple specification of the recognition problem. The correctness of the *parsing* algorithm can be found by extending this derivation to include the performance of semantic actions.

The derivation given in this section is a variation on the derivation in [LAKA92], which proves the correctness of LR(0) parsing. Derivations of LR(k) parsing, $k \geq 0$, can be found in [Hei81, Hes92]; a derivation of LR(1) parsing is given in [Aug93, Section 3.5.3].

Recall the definition of the new closure function from Section 4.2.3. We call a set q of items *closed* if *closure* $(q) = q$.

The following functions $[q]$, one for each closed set of items q, are the main functions for recognising strings. The input string is given by $x_1 \dots x_n$.

$$\begin{aligned}[q](i) \;=\; \{(A \to \alpha_\bullet \eta \beta, j) \mid\; & A \to \alpha \eta_\bullet \beta \in q \wedge \eta \to^* \epsilon \wedge \\ & \beta \to^* x_{i+1} \dots x_j \wedge \\ & (\alpha = \epsilon \Rightarrow A = S')\}\end{aligned}$$

An item returned by such a function occurs also in q, except that the dot may have been been shifted to the left over nullable members.

As usual we have an initial state given by

$$Q_0 = \mathit{closure}\ (\{S' \to \bullet S\})$$

The recognition of a string is implemented by the following statement.

$$\begin{array}{l} \mathit{IF}\ (S' \to \bullet S, n) \in [Q_0](0) \\ \mathit{THEN\ print}\ (\text{"}\mathit{success}\text{"}) \\ \mathit{END} \end{array}$$

For the remaining part of the derivation we need an auxiliary function $\overline{[q]}$ for each closed set of items q. Beside the argument indicating the input position, it also has a grammar symbol as an argument:

$$\begin{array}{rcl} \overline{[q]}(X,i) & = & \{(A \to \alpha\bullet\eta\beta, j) \mid A \to \alpha\eta\bullet\beta \in q \wedge \eta \to^* \epsilon \wedge \\ & & \qquad \beta \to^* X\gamma \wedge \gamma \to^* x_{i+1} \ldots x_j \wedge \\ & & \qquad (\alpha = \epsilon \Rightarrow A = S')\} \end{array}$$

Following the case distinction

$$\begin{array}{rcl} \beta \to^* x_{i+1} \ldots x_j & \equiv & i = j \wedge \beta \to^* \epsilon \quad \vee \\ & & i < j \wedge \exists\gamma[\beta \to^* x_{i+1}\gamma \wedge \gamma \to^* x_{i+2} \ldots x_j] \end{array}$$

we can rewrite the definition of $[q]$ in terms of $\overline{[q]}$ as follows.

$$\begin{array}{rcl} [q](i) & = & \mathit{IF}\ i < n \\ & & \mathit{THEN}\ T_0 \cup T_1 \\ & & \mathit{ELSE}\ T_0 \\ & & \mathit{END} \\ & & \\ & & \mathit{where} \\ & & \\ & & T_0 = \{(A \to \alpha\bullet\eta\beta, i) \mid A \to \alpha\eta\bullet\beta \in q \wedge \eta \to^* \epsilon \wedge \\ & & \qquad \beta \to^* \epsilon \wedge \\ & & \qquad (\alpha = \epsilon \Rightarrow A = S')\} \\ & & \\ & & T_1 = \overline{[q]}(x_{i+1}, i+1) \end{array}$$

Because q is closed we can simplify T_0 to

$$\begin{array}{rcl} T_0 & = & \{(A \to \alpha\bullet\eta, i) \mid A \to \alpha\eta\bullet \in q \wedge \eta \to^* \epsilon \wedge \\ & & \qquad (\alpha = \epsilon \Rightarrow A = S')\} \end{array}$$

We again make a case distinction. Following

$$\begin{array}{rcl} \exists\gamma[\beta \to^* X\gamma \wedge \gamma \to^* x_{i+1} \ldots x_j] & \equiv & \exists\mu,\gamma'[\beta = \mu X\gamma' \wedge \mu \to^* \epsilon \wedge \gamma' \to^* x_{i+1} \ldots x_j] \quad \vee \\ & & \exists C,\delta,\mu,\theta,k[\beta \to^* C\delta \wedge C \to \mu X\theta \wedge \mu \to^* \epsilon \wedge \\ & & \qquad \theta \to^* x_{i+1} \ldots x_k \wedge \delta \to^* x_{k+1} \ldots x_j] \end{array}$$

we rewrite the definition of $\overline{[q]}$ to

$$\begin{aligned}
\overline{[q]}(X,i) &= S_0 \cup S_1 \\
&\quad \textit{where} \\
S_0 &= \{(A \to \alpha\bullet\eta\mu X\gamma, j) \mid A \to \alpha\eta\bullet\mu X\gamma \in q \wedge \eta \to^* \epsilon \wedge \\
&\qquad\qquad \mu \to^* \epsilon \wedge \gamma \to^* x_{i+1} \ldots x_j \wedge \\
&\qquad\qquad (\alpha = \epsilon \Rightarrow A = S')\} \\
S_1 &= \{(A \to \alpha\bullet\eta\beta, j) \mid A \to \alpha\eta\bullet\beta \in q \wedge \eta \to^* \epsilon \wedge \\
&\qquad\qquad \beta \to^* C\delta \wedge C \to \mu X\theta \wedge \mu \to^* \epsilon \wedge \\
&\qquad\qquad \theta \to^* x_{i+1} \ldots x_k \wedge \delta \to^* x_{k+1} \ldots x_j \wedge \\
&\qquad\qquad (\alpha = \epsilon \Rightarrow A = S')\}
\end{aligned}$$

Because of the assumption that q is closed, S_0 can be simplified to

$$\begin{aligned}
S_0 &= \{(A \to \alpha\bullet\eta X\gamma, j) \mid A \to \alpha\eta\bullet X\gamma \in q \wedge \eta \to^* \epsilon \wedge \\
&\qquad\qquad \gamma \to^* x_{i+1} \ldots x_j \wedge \\
&\qquad\qquad (\alpha = \epsilon \Rightarrow A = S')\}
\end{aligned}$$

We now define the goto function for ϵ-LR parsing. It is the same as that for normal LR parsing apart from the different definition of the closure function.

$$\textit{goto}\ (q, X) = \textit{closure}\ (\{A \to \alpha X\bullet\beta \mid A \to \alpha\bullet X\beta \in q\})$$

We now want to prove that S_0 is equal to

$$\begin{aligned}
S_0' &= \{(A \to \alpha\bullet\eta X\gamma, j) \mid (A \to \alpha\eta X\bullet\gamma, j) \in [\textit{goto}\ (q, X)](i) \wedge \eta \to^* \epsilon \wedge \\
&\qquad\qquad (\alpha = \epsilon \Rightarrow A = S')\}
\end{aligned}$$

First we prove that $S_0 \subseteq S_0'$ as follows.

Assume that $A \to \alpha\bullet X\gamma \in q$ and $\gamma \to^* x_{i+1} \ldots x_j$. Then $A \to \alpha X\bullet\gamma \in \textit{goto}\ (q, X)$ and therefore $(A \to \alpha X\bullet\gamma, j) \in [\textit{goto}\ (q, X)](i)$.

Conversely, we prove that $S_0' \subseteq S_0$ as follows.

Assume that $(A \to \alpha X\bullet\gamma, j) \in [\textit{goto}\ (q, X)](i)$. Then for some η and β such that $\eta\beta = \gamma$ and $\eta \to^* \epsilon$ we have that $A \to \alpha X\eta\bullet\beta \in \textit{goto}\ (q, X)$ and $\beta \to^* x_{i+1} \ldots x_j$.

We have one of the following.

(1) There is some $B \to \delta\bullet X\theta \in q$ such that $\theta \to^* A\kappa$ and $\alpha X\eta \to^* \epsilon$; or

(2) There are α' and η' such that $\alpha' X\eta' = \alpha X\eta$, $A \to \alpha'\bullet X\eta'\beta \in q$, $\eta' \to^* \epsilon$, and

- **(2a)** $\alpha' = \alpha$ and $\eta' = \eta$; or
- **(2b)** $\alpha' X\eta' = \alpha' X\delta X\eta = \alpha X\eta$ (note that in this case $X\delta \to^* \epsilon$); or
- **(2c)** $\alpha' X\eta' = \alpha X\delta X\eta' = \alpha X\eta$ (also in this case $X\delta \to^* \epsilon$).

Taking into account that q is closed, we can immediately conclude that $A \to \alpha\bullet X\gamma \in q$ and that $\gamma \to^* x_{i+1} \ldots x_j$, in cases **(1)**, **(2a)**, and **(2b)**, thus proving the claim.

Only in case **(2c)** we need to investigate how the dot is shifted to the left over nullable members in the definitions of both S_0 and S_0', using the fact that $X\delta \rightarrow^* \epsilon$. We leave the details to the reader.

This concludes the proof of $S_0' \subseteq S_0$.

By folding the original definition of $\overline{[q]}$, we can rewrite the definition of S_1 to

$$\begin{aligned} S_1 \;=\; \{(A \rightarrow \alpha\bullet\beta, j) \mid\; & (A \rightarrow \alpha\bullet\beta, j) \in \overline{[q]}(C,k) \wedge \\ & C \rightarrow \mu X\theta \wedge \mu \rightarrow^* \epsilon \wedge \theta \rightarrow^* x_{i+1} \ldots x_k\} \end{aligned}$$

We now want to prove that S_1 is equal to

$$\begin{aligned} S_1' \;=\; \{(A \rightarrow \alpha\bullet\beta, j) \mid\; & (A \rightarrow \alpha\bullet\beta, j) \in \overline{[q]}(C,k) \wedge \\ & (C \rightarrow \mu X\bullet\theta, k) \in [goto\ (q,X)](i) \wedge \mu \rightarrow^* \epsilon\} \end{aligned}$$

First assume that $C \rightarrow \mu X\theta$ and $\mu \rightarrow^* \epsilon$.

We prove that $S_1' \subseteq S_1$ as follows. If we assume that $(C \rightarrow \mu X\bullet\theta, k) \in [goto\ (q,X)](i)$ then we have $\theta \rightarrow^* x_{i+1} \ldots x_k$.

Conversely, we prove that $S_1 \subseteq S_1'$ as follows. Assume that $(A \rightarrow \alpha\bullet\beta, j) \in \overline{[q]}(C,k)$, $C \rightarrow \mu X\theta$, $\mu \rightarrow^* \epsilon$, and $\theta \rightarrow^* x_{i+1} \ldots x_k$. Then $\beta \rightarrow^* C\delta$, for some δ, and therefore $C \rightarrow \mu\bullet X\theta \in q$, taking into account that q is closed. From this we conclude $C \rightarrow \mu X\bullet\theta \in goto\ (q,X)$ and therefore $(C \rightarrow \mu X\bullet\theta, k) \in [goto\ (q,X)](i)$.

Summarising the definitions of $[q](i)$ and $\overline{[q]}(X,i)$ we have

$$\begin{aligned} [q](i) \;=\;\; & \mathit{IF}\ i < n \\ & \mathit{THEN}\ T_0 \cup T_1 \\ & \mathit{ELSE}\ T_0 \\ & \mathit{END} \\ \\ & \mathit{where} \\ \\ & T_0 \;=\; \{(A \rightarrow \alpha\bullet\eta, i) \mid A \rightarrow \alpha\eta\bullet \in q \wedge \eta \rightarrow^* \epsilon \wedge \\ & \qquad\qquad (\alpha = \epsilon \Rightarrow A = S')\} \\ \\ & T_1 \;=\; \overline{[q]}(x_{i+1}, i+1) \end{aligned}$$

and

$$\begin{aligned} \overline{[q]}(X,i) \;=\;\; & \{(A \rightarrow \alpha\bullet\eta X\gamma, j) \mid (A \rightarrow \alpha\eta X\bullet\gamma, j) \in [goto\ (q,X)](i) \wedge \eta \rightarrow^* \epsilon \wedge \\ & \qquad\qquad (\alpha = \epsilon \Rightarrow A = S')\} \\ & \cup \\ & \{(A \rightarrow \alpha\bullet\beta, j) \mid (A \rightarrow \alpha\bullet\beta, j) \in \overline{[q]}(C,k) \wedge \\ & \qquad\qquad (C \rightarrow \mu X\bullet\theta, k) \in [goto\ (q,X)](i) \wedge \mu \rightarrow^* \epsilon\} \end{aligned}$$

Before making a first step to change this specification into a program, we define the following functions, which capture the movements of the dots within items.

$$\begin{array}{rcl} \mathit{final}\ (A \to \alpha\bullet\eta) & \equiv & \eta = \epsilon \\ \mathit{invent}\ (A \to \alpha\bullet\beta, j) & = & \{(A \to \alpha'\bullet\eta\beta, j) \mid \alpha'\eta = \alpha \wedge \eta \to^* \epsilon \wedge \\ & & \qquad (\alpha' = \epsilon \Rightarrow A = S')\} \end{array}$$

The last function above is called *invent* because it moves the dot over nonterminals for which no concrete parse tree needs to be found. The mere fact that these nonterminals are nullable is a sufficient condition.

We further have

$$\begin{array}{rcl} \mathit{pop}\ (A \to \alpha X\bullet\beta) & = & A \to \alpha\bullet X\beta \\ \mathit{after}\ (A \to \alpha\bullet\beta, X) & \equiv & \exists\alpha[\alpha = \alpha' X] \\ \mathit{initial}\ (A \to \eta\bullet\beta) & \equiv & \eta \to^* \epsilon \\ \mathit{lhs}\ (A \to \alpha\bullet\beta) & = & A \\ \mathit{rule}\ (A \to \alpha\bullet) & = & A \to \alpha \end{array}$$

To find a new definition for $[q](i)$ first note that

$$\begin{array}{rcl} [\emptyset](i) & = & \emptyset \\ \overline{[\emptyset]}(X, i) & = & \emptyset \end{array}$$

for every i and X.

For nonempty closed sets of items q we can incorporate the above definitions to arrive at

$$\begin{array}{rcl} [q](i) & = & \mathit{IF}\ i < n \\ & & \mathit{THEN}\ T_0 \cup T_1 \\ & & \mathit{ELSE}\ T_0 \\ & & \mathit{END} \\ & & \\ & & \mathit{where} \\ & & \\ & & T_0 = \bigcup\{\mathit{invent}\ (I, i) \mid I \in q \wedge \mathit{final}\ (I)\} \\ & & \\ & & T_1 = \overline{[q]}(x_{i+1}, i+1) \end{array}$$

and

$$\begin{array}{rcl} \overline{[q]}(X, i) & = & \bigcup\{\mathit{invent}\ (\mathit{pop}\ (I), j) \mid (I, j) \in [\mathit{goto}\ (q, X)](i) \wedge \mathit{after}\ (I, X)\} \\ & & \cup \\ & & \bigcup\{\overline{[q]}(\mathit{lhs}\ (I), j) \mid (I, j) \in [\mathit{goto}\ (q, X)](i) \wedge \mathit{after}\ (I, X) \wedge \mathit{initial}\ (\mathit{pop}\ (I))\} \end{array}$$

We now change the definitions of $[q]$ and $\overline{[q]}$ to nondeterministic procedures which may fail or return one of the elements of the set returned by the old definition.

We use the program construct

$$|[\ part_1 \parallel part_2 \parallel \ \ldots \ \parallel part_m\ ]|$$

to denote nondeterministic choice between one of the parts. The parts may have guards, in which case they are of the form

$$condition \longrightarrow statement$$

In such a construct the part fails if the condition does not hold, and otherwise the statement is executed.

We use the construct

$$part_1; part_2; \ldots; part_m$$

to denote successive execution of the parts, in the order indicated.

In general, if a part of a construct fails, then the construct as a whole fails.

We further have

$$SOME\ obj\ SUCH\ THAT\ condition\ END$$

This construct fails if there are no objects satisfying the condition, and otherwise yields one of the objects satisfying the condition.

In the definitions of $[q]$ and $\overline{[q]}$ below we have also changed the input pointer i from a parameter to a global variable. This requires the definition of *invent* to be changed in a straightforward way to work on items:

$$\begin{aligned} invent\ (A \to \alpha\bullet\beta)\ :\ & SOME\ A \to \alpha'\bullet\eta\beta\ SUCH\ THAT\ \alpha'\eta = \alpha \wedge \eta \to^* \epsilon \wedge \\ & \qquad\qquad (\alpha' = \epsilon \Rightarrow A = S') \\ & END\ . \end{aligned}$$

We now have

$$[\emptyset]\ :\ FAIL\ .$$

and for nonempty closed sets of items q we have

$$\begin{aligned} [q]\ :\ & |[\\ & \quad invent\ (SOME\ I\ SUCH\ THAT\ I \in q \wedge final\ (I)\ END\ ) \\ & \parallel \\ & \quad i < n \longrightarrow i := i + 1;\ \overline{[q]}(x_i) \\ &]|\ . \end{aligned}$$

and

$$
\begin{array}{lll}
\overline{[q]}(X) & : & I := [goto\ (q, X)]; \\
 & & |[after\ (I, X) \longrightarrow I := pop\ (I); \\
 & & \quad |[\\
 & & \quad\quad initial\ (I) \longrightarrow \overline{[q]}(lhs\ (I)) \\
 & & \quad \| \\
 & & \quad\quad invent\ (I) \\
 & & \quad]| \\
 & &]| \ .
\end{array}
$$

The recognition of a string is now implemented by

$$
\begin{array}{l}
i := 0; \\
I := [Q_0]; \\
|[\ I = (S' \rightarrow \bullet S) \wedge i = n \longrightarrow print\ (\text{"success"}) \\
]| \ .
\end{array}
$$

The formulation of the parsing strategy at this stage is reminiscent of recursive ascent parsers [KA88, Rob88, Rob89, BC88].

The next step includes the transformation from an implicit return stack to an explicit one. Furthermore, the arguments with which the functions $\overline{[q]}$ are called are incorporated into the stack.

We introduce a new construct of the form

REPEAT statement UNTIL condition END

which executes the statement until either the condition holds (the construct as a whole succeeds) or the statement fails (the construct as a whole fails).

The statement *assert* (*condition*) succeeds if the condition holds and fails otherwise.

We now have the program

$$
\begin{array}{l}
recognised := false; \\
i := 0; \\
push\ (Q_0); \\
REPEAT\ parse\ UNTIL\ recognised\ END \ .
\end{array}
$$

where

$$
\begin{array}{lll}
parse & : & q := top\ of\ stack; \\
 & & assert\ (q \neq \emptyset); \\
 & & |[\\
 & & \quad reduce\ (rule\ (SOME\ I\ SUCH\ THAT\ I \in q \wedge final\ (I)\ END\ )) \\
 & & \| \\
 & & \quad i < n \longrightarrow i := i + 1;\ push\ (x_i);\ push\ (goto\ (q, x_i)) \\
 & &]| \ .
\end{array}
$$

reduce $(A \to \alpha)$: Let $X_1, \ldots, X_m$ be such that there are $\alpha_0, \ldots, \alpha_m$ with

- $\alpha = \alpha_0 X_1 \alpha_1 \ldots X_m \alpha_m$
- $\alpha_0 \to^* \epsilon \wedge \ldots \wedge \alpha_m \to^* \epsilon$
- The top-most m grammar symbols on the stack are $X_1, \ldots, X_m$
- $m = 0 \Rightarrow A = S'$

Pop the top-most m grammar symbols and states off the stack;
$q :=$ *top of stack*;
IF $q = Q_0 \wedge A = S' \wedge i = n$
THEN print (*"success"*); *recognised* := *true*
ELSE push (A); *push* $(goto\ (q, A))$
END .

The above definition formally describes ϵ-LR(0) parsing. An executable parsing strategy is obtained by implementing the nondeterministic constructs using a backtracking mechanism or using a graph-structured stack [Tom86] (or equivalently, using a parse matrix [Lan74, BL89]).

4.4 Calculation of items

In this section we investigate the special properties of the closure function for ϵ-LR parsing. First we discuss the closure function for ϵ-LR(k) parsing and then the equivalent notion of kernel items in ϵ-LR parsing.

4.4.1 The closure function for ϵ-LR(k) parsing

If w is a string and k a natural number, then $k : w$ denotes w if the length of w is less than k, and otherwise it denotes the prefix of w of length k. We use lookaheads which may be less than k symbols long to indicate that the end of the string has been reached.

The initial state for ϵ-LR(k) parsing ($k > 0$) is

$$Q_0 = closure\ (\{[S' \to \bullet S, \epsilon]\})$$

The closure function for ϵ-LR(k) parsing is

$$\begin{aligned} closure\ (q) \quad = \quad & \{[B \to \delta\bullet\theta, x] \mid [A \to \alpha\bullet\beta, w] \in q \wedge \beta \to^* B\gamma \wedge B \to \delta\theta \wedge \\ & \qquad \exists v[v \neq \epsilon \wedge \delta\theta \to^* v] \wedge \delta \to^* \epsilon \wedge \\ & \qquad \exists y[\gamma \to^* y \wedge x = k : yw]\} \\ & \cup \\ & \{[A \to \alpha\delta\bullet\beta, w] \mid [A \to \alpha\bullet\delta\beta, w] \in q \wedge \delta \to^* \epsilon\} \end{aligned}$$

4.4.2 The determination of smallest representative sets

In traditional LR parsing, items are divided into *kernel* items and *nonkernel* items. Kernel items are $S' \rightarrow \bullet S$ and all items whose dots are not at the left end. The nonkernel items are all the others. (At this stage we abstain from lookahead.)

As we will only be looking in this section at sets of items which are either Q_0 or of the form *goto* (q, X), which result after application of the closure function, we have that the kernel items from a set of items q are a *representative subset* of q. This means that we can

- construct the complete set of items q by applying the closure function to the representative subset, and
- determine whether two sets of items are equal by determining the equality of their representative subsets.

Because the set of kernel items from a set q is in general much smaller than q itself, kernel items are very useful for the efficient generation of LR parsers.

Regrettably, in the case that the grammar contains many epsilon rules, the set of kernel items from a set q may not be much smaller than q itself. In this case therefore, kernel items may not be very useful for generation of ϵ-LR parsers.

Another approach to finding representative subsets for traditional LR parsing can be given in terms of the stages in which the goto function is executed. According to this principle, the representative subset of *goto* (q, X) is

$$K(q, X) = \{A \rightarrow \alpha X \bullet \beta | A \rightarrow \alpha \bullet X \beta \in q\}$$

and other items in *goto* (q, X) are obtained by applying the closure function to $K(q, X)$.

In the case of traditional LR parsing, K computes exactly the kernel items in *goto* (q, X), and therefore the two methods for finding representative subsets are equivalent. That this does not hold for ϵ-LR parsing can be easily seen by investigating the definition of *closure* in Section 4.2.3: according to the second part

$$\{A \rightarrow \alpha\delta \bullet \beta \mid A \rightarrow \alpha \bullet \delta\beta \in q \wedge \delta \rightarrow^* \epsilon\}$$

in this definition, the dot can be shifted over nullable members and therefore new items can be added whose dots are not at the left end. Therefore, some kernel items may not be in $K(q, X)$.

It turns out that we can also not use K for finding representative subsets in the case of ϵ-LR parsing. The reason is that K does not provide a *well-defined* method to find representative subsets. I.e. for some grammars we can find sets of items q_1 and q_2 and symbols X and Y such that *goto* $(q_1, X) =$ *goto* (q_2, Y) but $K(q_1, X) \neq K(q_2, Y)$.

The solution that we propose is more refined than the methods in traditional LR parsing.

First, we determine the equivalence relation of mutually left-recursive nonterminals, whose classes are denoted by $[A]$. Thus, $[A] = \{B | A \rightarrow^* B\alpha \wedge B \rightarrow^* A\beta\}$.

A nice property of these classes is that $A \rightarrow \bullet\alpha \in q$ and $B \in [A]$ together imply that $B \rightarrow \bullet\beta \in q$ for every rule $B \rightarrow \beta$. Using this fact, we can replace every item $A \rightarrow \bullet\alpha$ in q by $[A]$ without loss of information.

We define the set Z to be the union of the set of all items and the set of equivalence classes of mutually left-recursive nonterminals. The variables $E, E', \ldots$ range over elements from Z.

Our goal is to find a representative set $q' \subseteq Z$ for each set of items q.

First, we define the binary relation *induces* on elements from Z such that

- *induces* (I, J) for items I and J
  if and only if $I = A \to \alpha{\bullet}B\beta$ and $J = A \to \alpha B{\bullet}\beta$ and $B \to^* \epsilon$

- *induces* (I, E) for item I and class E
  if and only if $I = A \to \alpha{\bullet}B\beta$ and $B \in E$

- *induces* (E, E') for classes E and E'
  if and only if $E \neq E'$ and there are $A \in E$ and $B \in E'$ such that $A \to \alpha B\beta$ and $\alpha \to^* \epsilon$

- *induces* (E, I) for class E and item I
  if and only if there is $A \in E$ such that $I = A \to \alpha{\bullet}\beta$ and $\alpha \to^* \epsilon$

The smallest set *repr* $(q) \subseteq Z$ representing a set of items q can now be determined by the following steps:

1. Determine $q_1 \subseteq Z$ by replacing in q every item $A \to {\bullet}\alpha$ by $[A]$.

2. Let q_2 be the subset of q_1 which results from eliminating all items I such that *induces* (E, I) for some equivalence class $E \in q_1$.

3. Determine the set *repr* (q) defined by $\{E \in q_2 | \neg\exists E' \in q_2[\mathit{induces}\ (E', E)\}$.

The reason that no information is lost in step 3 is that the relation *induces* restricted to q_2 is not cyclic.

That *repr* (q) is the smallest set $q' \subseteq Z$ representing q can be formalized by stating that it is the smallest subset q' of Z such that *closure* $(q') = q$, where the definition of *closure* is redefined to

$$\begin{aligned} \mathit{closure}\ (q) \quad = \quad & \{B \to \delta{\bullet}\theta \mid ((A \to \alpha{\bullet}\beta \in q \wedge \beta \to^* B\gamma) \quad \vee \quad ([A] \in q \wedge A \to^* B\gamma)) \wedge \\ & \qquad\qquad B \to \delta\theta \wedge \\ & \qquad\qquad \exists v[v \neq \epsilon \wedge \delta\theta \to^* v] \wedge \delta \to^* \epsilon\} \\ & \cup \\ & \{A \to \alpha\delta{\bullet}\beta \mid A \to \alpha{\bullet}\delta\beta \in q \wedge \delta \to^* \epsilon\} \end{aligned}$$

It is evident that the *repr* $(\mathit{goto}\ (q, X))$ must be calculated from the $K(q, X)$ rather than from their closures *goto* (q, X) if efficient parser construction is required. The appropriate restatement of the algorithm calculating *repr* is straightforward.

4.5 Memory requirements

In this chapter we have described three methods of making the (generalized) LR parsing technique applicable to hidden left-recursive grammars:

1. Apply ϵ-*elim* to the grammar before constructing the LR automaton.
2. Apply *HLR-elim* to the grammar before constructing the LR automaton.
3. Construct the ϵ-LR automaton as opposed to the LR automaton.

The last method above is derived from the first one in the sense that an ϵ-LR automaton can be seen as a compressed LR automaton for the transformed grammar ϵ-*elim*(G). The second method is independent from the other two methods.

To investigate the static memory requirements of these methods, we have determined the number of states of the resulting automata for various grammars.

We first investigate the number of states for three kinds of characteristic grammars:

For every $k \geq 0$ we have the grammar G_1^k defined by the rules

$$\begin{array}{lcl} S & \rightarrow & B_1 \ldots B_k c \\ B_1 & \rightarrow & \epsilon \\ B_1 & \rightarrow & b_1 \\ & \vdots & \\ B_k & \rightarrow & \epsilon \\ B_k & \rightarrow & b_k \end{array}$$

For every $k \geq 1$ we have the grammar G_2^k defined by the rules

$$\begin{array}{lcl} S & \rightarrow & B_1 \ldots B_k S c \\ S & \rightarrow & d \\ B_1 & \rightarrow & \epsilon \\ B_1 & \rightarrow & b_1 \\ & \vdots & \\ B_k & \rightarrow & \epsilon \\ B_k & \rightarrow & b_k \end{array}$$

For every $k \geq 2$ we have the grammar G_3^k defined by the rules

$$\begin{array}{lcl} S & \rightarrow & B_1 \ldots B_k c \\ B_1 & \rightarrow & \epsilon \\ B_1 & \rightarrow & S \\ & \vdots & \\ B_k & \rightarrow & \epsilon \\ B_k & \rightarrow & S \end{array}$$

The grammars of the first group contain no left recursion. The grammars of the second group contain one occurrence of hidden left recursion, and there are k nullable nonterminals hiding the left recursion. The grammars of the third group contain $k-1$ occurrences of hidden left recursion, the j-th one of which is hidden by $j-1$ nullable nonterminals.

Method of construction	G_1^k $(k \geq 0)$	G_2^k $(k \geq 1)$	G_3^k $(k \geq 2)$	G_{Deltra}
LR(0) for G	$2 \cdot k + 3$	$2 \cdot k + 5$	$2 \cdot k + 2$	855
LR(0) for ϵ-*elim*(G)	$2^{k+1} + k + 1$	$3 \cdot 2^k + k + 1$	$2^{k+1} + 2$	1430
LR(0) for *HLR-elim*(G)	$2 \cdot k + 3$	$\frac{1}{2} \cdot k^2 + 4\frac{1}{2} \cdot k + 3$	$\frac{1}{2} \cdot k^2 + 2\frac{1}{2} \cdot k + 1$	1477
ϵ-LR(0) for G	$2 \cdot k + 3$	$k + 6$	6	709

Figure 4.5: The numbers of states resulting from four different methods of constructing LR and ϵ-LR automata.

Figure 4.5 shows the numbers of states of various automata for these grammars. It also shows the numbers of states of the LR(0) automata for the original grammars. This kind of automaton does of course not terminate in the case of hidden left recursion, except if the nondeterminism is realized using cyclic graph-structured stacks, against which we raised some objections in Section 4.1.

These results show that the number of states is always smallest for the ϵ-LR(0) automata. A surprising case is the group of grammars G_3^k, where the number of states of ϵ-LR(0) is 6, regardless of k, whereas the numbers of states of the LR(0) automata for ϵ-*elim*(G) and *HLR-elim*(G) are exponential and quadratic in k, respectively.

In the above grammars we have found some features which cause a difference in the number of states of the automata constructed by the mentioned four methods. The results suggest that ϵ-LR parsing is more efficient in the number of states for grammars containing more hidden left recursion.

The number of states of LR and ϵ-LR automata is however rather unpredictable, and therefore the above relations between the number of states for the four methods may deviate dramatically from those in the case of practical grammars.

Practical hidden left-recursive grammars do however not yet occur frequently in natural language research. The reason is that they are often considered "ill-designed" [NF89] as they cannot be handled using most parsing techniques.

Fortunately, we have been able to find a practical grammar which contains enough hidden left recursion to perform a serious comparison. This grammar is the context-free part of the Deltra grammar, developed at the Delft University of Technology [SB90]. After elimination of the occurrences and definitions of all predicates, this grammar contains 846 rules and 281 nonterminals, 120 of which are nullable. Hidden left recursion occurs in the definitions of 62 nonterminals. Rules are up to 7 members long, the average length being about 1.74 members.

The numbers of states of the automata for this grammar are given in Figure 4.5. These data suggest that for practical grammars containing much hidden left recursion, the relation between the numbers of states of the four different automata is roughly the same as for the

three groups of small grammars G_1^k, G_2^k, and G_3^k: the LR(0) automata for ϵ-*elim*(G) and *HLR-elim*(G) both have a large number of states. (Surprisingly enough, the former has a *smaller* number of states than the latter, although ϵ-*elim*(G) is about 50 % larger than *HLR-elim*(G), measured in the number of symbols.) The ϵ-LR(0) automaton for G has the smallest number of states, even smaller than the number of states of the LR(0) automaton for G.

Although these results are favourable to ϵ-LR parsing as a parsing technique requiring small parsers, not for all practical grammars will ϵ-LR automata be smaller than their traditional LR counterparts. Especially for grammars which are not left-recursive, we have found small increases in the number of states. We consider these grammars not characteristic however because they were developed explicitly for top-down parsing.

4.6 Conclusions

We have described a new solution to adapt (generalized) LR parsing to grammars with hidden left recursion. Also LR parsing of cyclic grammars has been discussed. We claim that our solution yields smaller parsers than other solutions, measured in the number of states. This has been corroborated by theoretical data on small grammars and by an empirical test on a practical grammar for a natural language.

Our solution requires the investigation of the parse stack. We feel however that this does not lead to deterioration of the time complexity of parsing: investigation of the stack for each reduction with some rule requires a constant amount of time. This amount of time is linear in the length of that rule, provided investigation of the symbols on the stack is implemented using a finite state automaton.

The results of our research are relevant to realization of generalized LR parsing using backtracking (possibly in combination with memo-functions) or using acyclic graph-structured stacks. Furthermore, various degrees of lookahead may be used.

We hope that our research will convince linguists and computer scientists that hidden left recursion is not an obstacle to efficient LR parsing of grammars. This may in the long term simplify the development of grammars, since hidden left recursion does not have to be avoided or eliminated.

Chapter 5

Top-Down Parsing for Left-Recursive Grammars

In this chapter we discuss a parsing algorithm called cancellation parsing. Deterministic cancellation parsing with lookahead k can handle the C(k) grammars, which include the LL(k) grammars and are contained in the LC(k) grammars.

The C(k) property is of theoretical interest in that it shows how to formalise the intuitive notion of the LL(k) property extended with the possibility of left recursion. The top-down nature of cancellation parsing gives it advantages over many other parsing algorithms.

No extensive analysis is needed to construct a nondeterministic cancellation parser. This makes cancellation parsing particularly useful when fast parser generation is required.

We also show how nondeterministic cancellation parsing can be refined to handle arbitrary grammars.

5.1 Introduction

Top-down (TD) parsing has strong advantages over other parsing algorithms:

- In the case of context-free grammars extended with arguments (such as definite clause grammars or attribute grammars), top-to-bottom flow of argument values is possible, which may serve to reject incorrect derivations at an early stage.

- The structure of a top-down (i.e. recursive-descent) parser is closely related to the structure of a grammar. This allow easy debugging of a grammar. Furthermore, a minimal amount of resources is needed to construct a parser.

Conventional top-down parsing can however not deal with left recursion. This is very unfortunate, because certain constructions of programming languages and of natural languages can be described most naturally using left recursion.

In this chapter we discuss *cancellation parsing*, which has all the advantages of top-down parsing mentioned above, but which can also deal with left recursion. The top-down nature of cancellation parsing also makes it of important theoretical interest, because it can be seen as the weakest extension to top-down parsing that can handle left recursion.

Cancellation parsing may be particularly useful for the implementation of definite clause grammars (DCGs), which can be argued as follows:

In the case that a Prolog environment is based on interpretation of Prolog programs as opposed to compilation, the implementation of definite clause grammars in that environment has to satisfy a special requirement: no compilation or analysis of a grammar should be needed before starting the parsing process; or in other words, it should be possible to interpret each grammar rule individually as a Prolog clause. The collection of all such Prolog clauses, possibly together with some auxiliary definitions, constitutes the parser.

This requirement excludes parsing algorithms requiring complicated parser-construction algorithms, such as deterministic, nondeterministic, or generalized LR parsing [RH88, Nil86, TN91a], and algorithms which apply lookahead [CH87]. This also excludes the application of extensive transformations [Dym92].

An additional requirement for a parsing algorithm for DCGs may be that the parsing procedures should not cause any side-effects. This is especially so in Prolog environments without non-logical operations. In this case tabular algorithms [Lan88a] cannot be used, such as Earley's algorithm [PW83] or tabular left-corner parsing [UOKT84, MS87] (see also Chapter 2).

Even when the context-free part of a tabular parsing algorithm can be described in a purely functional way, then still copying of DCG arguments is necessary [MS87], which can only be achieved using non-logical operations.[1]

An obvious parsing algorithm which satisfies both requirements mentioned above is top-down parsing, which is used in most implementations of DCGs [PW80]. TD parsing can however not handle left recursion, which seriously undermines the usefulness of DCGs.

Left-corner (LC) parsing is a viable alternative to TD parsing in DCG implementations [M$^+$83]. It does not completely satisfy the requirement that no analysis is needed to construct the parser however, because top-down filtering depends on results of an analysis. Omitting top-down filtering, as suggested in [M$^+$83], worsens the time complexity. Furthermore, the bottom-up nature of LC parsing complicates early rejection of derivations by means of failed unification.

The parsing algorithm that we introduce in this chapter, called *cancellation parsing*, shares with LC parsing the property that it can handle left recursion. It has however a more top-down nature, so that

- no analysis is needed to construct the parser, and
- in case a small analysis is allowed, the basic parsing algorithm can be adapted to allow more top-to-bottom flow of argument values (Section 5.2.6). This allows early rejection of derivations.

Section 5.2 explains nondeterministic cancellation parsing, starting at nondeterministic TD and LC parsing. We also show how the construction of a cancellation parser can be seen as a transformation between context-free grammars. The correctness of cancellation parsing is discussed in Section 5.3.

[1] Restricting the formalism of DCGs so that only bottom-to-top flow of argument values is possible is a solution to this problem [MS87], but we feel that this restriction undermines the usefulness of DCGs.

In Section 5.4 we investigate how cancellation parsing can be extended to use lookahead. Section 5.5 discusses the class of grammars that cancellation parsing can handle and how the basic parsing technique can be refined to handle even larger classes. The computational feasibility of cancellation parsing is argued in Section 5.6. Two ideas from the existing literature which are related to cancellation parsing are discussed in Section 5.7.

Section 5.8 serves as an appendix to this chapter, elaborating on Section 5.2.3, which discusses left-corner parsing.

Most of the notation and terminology in this chapter is explained in Section 1.1.

For notational convenience we will use a somewhat different notation for grammar rules, inspired by the notation generally used for definite clause grammars: members in a rhs are separated by commas and the rhs is terminated by a full stop. Terminals are enclosed in square brackets. Logical variables in definite clause grammars are written as lower-case letters.

As usual, we will denote the transitive closure of a relation R by R^+ and the reflexive and transitive closure by R^*.

Sometimes $e_1\ R^*\ e_n$ denotes some sequence $e_1\ R\ e_2\ R\ \ldots\ R\ e_{n-1}\ R\ e_n$. In this manner $\alpha_1 \rightarrow^* \alpha_n$ denotes some sequence $\alpha_1 \rightarrow \alpha_2 \rightarrow \ldots \rightarrow \alpha_n$. We call such a sequence a *derivation*.

Special derivations are denoted as follows:

- $\alpha \rightarrow_l^* \beta$ denotes a *leftmost* derivation, i.e. a derivation where it is always the leftmost nonterminal which is replaced by a rhs of a rule.

- $\alpha \rightarrow_{lc}^* \beta$ denotes a leftmost derivation where all rules used have a rhs beginning with a nonterminal.

- By $\alpha_\bullet\beta \rightarrow^* \gamma\beta$ we mean a derivation where β has not been affected, i.e. no nonterminal in β has been rewritten.

- $\alpha \rightarrow_l^* x\underline{A}\beta$ denotes a derivation $\alpha \rightarrow_l^* x_1 B\beta_1 \rightarrow_l x_1\gamma_\bullet A\beta_2\beta_1 \rightarrow_l^* x_1x_2A\beta_2\beta_1$, where $x = x_1x_2$, $\beta = \beta_2\beta_1$ and $\gamma \neq \epsilon$, or the trivial derivation S where $\alpha = S$, $x = \beta = \epsilon$ and $A = S$. Intuitively, this means that A is not the first member in the rhs of the rule that introduced it.

We call $\alpha \in V^*$ a *sentential form* if $S \rightarrow^* \alpha$.

We call a nonterminal A *reachable* if $S \rightarrow^* \alpha A\beta$ for some α and β. We call a grammar *reduced* if every nonterminal is reachable and generates some terminal string. We tacitly assume that given grammars are reduced and when we give a transformation between context-free grammars we integrate into the transformation the *reduction* of the grammar, which is the removal of nonterminals and rules to make the grammar reduced as discussed in [AU72].

We define a *spine* to be a path in a parse tree which begins at some node, then proceeds downwards every time taking the leftmost son. We call a spine *maximal* if it is not included in a larger spine.

5.2 Introduction to cancellation parsing

In this section we consider a number of nondeterministic parsing algorithms. We start with TD and LC parsing and finally arrive at the new kind of parsing, called *cancellation parsing*.

5.2.1 Standard interpretation of DCGs

The key idea to top-down parsing of grammars using Prolog is to make explicit in grammar rules how the members of rules read parts of the input: grammar rules

$$A \rightarrow Y_1, Y_2, \ldots, Y_n.$$

are turned into Prolog clauses of the form

$$A(i_0, i_n) :- Y_1(i_0, i_1),\ Y_2(i_1, i_2),\ \ldots,\ Y_n(i_{n-1}, i_n).$$

The input at different stages is represented by the variables $i_0, i_1, \ldots, i_n$. More precisely, in a call $Y(i, i')$, i is the remaining input when the call is initiated, and i' (a suffix of i) is the remaining input after the call has finished and read some string generated by Y. The variables $i_0, i_1, \ldots, i_n$ are called *string arguments* ([PW83], or *ioargs* in [Szp87]).

If Y_m $(1 \leq m \leq n)$ is a terminal $[a]$, $Y_m(i_{m-1}, i_m)$ is replaced by a call $token('a', i_{m-1}, i_m)$. The definition of *token* consists of the unit clause

$$token(a, [a|i], i).$$

where $[a|i]$ represents a list with head a and tail i.

The parsing process is activated by calling $S(i, [\,])$, where i is the input represented as a list of terminals.

The arguments already present in the DCG rules are maintained in the translation into Prolog. Restrictions on these arguments can be expressed by means of the *extra conditions* [AD89, MS87] in DCG rules, which are written between braces. The extra conditions remain unchanged by the translation into Prolog (except that the braces themselves are removed).

The above translation of DCGs into executable Prolog programs is very simple and therefore DCGs themselves can be seen as an executable formalism [PW80]. Both LC parsing and cancellation parsing, to be discussed shortly, can also be described in terms of string arguments. We simplify the exposition of these algorithms by keeping the string arguments implicit, and transforming a grammar G into a DCG G', instead of directly into Prolog.

Of course, G may itself be a DCG, in which case the parser construction is a source-to-source transformation. To limit the amount of confusion which may result from this, we use the term *DCG* exclusively for a description of a parser where the string arguments are kept implicit. We use the more neutral term *grammar* for any grammatical description with a context-free basis, without an explicit procedural interpretation.

5.2.2 Top-down parsing

Consider the left-recursive grammar G_1, defined by

$$\begin{array}{lcl} S & \rightarrow & A,\ [a]. \\ A & \rightarrow & B,\ [b]. \\ A & \rightarrow & [c],\ [c]. \\ B & \rightarrow & S,\ [s]. \end{array}$$

The top-down recognizer resulting from the identity translation into a DCG will not terminate on any input: consider the scenario where S calls A which calls B which calls S which calls A again, and so on.

There are some well-known solutions to force termination which make use of the length of the input. The easiest solution is to record the sum of the minimal lengths of strings generated by the grammar symbols on the continuation stack. The parser backtracks when that sum is larger than the length of the remaining input (which is called a *shaper test* [Kun65]).

A more complicated solution is to call nonterminals with an extra parameter which indicates how many symbols it should read from the input. When a rule is applied, this amount from the extra parameter of the lhs of the rule is nondeterministically partitioned into smaller amounts with which the members in the rhs are called. The parser backtracks if a nonterminal is called recursively with the same parameter value at the same input position [She76].

Instead of choosing nondeterministically how a value in the lhs is partitioned into smaller values for the members in the rhs, we may also apply an approach which allows this partition to be more selective. For example, we may employ an extra set of recognition procedures which indicate which of the possible partitions of the value in a lhs may allow a successful parse. No checks are performed to see whether nonterminals are called recursively with the same parameters. This is a very free interpretation of an algorithm in [Fro92].

Each of these three approaches yields correct parsers for cycle-free grammars. (The second approach also yields terminating parsers for arbitrary grammars.) None of these solutions is very practical however, the first two for efficiency reasons, especially for long input, and the third because it requires an extra set of recognition procedures.

We mention a fourth idea from the existing literature, realized as algorithm ET^* (*extension tables*) in [Die87]. This algorithm is closely related to our new kind of parsing, and in order to be able to make a comparison, we defer discussion of algorithm ET^* to Section 5.7.

5.2.3 Left-corner parsing

Contrary to TD recognizers, left-corner recognizers terminate for plain left-recursive grammars, such as G_1. LC recognition can be defined using the transformation below from a context-free grammar to a DCG. It is similar to the *goal-corner transformation* in [LRS76].

In order to avoid useless rules we need the set $GOAL$ consisting of S and all nonterminals which occur as i-th member in a rhs, where $i > 1$. This set is formally defined by

$$GOAL \ = \ \{S\} \cup \{A \mid B \rightarrow \alpha, A, \beta. \in P \wedge \alpha \neq \epsilon\}$$

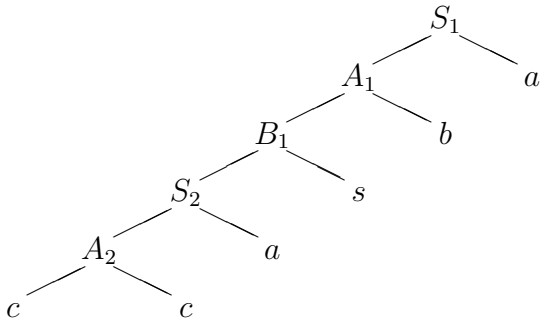


Figure 5.1: Parse tree for grammar G_1

Definition 5.2.1 *If $G = (T, N, P, S)$ then $LC(G)$ is the DCG with the same start symbol S and the rules below. The nonterminals of the DCG are a subset of $N \cup \{A' \mid A \in N\} \cup \{goal\}$. The nonterminals of the form A' or "goal" have one argument, which is the name of a nonterminal from GOAL.*

1. $A \rightarrow goal(A).$
   for all $A \in GOAL$

2. $goal(g) \rightarrow \{B \mathrel{\angle^*} g\},\ \beta,\ B'(g).$
   for all $B \rightarrow \beta. \in P$ such that β begins with a terminal or is ϵ

3. $C'(g) \rightarrow \{B \mathrel{\angle^*} g\},\ \beta,\ B'(g).$
   for all $B \rightarrow C, \beta. \in P$

4. $A'(A) \rightarrow \epsilon.$
   for all $A \in GOAL$

The use of the left-corner relation $\angle^*$, which avoids construction of subderivations which cannot fulfil the goal, is called *top-down filtering*.

For the simple grammar G_1, $GOAL$ has only one member, viz. S, and $LC(G_1)$ is the recognizer given by

$$\begin{aligned}
S &\rightarrow goal(S).\\
goal(g) &\rightarrow \{A \mathrel{\angle^*} g\},\ [c],\ [c],\ A'(g).\\
A'(g) &\rightarrow \{S \mathrel{\angle^*} g\},\ [a],\ S'(g).\\
B'(g) &\rightarrow \{A \mathrel{\angle^*} g\},\ [b],\ A'(g).\\
S'(g) &\rightarrow \{B \mathrel{\angle^*} g\},\ [s],\ B'(g).\\
S'(S) &\rightarrow \epsilon.
\end{aligned}$$

We give some hints which may help the reader to understand LC parsing. As an example, Figure 5.1 shows the only parse tree for the input *ccasba*. The labels of some nodes have been subscripted to simplify the discussion.

The LC recognizer recognizes the maximal spine $S_1 - A_1 - B_1 - S_2 - A_2 - c$ from bottom to top. The procedural meaning of a call such as $A'(g)$ can be informally conveyed as "Given the fact that an A has already been recognized, now recognize the rest of a g", where variable g holds for example S. (See also Chapters 2 and 3.)

A variant of LC parsing is discussed in Section 5.8.

5.2.4 Cancellation parsing

In this section we show how we can construct a *cancellation recognizer* by translating grammar rules into DCG rules, in the same way that we construct TD recognizers. Cancellation recognizers can however also deal with left recursion.

To show how cancellation recognition works, we return to the TD recognizer and the grammar from Section 5.2.2. Because of the left recursion, nonterminals are in danger of calling each other without any input being read.

The first attempt to solve this is to give each DCG nonterminal an argument which is a set of nonterminals, the so called *cancellation set*, which is initially empty. The idea is that every time a new nonterminal is encountered, this nonterminal is added to the cancellation set and the resulting set is passed on to the leftmost member of the next rule instance. If a nonterminal is to be recognized which is already in the cancellation set then the parser fails and backtracks.

In the part of a spine that is recognized in this way, each nonterminal now occurs at most once, which is a rather blunt way to force termination. At this point we have the recognizer for G_1 below. The DCG has a new start symbol S'. In this DCG the infix predicate $\_ \notin \_$ occurs, which has the obvious semantics with respect to the mixfix functor $\_ \cup \{\_\}$ and the nullary functor $\emptyset$.

$$\begin{array}{rcl} S' & \rightarrow & S(\emptyset). \\ S(x) & \rightarrow & \{S \notin x\},\ A(x \cup \{S\}),\ [a]. \\ A(x) & \rightarrow & \{A \notin x\},\ B(x \cup \{A\}),\ [b]. \\ A(x) & \rightarrow & \{A \notin x\},\ [c],\ [c]. \\ B(x) & \rightarrow & \{B \notin x\},\ S(x \cup \{B\}),\ [s]. \end{array}$$

A parsing algorithm similar to the one suggested above has been proposed in [Lee91]. Regrettably this is not the final solution to our problem. Because we restricted the top-down parsing as we did, we can no longer recognize for example the input *ccasba*. The recognizer succeeds in parsing a prefix of this, viz. *cca*, and recognizes the subtree of the parse tree in Figure 5.1 with the root S_2. The part between S_1 and S_2 cannot be recognized however, because at the time that top-down parsing from S_1 would arrive at S_2, S is already in the cancellation set which has been passed on to the call of S which is supposed to connect B_1 with S_2.

We solve this by having S put a new kind of terminal $\overline{S}$ in front of the remaining input, indicating that an S has been recognized, and then having S call itself. A new rule $S(x) \rightarrow [\overline{S}\,]$. is added to represent the fact that if an S has been recognized and an S is sought, then we are done.

We assume that the bar symbol $^-$ is available as a unary Prolog functor.

We further assume that a predicate *untoken* is available which puts its single argument in front of the remaining input. Including the (otherwise implicit) string arguments, its definition is the unit clause

$$untoken(n, i, [n|i]).$$

We now have the following recognizer.

$$S' \rightarrow S(\emptyset).$$

$$
\begin{array}{lcl}
S(x) & \rightarrow & \{S \notin x\},\ A(x \cup \{S\}),\ [a],\ \mathit{untoken}(\overline{S}),\ S(x). \\
A(x) & \rightarrow & \{A \notin x\},\ B(x \cup \{A\}),\ [b],\ \mathit{untoken}(\overline{A}),\ A(x). \\
A(x) & \rightarrow & \{A \notin x\},\ [c],\ [c],\ \mathit{untoken}(\overline{A}),\ A(x). \\
B(x) & \rightarrow & \{B \notin x\},\ S(x \cup \{B\}),\ [s],\ \mathit{untoken}(\overline{B}),\ B(x). \\
S(x) & \rightarrow & [\overline{S}\,]. \\
A(x) & \rightarrow & [\overline{A}\,]. \\
B(x) & \rightarrow & [\overline{B}\,].
\end{array}
$$

The extensions we made to the recognizer above allow the recognition of the complete spine of Figure 5.1: first the subtree with root S_2 is recognized. Then an $\overline{S}$ is pushed and S calls itself. Thereupon the part of the spine between S_1 and S_2 is recognized.

The grammar we used above only gives rise to one nontrivial maximal spine. In general we must make sure that for each maximal spine consisting of more than one node a separate cancellation set is introduced, since more than one spine may be under construction at the same time. In the top node of the spine the set is empty and when more nodes of the spine are recognized more nonterminals are added to it.

The grammar above also did not contain epsilon rules. If such rules do occur in a grammar it must be ensured that in the parser possible barred input symbols resulting from an *untoken* are not ignored, because these indicate an already recognized part of the spine. Therefore we assume that a predicate *notbarred* is available which fails if the first symbol of the remaining input is of the form $\overline{A}$, where A is some nonterminal, and succeeds otherwise. The predicate *notbarred* is defined by the clause

$$\mathit{notbarred}(i, i) :- \mathit{not}(i = [\overline{y}|i']).$$

where the predicate *not* succeeds only if its argument fails.

In full generality, a cancellation recognizer is now to be constructed according to the definition below.

Definition 5.2.2 *If $G = (T, N, P, S)$ then $C(G)$ is the DCG with the new start symbol S' and the following rules.*

1. $S' \rightarrow S(\emptyset)$.
2. $A(x) \rightarrow \{A \notin x\},\ B_1(x \cup \{A\}),\ v_1,\ B_2(\emptyset),\ v_2,\ \ldots,\ B_n(\emptyset),\ v_n,\ \mathit{untoken}(\overline{A}),\ A(x).$
   for all $A \rightarrow B_1, v_1, B_2, v_2, \ldots, B_n, v_n. \in P$, *where* $n > 0$
3. $A(x) \rightarrow \{A \notin x\},\ v_1,\ B_2(\emptyset),\ v_2,\ \ldots,\ B_n(\emptyset),\ v_n,\ \mathit{untoken}(\overline{A}),\ A(x).$
   for all $A \rightarrow v_1, B_2, v_2, \ldots, B_n, v_n. \in P$, *where* $n > 0$ *and* $v_1 \neq \epsilon$
4. $A(x) \rightarrow \{A \notin x\},\ \mathit{notbarred},\ \mathit{untoken}(\overline{A}),\ A(x).$
   for all $A \rightarrow \epsilon. \in P$
5. $A(x) \rightarrow [\overline{A}\,].$
   for all $A \in N$

Note that if a barred nonterminal occurs in the remaining input, it is the only one, and it occurs as first symbol of the remaining input.

Termination is discussed in Section 5.5.

5.2.5 Cancellation recognizers as context-free grammars

Any DCG can be transformed into a context-free grammar provided the arguments of the DCG range over a finite domain. Because the number of possible cancellation sets in $C(G)$ is finite for any G, we can also transform $C(G)$ into a context-free grammar, which we will denote as $C_{CF}(G)$.

This transformation can be performed as follows: each argument in $C(G)$ is consistently substituted with every possible cancellation set it can be successfully instantiated with. The conditions of the form $\{A \notin X\}$ are evaluated and a rule is eliminated if this condition does not hold (X denotes a constant set).

Whether a barred nonterminal has been pushed in front of the remaining input, and if so which one it is, is encoded as an extra argument of nonterminals.

To make sure that $C_{CF}(G)$ is reduced we need the following definitions of the sets $CALL \subseteq N \times \mathcal{P}(N)$, $U \subseteq P \times \mathcal{P}(N)$, and $\overline{U} \subseteq P \times \overline{N} \times \mathcal{P}(N)$, where $\overline{N}$ is defined to be $\{\overline{A} | A \in N\}$.

We define $CALL$ to contain all pairs (A, X) such that nonterminal A may be called with cancellation set X. Formally, for a nonterminal A and a sets of nonterminals X we define

$$(A, X) \in CALL$$
if and only if
$A_0 \rightarrow_{lc} A_1\alpha_0 \rightarrow_{lc} \ldots \rightarrow_{lc} A_{n-1}\alpha_{n-2}\ldots\alpha_0 \rightarrow_{lc} A_n\alpha_{n-1}\ldots\alpha_0$, where $n \geq 0$, $A_0 \in GOAL$, $|\{A_0, \ldots, A_{n-1}\}| = n$, $X = \{A_0, \ldots, A_{n-1}\}$, and $A = A_n$.

The informal meaning of the clause $|\{A_0, \ldots, A_{n-1}\}| = n$ is that $A_0, \ldots, A_{n-1}$ are all different.

We now have that $(A, X) \in CALL$ if and only if A may be called with cancellation set X.

For a rule $A \rightarrow \alpha.$ and a set of nonterminals X we define

$$((A \rightarrow \alpha.), X) \in U$$
if and only if
$A_0 \rightarrow_{lc} A_1\alpha_0 \rightarrow_{lc} \ldots \rightarrow_{lc} A_n\alpha_{n-1}\ldots\alpha_0 \rightarrow_{l} \alpha_n\alpha_{n-1}\ldots\alpha_0$, where $n \geq 0$, $A_0 \in GOAL$, α_n begins with a terminal or is ϵ, $|\{A_0, \ldots, A_n\}| = n + 1$, and where for some i we have $X = \{A_0, \ldots, A_{i-1}\}$, $A = A_i$, and $\alpha = A_{i+1}\alpha_i$ if $i < n$ and $\alpha = \alpha_n$ if $i = n$.

The intuition behind this definition can be explained as follows. If a cancellation recognizer begins with the recognition of a spine from nonterminal A_0, it first begins to work top-down, discovering only nonterminals it has not seen before in that spine. If $((A \rightarrow \alpha.), X) \in U$ then we have that during this process the nonterminal A is called with parameter X and that recognition may proceed successfully by taking the alternative corresponding to the rule $A \rightarrow \alpha$.

We define

$$((A \rightarrow \alpha.), \overline{B}, X) \in \overline{U}$$

if and only if
$A_0 \rightarrow_{lc} A_1\alpha_0 \rightarrow_{lc} \ldots \rightarrow_{lc} A_n\alpha_{n-1}\ldots\alpha_0 \rightarrow_{lc} A_{n+1}\alpha_n\alpha_{n-1}\ldots\alpha_0$,
where $n \geq 0$, $A_0 \in GOAL$, $|\{A_0, \ldots, A_n\}| = n+1$, and where for some i and j with $i \leq j \leq n$, we have $B = A_i = A_{n+1}$, $X = \{A_0, \ldots, A_{j-1}\}$, $A = A_j$, and $\alpha = A_{j+1}\alpha_j$.

Again we give a short explanation of the above definition. When a cancellation recognizer has recognized the complete subderivation below a node labeled B in a spine from A_0, then it puts a barred nonterminal $\overline{B}$ in front of the remaining input. It then starts working top-down again from $A_i = B$, in such a way that only nonterminals are to be processed which it has not seen before in the spine (not including the nonterminals it found after it found B for the first time). It will then find $A_{i+1}, \ldots, A_n$. If $((A \rightarrow \alpha.), \overline{B}, X) \in \overline{U}$ then we have that during this process nonterminal A is called with parameter X and that recognition may proceed successfully by taking the alternative corresponding to the rule $A \rightarrow \alpha$.

It can easily be verified that

$$\begin{aligned} CALL \;=\; & \{(A,X) \mid \exists_{A\rightarrow\alpha.\in P}[((A \rightarrow \alpha.), X) \in U]\} \cup \\ & \{(A,X) \mid \exists_{A\rightarrow\alpha.\in P}\exists_{C\in N}[((A \rightarrow \alpha.), \overline{C}, X) \in \overline{U}]\} \cup \\ & \{(A,X) \mid \exists_{B\rightarrow A,\alpha.\in P}\exists_{Y\subseteq N}[((B \rightarrow A,\alpha.), \overline{A}, Y) \in \overline{U} \wedge X = Y \cup \{B\}]\} \end{aligned}$$

under the implicit assumption that the grammar is reduced.

We now have

Definition 5.2.3 *If $G = (T, N, P, S)$ then $C_{CF}(G)$ is the DCG with the new start symbol $S(\emptyset)$ and the following rules.*

1. $A(X) \rightarrow B_1(Y),\ v_1,\ B_2(\emptyset),\ v_2,\ \ldots,\ B_n(\emptyset),\ v_n,\ A(X, \overline{A})$.
   for all $A \rightarrow B_1, v_1, B_2, v_2, \ldots, B_n, v_n. \in P$, where $n > 0$, and X and Y such that $((A \rightarrow B_1, v_1, \ldots, B_n, v_n.), X) \in U$ and $Y = X \cup \{A\}$

2. $A(X) \rightarrow v_1,\ B_2(\emptyset),\ v_2,\ \ldots,\ B_n(\emptyset),\ v_n,\ A(X, \overline{A})$.
   for all $A \rightarrow v_1, B_2, v_2, \ldots, B_n, v_n. \in P$, where $n > 0$, and $v_1 \neq \epsilon$, and X such that $((A \rightarrow v_1, B_2, v_2, \ldots, B_n, v_n.), X) \in U$

3. $A(X) \rightarrow A(X, \overline{A})$.
   for all $A \rightarrow \epsilon. \in P$ and X such that $((A \rightarrow \epsilon.), X) \in U$

4. $A(X, \overline{C}) \rightarrow B_1(Y, \overline{C}),\ v_1,\ B_2(\emptyset),\ v_2,\ \ldots,\ B_n(\emptyset),\ v_n,\ A(X, \overline{A})$.
   for all $A \rightarrow B_1, v_1, B_2, v_2, \ldots, B_n, v_n. \in P$, where $n > 0$, and X and Y such that $((A \rightarrow B_1, v_1, \ldots, B_n, v_n.), \overline{C}, X) \in \overline{U}$ and $Y = X \cup \{A\}$

5. $A(X, \overline{A}) \rightarrow \epsilon$.
   for all A and X such that $(A, X) \in CALL$

Note that an expression of the form $A(X)$ or $A(X, \overline{C})$, where $A, C \in N$ and $X \subseteq N$, can be interpreted as the name of a single nonterminal without parameters, so that $C_{CF}(G)$ can be seen as a context-free grammar. The nonterminals of the form $A(X, \overline{C})$ play the rôle of $A(X)$ in $C(G)$ when $\overline{C}$ has been pushed in front of the remaining input.

Example 5.2.1 For G_1 we have

$$\begin{aligned} CALL &= \{(S,\emptyset),\ (A,\{S\}),\ (B,\{S,A\}),\ (S,\{S,A,B\})\} \\ U &= \{((S \to A,[a].),\emptyset),\ ((A \to [c],[c].),\{S\})\} \\ \overline{U} &= \{((S \to A,[a].),\overline{S},\emptyset),\ ((A \to B,[b].),\overline{S},\{S\}),\ ((B \to S,[s].),\overline{S},\{S,A\})\} \end{aligned}$$

From these facts we derive that the recognizer $C_{CF}(G_1)$ is defined by

$$\begin{aligned} S(\emptyset) &\to A(\{S\}),\ [a],\ S(\emptyset,\overline{S}). \\ A(\{S\}) &\to [c],\ [c],\ A(\{S\},\overline{A}). \\ S(\emptyset,\overline{S}) &\to A(\{S\},\overline{S}),\ [a],\ S(\emptyset,\overline{S}). \\ A(\{S\},\overline{S}) &\to B(\{S,A\},\overline{S}),\ [b],\ A(\{S\},\overline{A}). \\ B(\{S,A\},\overline{S}) &\to S(\{S,A,B\},\overline{S}),\ [s],\ B(\{S,A\},\overline{B}). \\ S(\emptyset,\overline{S}) &\to \epsilon. \\ A(\{S\},\overline{A}) &\to \epsilon. \\ B(\{S,A\},\overline{B}) &\to \epsilon. \\ S(\{S,A,B\},\overline{S}) &\to \epsilon. \end{aligned}$$

Note that we may consider this to be a context-free grammar, since there are no variables.

An obvious optimization is to perform substitution wherever there is only one rule which may be applied, and then to remove useless rules.

For the above example we then obtain

$$\begin{aligned} S(\emptyset) &\to [c],\ [c],\ [a],\ S(\emptyset,\overline{S}). \\ S(\emptyset,\overline{S}) &\to [s],\ [b],\ [a],\ S(\emptyset,\overline{S}). \\ S(\emptyset,\overline{S}) &\to \epsilon. \end{aligned}$$

□

We see from the previous example that C_{CF} gives rise to an unusual grammar transformation.

5.2.6 From recognizers to parsers

If we want parsers which generate parse trees while recognizing the input, then we can make the following extension to C: each nonterminal has one extra argument, which represents a fragment of the parse tree constructed by that nonterminal. Barred nonterminals are pushed onto and popped from the remaining input together with those fragments of parse trees.

In general, a cancellation parser is constructed by the following definition.

Definition 5.2.4 *If $G = (T, N, P, S)$ then $C_{tree}(G)$ is the DCG with the new start symbol S' and the following rules.*

1. $S'(t) \to S(\emptyset, t)$.

2. $A(x,t) \to \{A \notin x\},\ B_1(x \cup \{A\}, t_1),\ v_1,\ B_2(\emptyset, t_2),\ v_2,\ \ldots,\ B_n(\emptyset, t_n),\ v_n,$
   $untoken(\overline{A}(A(t_1, v_1, \ldots, t_n, v_n))),\ A(x,t).$
   for all $A \to B_1, v_1, B_2, v_2, \ldots, B_n, v_n. \in P$, *where* $n > 0$

3. $A(x,t) \to \{A \notin x\},\ v_1,\ B_2(\emptyset, t_2),\ v_2,\ \ldots,\ B_n(\emptyset, t_n),\ v_n,$
   $untoken(\overline{A}(A(v_1, t_2, \ldots, t_n, v_n))),\ A(x,t).$
   for all $A \to v_1, B_2, v_2, \ldots, B_n, v_n. \in P$, *where* $n > 0$ *and* $v_1 \neq \epsilon$

4. $A(x,t) \to \{A \notin x\},\ notbarred,\ untoken(\overline{A}(A(\epsilon))),\ A(x,t).$
   for all $A \to \epsilon. \in P$

5. $A(x,t) \to [\overline{A}(t)].$
   for all $A \in N$

A member $[\overline{A}(t)]$ in a DCG rule is translated into Prolog by the call $token(\overline{A}(t), i, i')$, where i and i' are the appropriate string arguments.

For some nonterminals A we may have that for any cancellation set x, the call $A(x,t)$ after a call $untoken(\overline{A}(A(\ldots)))$ may only succeed by applying the rule $A(x,t) \to [\overline{A}(t)]$. The holds precisely for the nonterminals in Z, where Z is defined by

$$Z = \{A \mid \neg\exists_{A \to \alpha. \in P} \exists_{X \subseteq N} [((A \to \alpha.), \overline{A}, X) \in \overline{U}]\}$$

Note that nonterminals which are not left-recursive are elements of Z, but Z may also contain many left-recursive nonterminals, as shown below in Example 5.2.2.

The above observation leads to an obvious optimization of C_{tree}: for nonterminals which are in Z, the next two clauses may be used instead of the second and third clause of C_{tree}.[2]

2b. $A(x, A(t_1, v_1, \ldots, t_n, v_n)) \to \{A \notin x\},$
   $B_1(x \cup \{A\}, t_1),\ v_1,\ B_2(\emptyset, t_2),\ v_2,\ \ldots,\ B_n(\emptyset, t_n),\ v_n.$
   for all $A \to B_1, v_1, B_2, v_2, \ldots, B_n, v_n. \in P$, *where* $n > 0$

3b. $A(x, A(v_1, t_2, \ldots, t_n, v_n)) \to \{A \notin x\},\ v_1,\ B_2(\emptyset, t_2),\ v_2,\ \ldots,\ B_n(\emptyset, t_n),\ v_n.$
   for all $A \to v_1, B_2, v_2, \ldots, B_n, v_n. \in P$, *where* $n > 0$ *and* $v_1 \neq \epsilon$

The rules resulting from these clauses may be seen as resulting from the original Clauses *2.* and *3.* after substitution of rules resulting from Clause *5.*

Example 5.2.2 For G_1 we have $Z = \{A, B\}$. The optimized parser for G_1 now becomes

$$\begin{array}{rcl} S'(t) & \to & S(\emptyset, t) \\ S(x,t) & \to & \{S \notin x\},\ A(x \cup \{S\}, t_A),\ [a],\ untoken(\overline{S}(S(t_A, 'a'))),\ S(x,t). \\ A(x, A(t_B, 'b')) & \to & \{A \notin x\},\ B(x \cup \{A\}, t_B),\ [b]. \\ A(x, A('c', 'c')) & \to & \{A \notin x\},\ [c],\ [c]. \\ B(x, B(t_S, 's')) & \to & \{B \notin x\},\ S(x \cup \{B\}, t_S),\ [s]. \\ S(x,t) & \to & [\overline{S}(t)]. \end{array}$$

□

[2]In *2b.* and *3b.* we may also omit the condition $\{A \notin x\}$, since this condition always holds if $A \in Z$. Correspondingly, we may in *2b.* omit adding A to the cancellation set which is passed on to B_1.

The gain from the above optimization does not lie in the construction of parse trees through canonical DCG arguments but lies in the evaluation of arbitrary arguments which are explicitly present in the source grammar G.

Grammars for natural languages and programming languages may have arguments for roughly two reasons:

1. to allow *explicit* description of how parsing a sentence should yield a value reflecting the structure of the sentence (unlike the canonical generation of parse trees as achieved by $C_{tree}(G)$); or

2. to restrict the language generated by the context-free part of the grammar. Derivations are rejected by failure of unification.

Because the early rejection of derivations saves the parser much work, it is valuable to evaluate at an early stage the arguments which are used for the purpose of restricting the language.

The optimization of C_{tree} above suggests how top-to-bottom flow of argument values can be achieved for nonterminals in Z. We only give an example, instead of showing the full construction of a cancellation parser. The interested reader may try to formulate the general construction inspired by C_{tree} and its optimization.

Example 5.2.3 Let the grammar G_2 be defined by the following rules

$$\begin{array}{rcl} S & \rightarrow & A(1),\ [s]. \\ A(y) & \rightarrow & B(y),\ [a]. \\ B(1) & \rightarrow & [b]. \\ B(2) & \rightarrow & [b],\ S. \end{array}$$

The unoptimized construction of the parser, without using the knowledge that $S, A, B \in Z$, leads to

$$\begin{array}{rcl} S' & \rightarrow & S(\emptyset). \\ S(x) & \rightarrow & \{S \notin x\},\ A(x \cup \{S\}, 1),\ [s],\ \mathit{untoken}(\overline{S}),\ S(x). \\ A(x, y_A) & \rightarrow & \{A \notin x\},\ B(x \cup \{A\}, y_B),\ [a],\ \mathit{untoken}(\overline{A}(y_B)),\ A(x, y_A). \\ B(x, y) & \rightarrow & \{B \notin x\},\ [b],\ \mathit{untoken}(\overline{B}(1)),\ B(x, y). \\ B(x, y) & \rightarrow & \{B \notin x\},\ [b],\ S,\ \mathit{untoken}(\overline{B}(2)),\ B(x, y). \\ S(x) & \rightarrow & [\overline{S}\,]. \\ A(x, y) & \rightarrow & [\overline{A}(y)]. \\ B(x, y) & \rightarrow & [\overline{B}(y)]. \end{array}$$

In this parser, B will be called with an uninstantiated second argument, and a deterministic choice between the alternatives cannot be made. The optimization suggested above, which makes use of the fact that $S, A, B \in Z$, leads to a slightly different parser:

$$\begin{array}{rcl} S' & \rightarrow & S(\emptyset). \\ S(x) & \rightarrow & \{S \notin x\},\ A(x \cup \{S\}, 1),\ [s]. \end{array}$$

$$\begin{array}{lcl} A(x,y) & \rightarrow & \{A \notin x\},\ B(x \cup \{A\}, y),\ [a]. \\ B(x,1) & \rightarrow & \{B \notin x\},\ [b]. \\ B(x,2) & \rightarrow & \{B \notin x\},\ [b],\ S. \end{array}$$

The advantage of this parser is that B will be called with an instantiated second argument (in this case always 1), which allows a deterministic choice for the correct alternative (in this case always the first one). Therefore, this parser is more efficient. □

5.3 Correctness of cancellation parsing

With C_{tree} we can easily prove the correctness of cancellation parsing. In order to prove correctness of a backtracking parsing algorithm, two things must be proved:

- For every parse tree there is a unique search path (i.e. a sequence of actions of the parser not involving backtracking) in which the parser produces that parse tree while recognizing the string which is its yield.
- No parse trees are produced which do not derive the input string.

For cancellation parsing, the second of these two points is trivial. We therefore only give the proof of the first point.

An important goal is to prove the following.

> Assume that we have a tree T and a set X such that
>
> - the root of T is labelled A,
> - the nonterminals in the spine from the root of T are not in X, and
> - T has yield w.
>
> Then a call of $A(X,t)$ on input w yields tree T in a unique way in variable t.

After we have proved this statement, the correctness of the parsing algorithm follows by taking A to be the start symbol S and X to be $\emptyset$.

We will however need to simultaneously prove a second statement, where the notion of *tree* in the normal sense of *parse tree* is extended to include trees in which the lowest node in the spine from the root may be labelled with a barred nonterminal. Such a barred nonterminal $\overline{B}$ is then associated with some appropriate subtree of which the root is labelled B.

Our second goal is now to prove the following.

> Assume that we have a tree T and a set X such that
>
> - the root of T is labelled A,
> - the left-most leaf is labelled with a barred nonterminal, associated with some tree T',
> - the nonterminals in the spine from the root of T (not including the barred nonterminal in the left-most leaf) are not in X, and

- T has yield w.

(We also allow a trivial tree T of which the root, which is then the only node, is labelled with a barred nonterminal $\overline{A}$.) Then a call of $A(X,t)$ on input w yields tree T'' in a unique way in variable t, where T'' results from replacing the left-most leaf of T by T'.

We will prove the above two statements simultaneously, using induction on the size of the tree T.

Assume now that we have a tree T and a set X such that

- the root of T is labelled A,
- the nonterminals in the spine from the root of T (not including a possible barred nonterminal in the left-most leaf) are not in X, and
- T has yield w.

We investigate what happens upon a call of $A(X,t)$. For this, we distinguish between the following cases.

1. Suppose that T consists of a single node labelled with a barred nonterminal $\overline{A}$. Now we are immediately done because there is exactly one parser rule which is appropriate and which is $A(x,t) \rightarrow [\overline{A}(t)]$., the application of which obviously yields the required tree in variable t.

2. Otherwise, let A_0 be the lowest node in the spine from the root of T (excluding the lowest node in the spine) labelled with A. At A_0 a grammar rule is used of the form $A \rightarrow v_1, B_2, v_2, \ldots, B_n, v_n.$, where $n > 0$ and $v_1 \neq \epsilon$, of the form $A \rightarrow \epsilon.$, or of the form $A \rightarrow B_1, v_1, B_2, v_2, \ldots, B_n, v_n.$, where $n > 0$. We leave the first two possibilities as an exercise to the reader and concentrate on the last possibility.

   The only rule in the parser which is appropriate in this situation is

$$\begin{aligned} A(x,t) \;\rightarrow\; & \{A \notin x\},\ B_1(x \cup \{A\}, t_1),\ v_1,\ B_2(\emptyset, t_2),\ v_2,\ \ldots,\ B_n(\emptyset, t_n),\ v_n, \\ & \mathit{untoken}(\overline{A}(A(t_1, v_1, \ldots, t_n, v_n))),\ A(x,t). \end{aligned}$$

   No parser rule can be applied which recognizes an occurrence higher in the spine. This is because parser rules of this form add A to the cancellation set which is passed down to other nonterminals recognizing lower parts of the spine. This would prevent A_0 from being recognized.

   From the induction hypothesis we know that the call $B_1(X \cup \{A\}, t_1)$ yields in t_1 in a unique way the subtree T_1 whose root is the leftmost son of A_0. (In the case that the left-most leaf of T_1 is a barred nonterminal associated with some tree T', this leaf has been replaced in t_1 by T'.) Note that nonterminals from X do not occur in the spine from the root of T_1 because of the assumption that they do not occur in the spine from the root of T. And also note that A does not occur in the spine from the root of T_1 because of the assumption that the root of T_1 is the leftmost son of A_0, which is the lowest node labelled A in the spine from the root of T.

From the induction hypothesis we know that a call of $B_i(\emptyset, t_i)$ yields in t_i in a unique way the subtree T_i of which the root is the i-th son of A_0 labelled with a nonterminal, where $2 \leq i \leq n$. After this has taken place, the barred nonterminal $\overline{A}$ is pushed in front of the remaining input, associated with the tree T''', which is constructed from $T_1, \ldots, T_n, v_1, \ldots, v_n$, and A as indicated in the rule above. Again we use the induction hypothesis to conclude that the subsequent call of $A(X, t)$ yields the required tree in a unique way in variable t.

This concludes the proof by induction.

Without proof, we state that nondeterministic cancellation parsing terminates on all input for the same class of grammars as nondeterministic left-corner parsing does, viz. for the cycle-free grammars without hidden left recursion. Further discussion on termination can be found in Section 5.5.

5.4 Deterministic parsing

All backtracking parsing algorithms are exponential (in the length of the input) for some grammar. The time complexity may be reduced by applying *lookahead*: investigating the next k symbols of the remaining input, for some fixed $k > 0$, may provide enough information to reject some of the alternatives before they are even tried. In some cases where lookahead is used, the parsing algorithm may even be deterministic, i.e. a choice between alternatives can be uniquely determined. Deterministic parsing algorithms have a linear time complexity.

In this section we will apply lookahead by evaluating certain predicates which are added in front of the actual alternatives of the parser.

Predicates that we will often need are the predicates $\mathit{lookahead}_k$, for $k > 0$, which succeed if and only if the remaining input truncated to length k is in the set which is their only argument. For $k = 1$, $\mathit{lookahead}_1$ is defined by the clauses

$$\begin{aligned} \mathit{lookahead}_1(f, [a|i], [a|i]) &:- [a] \in f. \\ \mathit{lookahead}_1(f, [\,], [\,]) &:- [\,] \in f. \end{aligned}$$

where we distinguish between the cases that the remaining input is empty or non-empty. The reader should not have any difficulty finding a general definition for $\mathit{lookahead}_k$.

If w is a string of terminals, then $k : w$ denotes w if the length of w is less than or equal to k and it denotes the prefix of w of length k, otherwise.

We define the operators $\_ \circ_k \_$ such that

$$I \circ_k J \;=\; \{k : vw \mid v \in I, w \in J\}$$

where I and J are sets of strings. $L_k(\alpha)$ denotes the set $\{k : x \mid \alpha \rightarrow^* x\}$, i.e. the set of strings that are generated by α, truncated to length k. Where the predicate $\_ = \_$ occurs we assume that it has obvious semantics.

We will first investigate two ways of using lookahead for top-down parsing, and subsequently we will show that there are three distinct ways of using lookahead for cancellation parsing.

5.4.1 Deterministic top-down parsing

There are two ways of using lookahead for TD parsing:

1. An LL(k) parser investigates the first k symbols of the remaining input and the strings (truncated to length k) of which it has been established *dynamically* that they may follow the current nonterminal instance.

2. A strong LL(k) also investigates the first k symbols of the remaining input, but it does not compute dynamically which strings may follow nonterminal instances. Instead it makes use of the strings (again truncated to length k) which may follow the nonterminals in *any* derivation.

The latter obviously leads to the less precise way of using lookahead, because it does not distinguish between individual instances of the same nonterminal.

LL(k) parsers are given by the following construction.

Definition 5.4.1 *If $G = (T, N, P, S)$ then $TD_k(G)$ is the DCG with the new start symbol S' and the following rules.*

1. $S' \rightarrow S(\{\epsilon\})$.

2. $A(f) \rightarrow \{f_0 = F_0 \circ_k f\},\ lookahead_k(f_0),\ v_0,$
   $\{f_1 = F_1 \circ_k f\},\ B_1(f_1),\ v_1,$
   $\{f_2 = F_2 \circ_k f\},\ B_2(f_2),\ v_2,$
   $\ldots,$
   $\{f_n = F_n \circ_k f\},\ B_n(f_n),\ v_n.$
   for all $A \rightarrow v_0, B_1, v_1, \ldots, B_n, v_n. \in P$, where $n \geq 0$ and F_i are the constant sets $L_k(v_i B_{i+1} v_{i+1} \ldots B_n v_n)$

A parser is called *deterministic* if for every call of some DCG nonterminal A we have that in at most one alternative of A the predicate $lookahead_k$ succeeds on its argument. Grammars for which the above parsers are deterministic are said to be LL(k).

We now investigate a less refined form of top-down parsing, for which we need to compute the (truncated) terminal strings which may follow a nonterminal in any sentential form: for every nonterminal $A \in N$ we define

$$FOLLOW_k(A) \quad = \quad \{k : v \mid \exists_w [S \rightarrow^* wAv]\}$$

The strong LL(k) parsers are now given by the following construction.

Definition 5.4.2 *If $G = (T, N, P, S)$ then $STD_k(G)$ is the DCG with the following rules.*

1. $A \rightarrow lookahead_k(F),\ \alpha.$
   for all $A \rightarrow \alpha. \in P$, where F is the constant set $L_k(\alpha) \circ_k FOLLOW_k(A)$

Grammars for which the above parsers are deterministic are said to be *strong* LL(k).

In a similar way we also have two kinds of left-corner parsers which are deterministic for the LC(k) and *strong* LC(k) grammars, respectively. A more elaborate discussion of these parsers can be found in [RL70].

5.4.2 Deterministic cancellation parsing

We now discuss three kinds of cancellation parsers which use lookahead. Each of the three investigates the first k symbols of the remaining input. The difference between the three kinds of using lookahead can be given informally by the following.

1. An C(k) parser investigates the strings (truncated to length k) of which it has been established *dynamically* that they may follow the current nonterminal instance.

2. A strong C(k) parser may investigate the cancellation set of the called nonterminal and uses the strings (again truncated to length k) which may follow the nonterminals in *any* derivation where the current cancellation set is operative.

3. A severe C(k) parser may not investigate the cancellation set of the called nonterminal. It makes use of the (truncated) strings which may follow a nonterminal in *any* sentential derivation where *any* cancellation set is operative.

To define these kinds of parsers we first need some notation to refer to derivations which are special for cancellation parsing. We let $vA\alpha \overset{X}{\rightarrow}{}^*_l\ vB\beta\alpha$ denote a derivation $vA\alpha \rightarrow^*_l vB\beta\alpha$ where all rules used have a lhs which is not in cancellation set X.

We let $wA\alpha \overset{X}{\rightarrow}{}^*_{sp}\ w\delta$ denote a derivation $wA\alpha \overset{X}{\rightarrow}{}^*_{lc}\ wB\beta\alpha \overset{X}{\rightarrow}_l\ w\gamma\beta\alpha = w\delta$, where γ begins with a terminal or is ϵ; i.e. a derivation where a spine from A is recognized without any nonterminals from X.

Similar to $L_k(A)$ we also define $L_k(A, X)$ where X is a cancellation set. To be able to deal with the barred nonterminals, which can occur as first symbol of the remaining input, $L_k(A, X)$ also contains strings beginning with a barred nonterminal. The amount of lookahead k only applies to the symbols from T however. We define $L_k(A, X)$ to be $\{k : v \mid A \overset{X}{\rightarrow}{}^*_{sp}\ \alpha \rightarrow^* v\} \cup \{\overline{B}(k : v) \mid A \overset{X}{\rightarrow}{}^*_{lc}\ B\bullet\alpha \rightarrow^* w\alpha \rightarrow^* wv\}$.[3] We have that $L_k(A, X)$ is the set of strings (truncated) that may be read by a call of $A(X)$ in $C(G)$.

We define the set $REC_k(A, X)$ to be $\{k : v \mid A \overset{X}{\rightarrow}{}^*_{lc}\ A\bullet\alpha \rightarrow^* w\alpha \rightarrow^* wv\}$ which is the set of terminal strings (truncated) that may be read when a DCG nonterminal calls itself at the end of an alternative after a barred nonterminal has been pushed.

We refine the definition of $lookahead_k$ such that $lookahead_k(F)$ succeeds if and only if either

- the first symbol of the remaining input is not a barred nonterminal and the remaining input truncated to length k is in F; or

- the first symbol of the remaining input is a barred nonterminal and the remaining input truncated to length $k + 1$ is in F.

We also refine the definition of $\circ_k$ such that it takes into account that the strings in its first argument may begin with a barred nonterminal: we redefine $\circ_k$ so that

$$I \circ_k J \;=\; \{k : vw \mid v \in I \cap T^*, w \in J\} \cup \{\overline{A}(k : vw) \mid \overline{A}v \in I, w \in J\}$$

[3] If there is a barred nonterminal $\overline{B}$ in front of the remaining input, then a call of $L_k(A, X)$ is only performed when $B \in X$. We can therefore replace the second part of the definition by $\{\overline{B}(k : v) \mid A \overset{X}{\rightarrow}{}^*_{lc}\ B\bullet\alpha \rightarrow^* w\alpha \rightarrow^* wv \wedge B \in X\}$ without consequences for the validity of the algorithms in the sequel. The original definition is theoretically more useful however.

5.4.2.1 Cancellation parsing with lookahead k

A cancellation parser with lookahead k is defined by

Definition 5.4.3 *If $G = (T, N, P, S)$ then $C_k(G)$ is the DCG with the new start symbol S' and the following rules.*

1. $S' \rightarrow S(\emptyset, \{\epsilon\})$.

2. $A(x, f) \rightarrow \{A \notin x\},$
$\{f_0 = L_k(B_1, x \cup \{A\}) \circ_k F_1 \circ_k REC_k(A, x) \circ_k f\},\ lookahead_k(f_0),$
$\{f_1 = F_1 \circ_k REC_k(A, x) \circ_k f\},\ B_1(x \cup \{A\}, f_1),\ v_1,$
$\{f_2 = F_2 \circ_k REC_k(A, x) \circ_k f\},\ B_2(\emptyset, f_2),\ v_2,$
$\ldots,$
$\{f_n = F_n \circ_k REC_k(A, x) \circ_k f\},\ B_n(\emptyset, f_n),\ v_n,$
$untoken(\overline{A}),\ A(x, f).$
for all $A \rightarrow B_1, v_1, B_2, v_2, \ldots, B_n, v_n. \in P$, such that $n > 0$, and where F_i are the constant sets $L_k(v_i B_{i+1} v_{i+1} \ldots B_n v_n)$

3. $A(x, f) \rightarrow \{A \notin x\},$
$\{f_1 = F_1 \circ_k REC_k(A, x) \circ_k f\},\ lookahead_k(f_1),\ v_1,$
$\{f_2 = F_2 \circ_k REC_k(A, x) \circ_k f\},\ B_2(\emptyset, f_2),\ v_2,$
$\ldots,$
$\{f_n = F_n \circ_k REC_k(A, x) \circ_k f\},\ B_n(\emptyset, f_n),\ v_n,$
$untoken(\overline{A}),\ A(x, f).$
for all $A \rightarrow v_1, B_2, v_2, \ldots, B_n, v_n. \in P$, such that $n = 1$ or $n > 1 \wedge v_1 \neq \epsilon$, and where F_i are the constant sets $L_k(v_i B_{i+1} v_{i+1} \ldots B_n v_n)$

4. $A(x, f) \rightarrow \{f_1 = \{\overline{A}\} \circ_k f\},\ lookahead_k(f_1),\ [\overline{A}].$
for all $A \in N$

Note that the use of the predicate $lookahead_k$ obviates the need for the predicate *notbarred* as in the fourth clause of Definition 5.2.2 where we handled epsilon rules. Therefore, in the above definition we were able to handle in a uniform way all rules whose right-hand sides do not begin with nonterminals.

A grammar G which is such that $C_k(G)$ is deterministic is said to be C(k). For a less operational definition of C(k) we need two more definitions.

For $A \in GOAL$ we let $wA\alpha \rightarrow^*_{can'(X)} wB\beta$ denote a derivation

$$
\begin{array}{rcccc}
wA\alpha = & wB_1\alpha_1 & \xrightarrow{X_0}{}^*_{lc} & wB_1\alpha'_1 & \xrightarrow{X_0}_{lc} \\
& wB_2\alpha_2 & \xrightarrow{X_1}{}^*_{lc} & wB_2\alpha'_2 & \xrightarrow{X_1}_{lc} \\
& wB_3\alpha_3 & \xrightarrow{X_2}{}^*_{lc} & wB_3\alpha'_3 & \xrightarrow{X_2}_{lc} \\
& \vdots & \vdots & \vdots & \\
& wB_{n-1}\alpha_{n-1} & \xrightarrow{X_{n-2}}{}^*_{lc} & wB_{n-1}\alpha'_{n-1} & \xrightarrow{X_{n-2}}_{lc} \\
& wB_n\alpha_n = wB\beta & & &
\end{array}
$$

where $n \geq 1$, $X_i = \{B_1, \ldots, B_i\}$, and $X = X_{n-1}$. We let $wA\alpha \rightarrow^*_{can(X)} wB\beta'$ denote a derivation $wA\alpha \rightarrow^*_{can'(X)} wB\beta \stackrel{X}{\rightarrow}^*_{lc} wB\beta'$.

Intuitively, if $wA\alpha \rightarrow^*_{can'(X)} wB\beta$ or $wA\alpha \rightarrow^*_{can(X)} wB\beta$ then there is a subtree t of some parse tree such that one of these derivations is the prefix of some leftmost derivation corresponding with t, where these derivations determine a part of the spine of t. Furthermore, when a cancellation parser is in the process of constructing t then B is called with cancellation set x. The second of the above derivations differs from the first in that also the case is covered where the parser pushes $\overline{B}$ on the stack and then calls B which then recognizes a non-empty part of the spine before reading that $\overline{B}$ from the input.

We can now also define the C(k) property in the following way.

Definition 5.4.4 *A grammar is said to be C(k), for some $k > 0$, if the following conditions hold.*

1. *The conditions*

   (a) $S \rightarrow^*_l w\underline{A}\alpha_1 \rightarrow^*_{can(X)} wB\beta_1\alpha_1 \stackrel{X}{\rightarrow}_{lc} w\gamma_1\beta_1\alpha_1 \stackrel{X\cup\{B\}}{\longrightarrow}^*_{sp} w\gamma_1'\beta_1\alpha_1 \rightarrow^* wz_1$
   or
   $S \rightarrow^*_l w\underline{A}\alpha_1 \rightarrow^*_{can(X)} wB\beta_1\alpha_1 \stackrel{X}{\rightarrow}_{sp} w\gamma_1\beta_1\alpha_1 \rightarrow^* wz_1$

   (b) $S \rightarrow^*_l w\underline{A}\alpha_2 \rightarrow^*_{can(X)} wB\beta_2\alpha_2 \stackrel{X}{\rightarrow}_{lc} w\gamma_2\beta_2\alpha_2 \stackrel{X\cup\{B\}}{\longrightarrow}^*_{sp} w\gamma_2'\beta_2\alpha_2 \rightarrow^* wz_2$
   or
   $S \rightarrow^*_l w\underline{A}\alpha_2 \rightarrow^*_{can(X)} wB\beta_2\alpha_2 \stackrel{X}{\rightarrow}_{sp} w\gamma_2\beta_2\alpha_2 \rightarrow^* wz_2$

   (c) $k : z_1 = k : z_2$

   imply $\gamma_1 = \gamma_2$.

2. *The conditions*

   (a) $S \rightarrow^*_l w\underline{A}\alpha_1 \rightarrow^*_{can(X)} wB\beta_1\alpha_1 \stackrel{X}{\rightarrow}_{lc} w\gamma_1\beta_1\alpha_1 \stackrel{X\cup\{B\}}{\longrightarrow}^*_{lc} wC\bullet\gamma_1'\beta_1\alpha_1 \rightarrow^*$
   $wv_1\gamma_1'\beta_1\alpha_1 \rightarrow^* wv_1z_1$

   (b) $S \rightarrow^*_l w\underline{A}\alpha_2 \rightarrow^*_{can(X)} wB\beta_2\alpha_2 \stackrel{X}{\rightarrow}_{lc} w\gamma_2\beta_2\alpha_2 \stackrel{X\cup\{B\}}{\longrightarrow}^*_{lc} wC\bullet\gamma_2'\beta_2\alpha_2 \rightarrow^*$
   $wv_2\gamma_2'\beta_2\alpha_2 \rightarrow^* wv_2z_2$

   (c) $k : z_1 = k : z_2$

   imply $\gamma_1 = \gamma_2$.

3. *The statements*

   (a) $S \rightarrow^*_l w\underline{A}\alpha_1 \rightarrow^*_{can(X)} wB\beta_1\alpha_1 \stackrel{X}{\rightarrow}_{lc} w\gamma\beta_1\alpha_1 \stackrel{X\cup\{B\}}{\longrightarrow}^*_{lc} wB\bullet\gamma'\beta_1\alpha_1 \rightarrow^*$
   $wv_1\gamma'\beta_1\alpha_1 \rightarrow^* wv_1z_1$

   (b) $S \rightarrow^*_l w\underline{A}\alpha_2 \rightarrow^*_{can'(X)} wB\bullet\beta_2\alpha_2 \rightarrow^* wv_2\beta_2\alpha_2 \rightarrow^* wv_2z_2$

   (c) $k : z_1 = k : z_2$

   do not hold at the same time.

We very roughly explain why the above definition is equivalent to $C_k(G)$ being deterministic. The first condition states that if there is no barred nonterminal in front of the remaining input then at all times there is only one alternative in $C_k(G)$ of nonterminal B which can be successfully applied.

The second and third conditions cover the case where there is a barred nonterminal in front of the remaining input. According to the second condition there is only one alternative produced by the second clause of Definition 5.4.3 which can be successfully applied. According to the third condition, if the (unique) alternative for B produced by the fourth clause of Definition 5.4.3 can be successfully applied then no alternative for B produced by the second clause can be successfully applied.

By investigating the definitions of LL(k) and LC(k) in [RL70] we can easily see that for every $k > 0$ the LL(k) grammars are included in the C(k) grammars, which are included in the LC(k) grammars.[4] It is straightforward to show that these inclusions are proper.

5.4.2.2 Strong cancellation parsing with lookahead k

Analogously to the strong counterparts of the top-down and left-corner parsers with lookahead k we also have *strong* cancellation parsing with lookahead k.

We compute which strings may follow a nonterminal taking into account that a nonterminal can be called with different cancellation sets. We define

$$FOLLOW_k(A, X) \;=\; \{k : z \mid S \rightarrow_l^* w\underline{B}\alpha \rightarrow_{can'(X)}^* wA\bullet\beta\alpha \rightarrow^* wv\beta\alpha \rightarrow^* wvz\}$$

Strong cancellation parsers with lookahead k are now defined by

Definition 5.4.5 *If $G = (T, N, P, S)$ then $STC_k(G)$ is the DCG with the new start symbol S' and the following rules.*

1. $S' \rightarrow S(\emptyset)$

2. $A(x) \rightarrow \{A \notin x\},$
   $\{f = L_k(B_1, x \cup \{A\}) \circ_k F \circ_k REC_k(A, x) \circ_k FOLLOW_k(A, x)\},$
   $lookahead_k(f),$
   $B_1(x \cup \{A\}),\ v_1,\ B_2(\emptyset),\ v_2,\ \ldots,\ B_n(\emptyset),\ v_n,\ untoken(\overline{A}),\ A(x).$
   for all $A \rightarrow B_1, v_1, B_2, v_2, \ldots, B_n, v_n. \in P$, such that $n > 0$, and where F is the constant set $L_k(v_1 B_2 v_2 \ldots B_n v_n)$

3. $A(x) \rightarrow \{A \notin x\},$
   $\{f = F \circ_k REC_k(A, x) \circ_k FOLLOW_k(A, x)\},\ lookahead_k(f),$
   $v_1,\ B_2(\emptyset),\ v_2,\ \ldots,\ B_n(\emptyset),\ v_n,\ untoken(\overline{A}),\ A(x).$
   for all $A \rightarrow v_1, B_2, v_2, \ldots, B_n, v_n. \in P$, such that $n = 1$ or $n > 1 \wedge v_1 \neq \epsilon$, and where F is the constant set $L_k(v_1 B_2 v_2 \ldots B_n v_n)$

4. $A(x) \rightarrow \{f = \{\overline{A}\} \circ_k FOLLOW_k(A, x)\},\ lookahead_k(f),\ [\overline{A}].$
   for all $A \in N$

A grammar G which is such that $STC_k(G)$ is deterministic is said to be *strong* C(k). A less operational definition of strong C(k) can be obtained from Definition 5.4.4 by replacing w and A by w_1 and A_1 in parts *(a)* and by w_2 and A_2 in parts *(b)*.

[4] There is an alternative definition of LC(k) in [SSU79], which is incompatible with the one in [RL70].

5.4.2.3 Severe cancellation parsing with lookahead k

Note that in $STC_k(G)$ the predicate $lookahead_k$ is applied on a set of strings which is computed using the cancellation set. We now formulate a third kind of cancellation parser which does not even make use of the cancellation set for the purpose of looking ahead.

For instance, in the second clause of Definition 5.4.5, the set f is computed by

$$L_k(B_1, x \cup \{A\}) \circ_k L_k(v_1 B_2 v_2 \ldots B_n v_n) \circ_k REC_k(A, x) \circ_k FOLLOW_k(A, x)$$

If we are to compute a set f for the purpose of looking ahead without the possibility of using a particular cancellation set X, then the best we can do is to take the union of all the sets resulting from evaluating the above expression for all X that A can be called with, such that $A \notin X$.

Now we define the function $FIRST_k$ from rules in P to sets of strings by

- $FIRST_k(A \to B_1, v_1, B_2, v_2 \ldots, B_n, v_n.) =$
  $\cup\{L_k(B_1, X \cup \{A\}) \circ_k L_k(v_1 B_2 v_2 \ldots B_n v_n) \circ_k REC_k(A, X) \circ_k FOLLOW_k(A, X) \mid$
  $(A, X) \in CALL \wedge A \notin X\}$
  where $n > 0$.

- $FIRST_k(A \to v_1, B_2, v_2 \ldots, B_n, v_n.) =$
  $\cup\{L_k(v_1 B_2 v_2 \ldots B_n v_n) \circ_k REC_k(A, X) \circ_k FOLLOW_k(A, X) \mid$
  $(A, X) \in CALL \wedge A \notin X\}$
  where $n = 1$ or $n > 1 \wedge v_1 \neq \epsilon$.

Although the definition of $FIRST_k(A \to v_1, B_2, v_2 \ldots, B_n, v_n.)$ can be simplified to $L_k(v_1 B_2 v_2 \ldots B_n v_n) \circ_k FOLLOW_k(A)$, it does not seem possible to simplify the definition of $FIRST_k(A \to B_1, v_1, B_2, v_2 \ldots, B_n, v_n.)$ in a meaningful way.

With $FIRST_k$ we can compute sets of strings similar to f in the second and third clause of Definition 5.4.5, but without using the cancellation sets. It would be straightforward to do a similar thing for the fourth clause and to define a function which, given A, computes the union of $FOLLOW_k(A, X)$ for all X such that $(A, X) \in CALL$. (This function is exactly $FOLLOW_k(A)$.) This would however result in a parsing algorithm which could not treat left recursion deterministically. To remedy this problem to some extent, we distinguish between the cases $A \notin X$ and $A \in X$.

We now define

- $FOLLOW_k^{\notin}(A) = \cup\{FOLLOW_k(A, X) \mid (A, X) \in CALL \wedge A \notin X\}$
- $FOLLOW_k^{\in}(A) = \cup\{FOLLOW_k(A, X) \mid (A, X) \in CALL \wedge A \in X\}$

Note that $FOLLOW_k(A) = FOLLOW_k^{\notin}(A) \cup FOLLOW_k^{\in}(A)$.

If we apply the new functions discussed above, we obtain the class of *severe* cancellation parsers with lookahead k, which are defined by

Definition 5.4.6 *If $G = (T, N, P, S)$ then $SEVC_k(G)$ is the DCG with the new start symbol S' and the following rules.*

1. $S' \to S(\emptyset)$.

2. $A(X) \rightarrow \{A \notin X\},\ lookahead_k(F),$
$B_1(X \cup \{A\}),\ v_1,\ B_2(\emptyset),\ v_2,\ \ldots,\ B_n(\emptyset),\ v_n,\ untoken(\overline{A}),\ A(X).$
for all $A \rightarrow B_1, v_1, B_2, v_2, \ldots, B_n, v_n. \in P$, *such that* $n > 0$ *and where* F *is the constant set* $FIRST_k(A \rightarrow B_1, v_1, B_2, v_2 \ldots, B_n, v_n.)$

3. $A(X) \rightarrow \{A \notin X\},\ lookahead_k(F),$
$v_1,\ B_2(\emptyset),\ v_2,\ \ldots,\ B_n(\emptyset),\ v_n,\ untoken(\overline{A}),\ A(X).$
for all $A \rightarrow v_1, B_2, v_2, \ldots, B_n, v_n. \in P$, *such that* $n = 1$ *or* $n > 1 \wedge v_1 \neq \epsilon$, *and where* F *is the constant set* $FIRST_k(A \rightarrow v_1, B_2, v_2 \ldots, B_n, v_n.)$

4. $A(X) \rightarrow \{A \notin X\},\ lookahead_k(F),\ [\overline{A}\,].$
for all $A \in N$, *where* F *is the constant set* $\{\overline{A}\} \circ_k FOLLOW_k^{\notin}(A)$

5. $A(X) \rightarrow \{A \in X\},\ lookahead_k(F),\ [\overline{A}\,].$
for all $A \in N$, *where* F *is the constant set* $\{\overline{A}\} \circ_k FOLLOW_k^{\in}(A)$

A grammar G which is such that $SEVC_k(G)$ is deterministic is said to be *severe* C(k). A less operational definition of severe C(k) can be obtained from Definition 5.4.4 by replacing w, A, and X by w_1, A_1, and X_1 in parts *(a)* and by w_2, A_2, and X_2 in parts *(b)*, and by adding the condition $B \notin X_2$ in part *3(b)*.

Example 5.4.1 Let the grammar G_3 be defined by the following rules

$$\begin{array}{lcl} E & \rightarrow & E,\ [+],\ T. \\ E & \rightarrow & E,\ [-],\ T. \\ E & \rightarrow & T. \end{array}$$

$$\begin{array}{lcl} T & \rightarrow & [-],\ T. \\ T & \rightarrow & [i]. \\ T & \rightarrow & [(],\ E,\ [)]. \end{array}$$

We give the relevant facts for constructing $SEVC_1(G_3)$.

We can easily derive the following.

$$\begin{array}{rcl} GOAL & = & \{E,\ T\} \\ CALL & = & \{(E, \emptyset),\ (E, \{E\}),\ (T, \{E\}),\ (T, \emptyset)\} \end{array}$$

The function $FOLLOW_1$ can be computed by standard algorithms.

$$\begin{array}{rcl} FOLLOW_1(E) & = & \{\epsilon, +, -,)\} \\ FOLLOW_1(T) & = & \{\epsilon, +, -,)\} \end{array}$$

We now compute the sets we need for looking ahead for the second and third clause of Definition 5.4.6.

$$FIRST_1(E \rightarrow E, [+], T.) \ = \ L_1(E, \{E\}) \circ_1 L_1(+T) \circ_1 REC_1(E, \emptyset) \circ_1 FOLLOW_1(E, \emptyset)$$

$$
\begin{array}{rcl}
 & = & \{\overline{E}\} \circ_1 \{+\} \circ_1 \mathit{REC}_1(E,\emptyset) \circ_1 \mathit{FOLLOW}_1(E,\emptyset) \\
 & = & \{\overline{E}+\} \\
\mathit{FIRST}_1(E \to E,[-],T.) & = & \{\overline{E}-\} \\
\mathit{FIRST}_1(E \to T.) & = & L_1(T,\{E\}) \circ_1 L_1(\epsilon) \circ_1 \mathit{REC}_1(E,\emptyset) \circ_1 \mathit{FOLLOW}_1(E,\emptyset) \\
 & = & \{-,\ i,\ (,\ \overline{T}\}\ \circ_1\ \{\epsilon\}\ \circ_1\ \{\epsilon,\ +,\ -\}\ \circ_1\ \{\epsilon,\ )\} \\
 & = & \{-,\ i,\ (,\ \overline{T},\ \overline{T}+,\ \overline{T}-,\ \overline{T})\} \\
\mathit{FIRST}_1(T \to [-],T.) & = & L_1(-T) \circ_1 \mathit{FOLLOW}_1(T)\ =\ \{-\} \\
\mathit{FIRST}_1(T \to [i].) & = & \{i\} \\
\mathit{FIRST}_1(T \to [(],E,[)].) & = & \{(\}
\end{array}
$$

For looking ahead for the fourth and fifth clause, we need

$$
\begin{array}{rcl}
\{\overline{E}\} \circ_1 \mathit{FOLLOW}_1^{\notin}(E) & = & \{\overline{E}\} \circ_1 \mathit{FOLLOW}_1(E,\emptyset) \\
 & = & \{\overline{E}\} \circ_1 \{\epsilon,\ )\} \\
 & = & \{\overline{E},\ \overline{E})\} \\
\{\overline{T}\} \circ_1 \mathit{FOLLOW}_1^{\notin}(T) & = & \{\overline{T}\} \circ_1 (\ \mathit{FOLLOW}_1(T,\{E\})\ \cup\ \mathit{FOLLOW}_1(T,\emptyset)\ ) \\
 & = & \{\overline{T}\} \circ_1 (\ \{\epsilon,\ +,\ -,\ )\}\ \cup\ \{\epsilon,\ +,\ -,\ )\}\ ) \\
 & = & \{\overline{T},\ \overline{T}+,\ \overline{T}-,\ \overline{T})\} \\
\{\overline{E}\} \circ_1 \mathit{FOLLOW}_1^{\in}(E) & = & \{\overline{E}\} \circ_1 \mathit{FOLLOW}_1(E,\{E\}) \\
 & = & \{\overline{E}\} \circ_1 \{+,\ -\} \\
 & = & \{\overline{E}+,\ \overline{E}-\} \\
\{\overline{T}\} \circ_1 \mathit{FOLLOW}_1^{\in}(T) & = & \emptyset
\end{array}
$$

If we investigate Definition 5.4.6 and the above facts, we can easily see that grammar G_3 is severe C(1). □

5.4.3 A hierarchy of grammar classes

The less information is to be used for the purpose of looking ahead, the fewer are the grammars that can be handled deterministically. We have that the severe C(k) grammars are a proper subset of the strong C(k) grammars which are a proper subset of the C(k) grammars, for $k > 1$.

Example 5.4.2 Let the grammar G_4 be defined by the following rules

$$
\begin{array}{rcl}
S & \to & [a],\ A,\ [b]. \\
S & \to & [b],\ A,\ [a],\ [b]. \\
A & \to & \epsilon. \\
A & \to & [a]. \\
A & \to & A,\ [c].
\end{array}
$$

This grammar is C(2) but not strong C(2). We can explain this informally as follows. In $C_2(G_4)$ the strings (truncated to length 2) that follow A in one of the alternatives of S are

passed around the DCG rules of the parser. This information makes it possible to decide what alternative of A to choose for the lowest occurrence in the spine depending on the first two symbols of the remaining input.

In $STC_2(G_4)$ this information is not available. If the first two symbols of the remaining input are ab then $STC_2(G_4)$ will try both the first and the second alternative for the lowest occurrence of A in the spine. □

Example 5.4.3 Let the grammar G_5 be defined by the following rules

$$\begin{array}{lcl} S & \rightarrow & [a],\ A,\ [b]. \\ S & \rightarrow & [b],\ B,\ [a],\ [b]. \\ B & \rightarrow & A. \\ A & \rightarrow & \epsilon. \\ A & \rightarrow & [a]. \\ A & \rightarrow & A,\ [c]. \end{array}$$

This grammar is strong C(2) but not severe C(2). This can be explained similarly to the previous example. When an alternative has to be chosen for the lowest occurrence of A then in $STC_2(G_5)$ the cancellation set (either $\emptyset$ or $\{B\}$) gives the needed information about what alternative of S has been chosen. This information makes it possible to decide what alternative of A to choose for the lowest occurrence in the spine depending on the first two symbols of the remaining input. □

The class of C(1) grammars is equal to the class of strong C(1) grammars. These classes are not equal to the class of severe C(1) grammars however, although this was suggested in [Ned93c].

Example 5.4.4 A grammar which is strong C(1) need not be severe C(1): the transition from STC_1 to $SEVC_1$ may introduce nondeterminism for the case that $A \notin X$; a choice between the rule according to the fourth clause of Definition 5.4.6 and a rule according to the second clause can in this case not be made by looking one symbol ahead.

As an example, consider the grammar G_6 defined by

$$\begin{array}{lcl} S & \rightarrow & [x],\ A,\ [y],\ B,\ [z]. \\ A & \rightarrow & B. \\ A & \rightarrow & [a]. \\ B & \rightarrow & A,\ [b]. \end{array}$$

This grammar is strong C(1). It is not severe C(1), however. We demonstrate this by investigating what happens in $STC_1(G_6)$ at the calls of A and B in the parser rule corresponding with $S \rightarrow [x], A, [y], B, [z].$, where we assume that in both cases the remaining input starts with ab.

1. For the call of A, after the lowest occurrence of A in the spine has been recognized (which reads a from the input), a barred nonterminal $\overline{A}$ is pushed onto the remaining input and A is called again with set $\emptyset$.

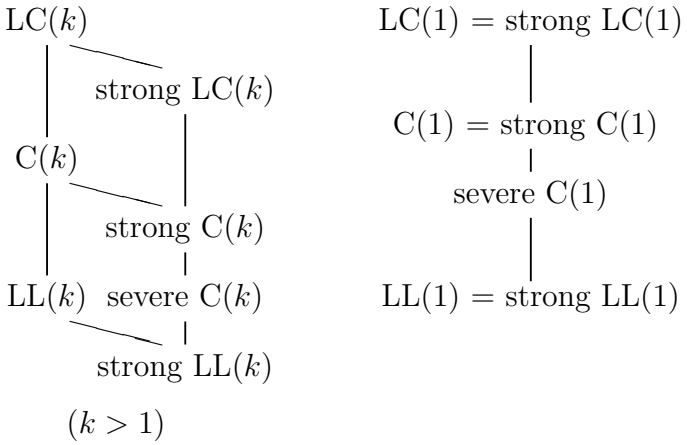


Figure 5.2: Grammar class inclusion hierarchy

2. For the call of B, first the lowest occurrence of B in the spine is recognized, at which rule $B \to A, [b].$ is used. Then A is called with set $\{B\}$. Now the lowest occurrence of A in the spine is recognized (which reads a from the input). After this, a barred nonterminal $\overline{A}$ is pushed onto the remaining input, and A is called again with set $\{B\}$.

In both of these situations, the remaining input starts with $\overline{A}b$. In the first situation, the proper parser rule to select is the one which is produced by the second clause of Definition 5.4.5, based on the grammar rule $A \to B$. In the second situation, the proper parser rule to select is the one stemming from the fourth clause of Definition 5.4.5.

$SEVC_1(G_6)$ cannot distinguish between these two cases, because it cannot investigate the cancellation sets for the purpose of using lookahead. The grammar is therefore not severe C(1). (In fact, it is not even severe C(k) for any k.) □

When we investigate the definitions of strong LL(k) and strong LC(k) in [RL70] we can easily see that for every $k > 0$ the strong LL(k) grammars are included in the severe C(k) grammars, and the strong C(k) grammars are included in the strong LC(k) grammars. It is straightforward to show that these inclusions are proper.

Figure 5.2 summarises the inclusion relations between various classes of grammars. We distinguish between the cases $k > 1$ and $k = 1$. All inclusions in the figure are proper. For the case $k > 1$, if an inclusion is not shown explicitly, then two grammar classes are incomparable.

In addition to the hierarchy we have the following theorems.

1. If a grammar is not left-recursive then it is LL(k) if and only if it is C(k) and it is strong LL(k) if and only if it is severe (not strong!) C(k).

2. A grammar G is (strong) C(k) if and only if $C_{CF}(G)$ is (strong) LL(k).

5.5 Grammars with hidden left recursion

Cancellation parsing as well as LC parsing cannot handle grammars which contain hidden left recursion or which are not cycle-free (i.e. exactly for these grammars termination is not

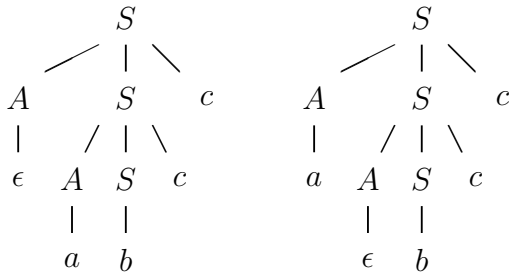


Figure 5.3: Two parse trees showing ambiguity of G_7

guaranteed). Hidden left recursion occurs very often in grammars containing epsilon rules.

A typical example of hidden left recursion occurs in the grammar G_7, defined by

$$\begin{aligned} S &\rightarrow A,\ S,\ [c]. \\ S &\rightarrow [b]. \\ A &\rightarrow \epsilon. \\ A &\rightarrow [a]. \end{aligned}$$

In the recognizer $C(G_7)$ the following may happen. S is called with cancellation set $\emptyset$ and S calls A with set $\{S\}$ according to the alternative $S \rightarrow A, S, [c]$. Then for A the first alternative $A \rightarrow \epsilon$ is taken. After some more actions, S is called again with set $\emptyset$ because it is the second member in the rule $S \rightarrow A, S, [c]$.

This process may repeat itself indefinitely because no input is being read. Thus cancellation parsing does not terminate on all input for grammar G_7. This holds even if lookahead is applied.

Note that G_7 is ambiguous (Figure 5.3), as is any other grammar with hidden left recursion if the empty-generating nonterminal which hides the left recursion also generates a non-empty string.

The key idea of our solution to hidden left recursion is that subderivations which derive the empty string should not interfere with the mechanism of cancellation sets, which would normally cause termination. For this purpose, we use normal TD parsing for subderivations deriving the empty string (which must terminate if the grammar is not cyclic). We change the cancellation parsing strategy for subderivations deriving a non-empty string as follows: in case of a grammar rule which begins with a number of nonterminals, the same cancellation set is passed on to each of the nonterminal members until one of them reads a non-empty string; in other words, the nonterminals which do not read any input are ignored for the sake of the cancellation set.

That a call of a nonterminal should read (or has read) a non-empty or an empty string is indicated by an extra parameter which is *TRUE* or *FALSE* in these respective cases.

In the new version of cancellation parsing, which we call *extended cancellation parsing*, the cancellation sets achieve termination of the recognition of possible maximal *backbones*, where a backbone is similar to a spine, except that it ignores nodes in the parse tree which derive the empty string. More precisely, we define a *backbone* to be a path in a parse tree

which begins at some node not deriving the empty string, then proceeds downwards every time taking the leftmost son not deriving the empty string.

Example 5.5.1 Let the grammar G_8 be defined by

$$\begin{array}{lcl} S & \rightarrow & A,\ A,\ S,\ [s],\ A. \\ S & \rightarrow & A,\ A. \\ A & \rightarrow & S,\ [a]. \\ A & \rightarrow & \epsilon. \end{array}$$

The recognizer $C_{ext}(G_8)$ is defined by the following rules.

$$\begin{array}{rcll} S' & \rightarrow & S(\emptyset, e). & (1) \\ S(x, TRUE) & \rightarrow & \{S \notin x\},\ \{x_1 = x \cup \{S\}\},\ A(x_1, e_1), & (2) \\ & & \{\mathit{if}\ e_1\ \mathit{then}\ x_2 = \emptyset\ \mathit{else}\ x_2 = x_1\},\ A(x_2, e_2), & \\ & & \{\mathit{if}\ e_1 \vee e_2\ \mathit{then}\ x_3 = \emptyset\ \mathit{else}\ x_3 = x_1\},\ S(x_3, e_3), & \\ & & [s],\ A(\emptyset, e_4),\ \mathit{untoken}(\overline{S}),\ S(x, TRUE). & \\ S(x, TRUE) & \rightarrow & \{S \notin x\},\ \{x_1 = x \cup \{S\}\},\ A(x_1, e_1), & (3) \\ & & \{\mathit{if}\ e_1\ \mathit{then}\ x_2 = \emptyset\ \mathit{else}\ x_2 = x_1\},\ A(x_2, e_2), & \\ & & \{e_1 \vee e_2 = TRUE\},\ \mathit{untoken}(\overline{S}),\ S(x, TRUE). & \\ S(x, TRUE) & \rightarrow & [\overline{S}\,]. & (4) \\ S(x, FALSE) & \rightarrow & S. & (5) \\ S & \rightarrow & A,\ A. & (6) \\ A(x, TRUE) & \rightarrow & \{A \notin x\},\ \{x_1 = x \cup \{A\}\},\ S(x_1, e_1), & (7) \\ & & [a],\ \mathit{untoken}(\overline{A}),\ A(x, TRUE). & \\ A(x, TRUE) & \rightarrow & [\overline{A}\,]. & (8) \\ A(x, FALSE) & \rightarrow & A. & (9) \\ A & \rightarrow & \epsilon. & (10) \end{array}$$

We assume the existence of a predicate *if* _ *then* _ *else* _ with obvious semantics.

In (2) it can be seen how the cancellation set $x \cup \{S\}$ is given to each nonterminal until some input has been read. An empty cancellation set is given to nonterminals only after input has been read. E.g. the member S is given the set x_3, which is $x \cup \{S\}$ if no input has been read by the previous two calls to A (i.e. if $e_1 \vee e_2$ is $FALSE$), and which is $\emptyset$, otherwise.

In (3) the condition $\{e_1 \vee e_2 = TRUE\}$ is necessary to make sure that application of this rule reads some input.

Rules (6) and (10) perform ordinary top-down parsing of the empty string, without using any cancellation set. □

In full generality, an extended cancellation recognizer is to be constructed by the following definition.

Definition 5.5.1 *If $G = (T, N, P, S)$ then $C_{ext}(G)$ is the DCG with the new start symbol S' and the following rules.*

1. $S' \rightarrow S(\emptyset, e)$

2. $A(x, TRUE) \rightarrow \{A \notin x\},\ \{x_1 = x \cup \{A\}\},\ B_1(x_1, e_1),$
   $\{if\ e_1\ then\ x_2 = \emptyset\ else\ x_2 = x_1\},\ B_2(x_2, e_2),$
   $\{if\ e_1 \vee e_2\ then\ x_3 = \emptyset\ else\ x_3 = x_1\},\ B_3(x_3, e_3),$
   $\ldots,$
   $\{if\ e_1 \vee \ldots \vee e_{n-1}\ then\ x_n = \emptyset\ else\ x_n = x_1\},\ B_n(x_n, e_n),$
   $v_1, C_2(\emptyset, e'_1), v_2, \ldots, C_m(\emptyset, e'_m), v_m,\ untoken(\overline{A}),\ A(x, TRUE).$
   for all $A \rightarrow B_1, \ldots, B_n, v_1, C_2, v_2, \ldots, C_m, v_m. \in P$, where $n > 0$ and $m > 0$, such that $B_1 \ldots B_{n-1} \rightarrow^* \epsilon$ and $\neg(B_n v_1 \rightarrow^* \epsilon)$

3. $A(x, TRUE) \rightarrow \{A \notin x\},\ \{x_1 = x \cup \{A\}\},\ B_1(x_1, e_1),$
   $\{if\ e_1\ then\ x_2 = \emptyset\ else\ x_2 = x_1\},\ B_2(x_2, e_2),$
   $\{if\ e_1 \vee e_2\ then\ x_3 = \emptyset\ else\ x_3 = x_1\},\ B_3(x_3, e_3),$
   $\ldots,$
   $\{if\ e_1 \vee \ldots \vee e_{n-1}\ then\ x_n = \emptyset\ else\ x_n = x_1\},\ B_n(x_n, e_n),$
   $\{e_1 \vee \ldots \vee e_n = TRUE\},\ untoken(\overline{A}),\ A(x, TRUE).$
   for all $A \rightarrow B_1, \ldots, B_n. \in P$, where $n > 0$, such that $B_1 \ldots B_n \rightarrow^* \epsilon$ and $\exists_v[v \neq \epsilon \wedge B_1 \ldots B_n \rightarrow^* v]$

4. $A(x, TRUE) \rightarrow \{A \notin x\},\ v_1,\ B_2(\emptyset, e_2),\ v_2,\ \ldots,\ B_n(\emptyset, e_n),\ v_n,$
   $untoken(\overline{A}),\ A(x, TRUE).$
   for all $A \rightarrow v_1, B_2, v_2, \ldots, B_n, v_n. \in P$, where $n > 0$ and $v_1 \neq \epsilon$

5. $A(x, TRUE) \rightarrow [\overline{A}\,].$
   for all $A \in N$ such that $\exists_v[v \neq \epsilon \wedge A \rightarrow^* v]$

6. $A(x, FALSE) \rightarrow A.$
   for all $A \in N$ such that $A \rightarrow^* \epsilon$

7. $A \rightarrow B_1,\ \ldots,\ B_n.$
   for all $A \rightarrow B_1, \ldots, B_n. \in P$, where $n > 0$, such that $B_1 \ldots B_n \rightarrow^* \epsilon$

8. $A \rightarrow \epsilon.$
   for all $A \rightarrow \epsilon. \in P$

Note that for the construction of extended cancellation parsers we need the knowledge of which nonterminals generate the empty string and which generate a non-empty string.

Further discussion of hidden left recursion can be found in Chapters 2 and 4.

The parser $C_{ext}(G)$ terminates on all input for every grammar G which is cycle-free. Termination can also be obtained for arbitrary grammars in a way which depends on the application.

If C_{ext} is to be applied to context-free grammars which are extended with arguments, then termination may be forced by so called attribute-influenced parsing [Mah87]. This holds for formalisms such as DCGs and affix grammars [Kos71].

In other cases, the parser construction itself should be extended such that at most a finite subset of infinitely many possible parse trees is recognized. We give some hints how C_{ext} can be further extended in order to handle arbitrary grammars.

There are two ways in which $C_{ext}(G)$ may not terminate. The first way is when parsing with rules produced by Clauses 7 and 8 does not terminate. This can be easily fixed by using sets similar to the cancellation sets (see the discussion on the algorithm in [She76] in Section 5.2.2).

Another way in which $C_{ext}(G)$ may not terminate is when parsing using the rules produced by Clauses 2, 3, 4, and 5 does not terminate. In this case the following happens. At the end of an alternative a barred nonterminal is pushed in front of the remaining input, the nonterminal then calls itself, and recognizes a non-empty part of the spine without reading any input except the barred nonterminal. After this, again a barred nonterminal is pushed and the process starts from the beginning.

To force this process to terminate, we must make sure that if a barred nonterminal $\overline{A}$ is pushed, then the subsequent call to A either chooses the alternative $A(X, \mathit{TRUE}) \to [\overline{A}\,]$. or else reads some non-empty terminal string from the input (besides the barred nonterminal), before pushing another $\overline{A}$.

The details of how to extend C_{ext} for this purpose are left to the reader.

5.6 Run-time costs of cancellation parsing

In this section we investigate the costs of cancellation parsing, especially with regard to the costs of top-down parsing from which it is derived.

First we remark that if a cancellation parser terminates on all input, then the grammar is cycle-free and every parse tree has a size which is linear in the length of the input w. This means that also the number of cancellation sets in use in a parser at any time is linear in $|w|$.

For the implementation of the cancellation sets we distinguish between deterministic and nondeterministic parsing.

5.6.1 Costs of nondeterministic parsing

In the case of nondeterministic parsing, each of the $2^{|N|}$ possible subsets of N may at some time be in use as a cancellation set. The most efficient way of representing cancellation sets is then by using arrays of boolean values. Different cancellation sets for the purpose of recognition of a single maximal spine may share the same array. The set $X \cup \{A\}$ is created out of X simply by setting the value in the array indexed by A to *TRUE*. Conversely, the set X is created out of $X \cup \{A\}$ simply by setting the value in the array indexed by A to *FALSE* again.

The space requirements for nondeterministic parsing are now $\mathcal{O}(|w| \times |N|)$.

5.6.2 Costs of deterministic parsing

In the case of deterministic parsing, we can easily show that the number of different cancellation sets that nonterminals may ever be called with is bounded by an expression which

depends amongst others on k.

For strong cancellation parsing with lookahead k this expression is

$$|GOAL| \times (|T|^k + |N| \times |T|^k) \times (|N| + 1)$$

This expression can be justified as follows: when a nonterminal A in $GOAL$ is called, with which recognition of a spine is initiated, then at most $|N| + 1$ nonterminals are deterministically called with different cancellation sets, based on the remaining input. The relevant part of the remaining input can be of the form w or $\overline{B}w$, where $|w| = k$ (we neglect the case that the remaining input has length smaller than k). We have simplified the situation here by assuming that, after a barred nonterminal $\overline{B}$ is pushed onto the remaining input, again the original nonterminal A in $GOAL$ is called, instead of nonterminal B.

For cancellation parsing with lookahead k we must allow for the possibility that a nonterminal in $GOAL$ generates different strings of lengths smaller than k, one of which may be a prefix of another. The number of cancellation sets is now bounded by

$$\begin{aligned} & |GOAL| \times (1 + |N|) \times (|T|^0 + |T|^1 + \ldots + |T|^k) \times (|N| + 1) \\ \leq\ & |GOAL| \times (1 + |N|) \times (|T| + 1)^k \times (|N| + 1) \end{aligned}$$

For both cancellation parsing with lookahead k and for strong cancellation parsing with lookahead k, the number of possible cancellation sets is consequently $\mathcal{O}(|N|^3 \times |T|^k)$.

In the case of severe cancellation parsing with lookahead k, we may completely dispense with the use of cancellation sets. This can be explained as follows: assume that deterministic choices are possible between rules according to the five clauses of Definition 5.4.6. If we now call nonterminal A with set X, during recognition of some *correct* input, then we may have the following cases.

- If the first symbol of the remaining input is not the barred nonterminal $\overline{A}$, then the rules according to the fourth and fifth clause are not applicable, and therefore we know that a rule according to the second clause should be applicable, which implies $A \notin X$.
- If the first symbol of the remaining input is the barred nonterminal $\overline{A}$, then depending on whether $A \in X$ a rule according to the fourth or fifth clause may be applicable. If not, then as above we know that a rule according to the second clause should be applicable, which implies $A \notin X$.

We conclude that for *correct* input, we only need to know whether $A \in X$ when the barred nonterminal $\overline{A}$ occurs in the remaining input. For this we do not need to maintain X itself. (Details are left to the imagination of the reader.)

It can be easily seen that if we dispense with cancellation sets and the tests $\{A \in x\}$, termination is also guaranteed for *incorrect* input. Furthermore, we conjecture that error-detection is not delayed (proof is currently lacking however).

In practice, the number of possible cancellation sets for deterministic parsing in the case of $C_k(G)$ and $STC_k(G)$ is much less than $2^{|N|}$ and even much less than the very rough upper bounds we saw above.

We suggest that each cancellation set X for nonterminal A is represented by a unique integer $i_{A,X} \in \{1, \ldots, |CALL|\}$. We then need a table of size $|CALL| \times |N|$ in order to be able to compute $i_{B,X\cup\{A\}}$ from $i_{A,X}$ and B. The dynamic space complexity of cancellation parsing is now of the same order of magnitude as that of top-down parsing. We further need a table of size $|CALL| \times (|T|^k + |N| \times |T|^k)$ to store $L_k(\_, \_)$. We need some smaller tables to store $REC_k(\_, \_)$ and $FOLLOW_k(\_, \_)$.

For $SEVC_k(G)$ we need a table of size $|P| \times (|T|^k + |N| \times |T|^k)$ to store $FIRST_k(\_, \_)$. This is the largest table we need for this kind of parsing. Note that the size of this table is only a factor $1 + |N|$ larger than the size of the table we need to store $L_k(\_)$ for $STD_k(G)$.

The most important element in the time complexity of parsing with $C_k(G)$ is the evaluation of $\circ_k$. Evaluation of the same function occurs in $TD_k(G)$, although $\circ_k$ has to be evaluated less often in this case. We conclude that the time complexities of deterministic parsing using $C_k(G)$ and $TD_k(G)$ are of the same order of magnitude. The same holds for $SEVC_k(G)$ and $STD_k(G)$ for which the function $\circ_k$ does not have to be evaluated at run-time. The parser $STC_k(G)$ holds the middle between $C_k(G)$ and $SEVC_k(G)$ in the sense that $\circ_k$ may be evaluated either at run-time or at compile-time.

An important property of cancellation parsing is that it is closely related to top-down parsing. A consequence is that in case of a grammar which is "almost" (strong) LL(k) except for some left recursion, we have the following: the parts of a cancellation parser which correspond with the rules of the grammar not containing the left recursion are not much larger than the corresponding parts would be in a top-down parser. A similar fact holds for the time complexity.

5.7 Related literature

Cancellation parsing is reminiscent of the algorithm ET^* (*extension tables*) in [Die87]. This algorithm is applicable to arbitrary logic programs, but if we only consider its use for TD parsing then we can make the following interesting comparison. The algorithm ET^* ensures termination by blocking the recursive (and in fact even all subsequent) calls to each nonterminal at each input position, similarly to how cancellation parsing solves the problem of termination. However, ET^* has a different approach to recognizing spines with more than one occurrence of a single nonterminal, which is informally expressed by saying that global mechanisms are used where cancellation parsing would use mechanisms local to grammar rules. Going into more detail, we may capture the differences by the following:

- Cancellation parsing uses *untoken*($\overline{A}$) to *temporarily* store the fact that an occurrence of A has been found. Algorithm ET^* stores similar facts (e.g. that A derives a particular part of the input) in a global table, the *extension table*. Such facts remain stored in the table indefinitely, and can be retrieved by arbitrary calls to nonterminals A at appropriate input positions.

- Cancellation parsing allows a nonterminal to be called again at the end of a rule. Algorithm ET^*, on the other hand, iterates by repeatedly calling the start symbol S at the beginning of the input, until eventually an iteration yields no new (sub)parses. This approach causes all subparses to be recomputed during each iteration of the algorithm.

Algorithm ET^* seems to be less efficient than cancellation parsing, because it requires all subparses to be recomputed during each iteration (and the number of iterations needed is linear in the length of the input), although, on the other hand, subparses are never computed more than once during a single iteration.

A more sophisticated algorithm is described in [LCVH93]. Apart from superficial differences with ET^* (the algorithm in [LCVH93] deals with functions instead of with logic programs) the main difference is an optimization which avoids some unnecessary recomputations. This is accomplished by recording the dependences between results. (In our application domain such a result would be the set of all values j yielded by calls to $A(i, j)$, with A a nonterminal and i an instantiated string argument.) The computation of some result is redone only if some other result upon which it depends has been updated. It may therefore be concluded that even less results are recomputed than in the case of cancellation parsing. However, unlike cancellation parsing, the algorithm in [LCVH93] uses global tables, which complicates its use for some purposes for which cancellation parsing was specially devised.

5.8 Semi left-corner parsing

In the previous sections we have seen that cancellation parsing has a more top-down nature than left-corner parsing has. In this section, which has the character of an appendix to this chapter, we show that left-corner parsing can be adapted to allow some top-down processing, and therefore some top-to-bottom flow of argument values. The resulting parsing algorithm will be called *semi left-corner parsing*, since it applies the left-corner strategy only to a restricted part of a grammar. It was first mentioned in [Fos68].

We first need some definitions. We define the equivalence relation $\sim$ on nonterminals by

$$A \sim B \quad \text{if and only if} \quad A \angle^* B \ \wedge \ B \angle^* A$$

If two nonterminals A and B are such that $A \sim B$, then we say that A and B are *mutually left-recursive*.

The idea of semi LC parsing is that the application of the left-corner parsing strategy is restricted to left-recursive nonterminals, and that the normal top-down parsing strategy is applied to all other parts of the grammar.

A second way to explain this is by dividing spines into smaller parts, called *spinelets*. The nodes in a spinelet are labelled with nonterminals which are pairwise mutually left-recursive. Semi left-corner parsing recognizes the spinelets in a spine in a top-down manner, although the nonterminals within a spinelet are recognized in a bottom-up manner.

As defined in Section 5.2.3, the set $GOAL$ contains all nonterminals which may occur at the highest node of a spine. We now define the set $GOAL'$ of all nonterminals which may occur at the highest node of a spinelet by

$$GOAL' \ = \ GOAL \ \cup \ \{B \mid \exists_{A \to B, \alpha . \in P}[A \not\sim B]\}$$

The nonterminals which are left-recursive, or in other words, which may occur in a spinelet consisting of more than one node, are defined by

$$LR \ = \ \{A \mid A \angle^+ A\}$$

The construction of semi left-corner parsers is now given by the following definition.

Definition 5.8.1 *If $G = (T, N, P, S)$ then $SLC(G)$ is the DCG with the same start symbol S and the rules below.*

1. $A \rightarrow \alpha.$
   for all $A \rightarrow \alpha. \in P$ *such that* $A \in GOAL' \setminus LR$

2. $A \rightarrow goal(A).$
   for all $A \in GOAL' \cap LR$

3. $goal(g) \rightarrow \{B \sim g\},\ \beta,\ B'(g).$
   for all $B \rightarrow \beta. \in P$ *such that* β *begins with a terminal or with a nonterminal* C *with* $C \not\sim B$, *or* β *is* ϵ

4. $C'(g) \rightarrow \beta,\ B'(g).$
   for all $B \rightarrow C, \beta. \in P$ *such that* $B \sim C$

5. $A'(A) \rightarrow \epsilon.$
   for all $A \in GOAL' \cap LR$

Note that a recognizer $SLC(G)$ does not apply the left-corner relation $\angle^*$, which means that the size of the recognizer is linear in the size of the grammar G.

Example 5.8.1 Let the grammar G_9 be defined by

$$\begin{array}{lcl} S & \rightarrow & A,\ [a]. \\ A & \rightarrow & S,\ [s]. \\ A & \rightarrow & B. \\ B & \rightarrow & [b]. \\ B & \rightarrow & [c],\ S,\ A. \end{array}$$

The equivalence classes of $\sim$ are $\{B\}$ and $\{S, A\}$. We further have $GOAL' = \{S, A, B\}$, and $LR = \{S, A\}$.

The recognizer $SLC(G_9)$ is now defined by

$$\begin{array}{rcl} S & \rightarrow & goal(S). \\ A & \rightarrow & goal(A). \\ goal(g) & \rightarrow & \{A \sim g\},\ B,\ A'(g). \\ B & \rightarrow & [b]. \\ B & \rightarrow & [c],\ S,\ A. \\ A'(g) & \rightarrow & [a],\ S'(g). \\ S'(g) & \rightarrow & [s],\ A'(g). \\ S'(S) & \rightarrow & \epsilon. \\ A'(A) & \rightarrow & \epsilon. \end{array}$$

□

Nondeterministic semi LC parsing terminates for the same class of grammars as normal LC parsing does, viz. for all cycle-free grammars without hidden left recursion.

Both cancellation parsing and semi LC parsing can be regarded to recognize a spine partly using a top-down parsing strategy. The difference is that semi LC parsing uses a top-down strategy only for nonterminals which are not left-recursive, whereas cancellation parsing also uses a top-down strategy for other nonterminals in Z (see Section 5.2.6 for a definition of Z). An important consequence is that more top-to-bottom flow of argument values is possible using cancellation parsing, which may be useful for early rejection of incorrect derivations.

In [Den94] another way to combine TD parsing with LC parsing is discussed: the LC strategy is used for groups of rules in which parameter values are passed unchanged from the lhs to the first member in the rhs, and for the other cases the top-down strategy is used. In this way, top-to-bottom flow of argument values can always be realized, although termination in the presence of left recursion cannot be guaranteed.

5.9 Conclusions

In this chapter we have introduced a new class of parsing algorithms, which is collectively called *cancellation parsing*.

We have shown that the space and time requirements of cancellation parsing are not much larger that those of top-down parsing. In this way we have demonstrated the feasibility of cancellation parsing.

As we have shown, this kind of parsing has a unique combination of properties: it can deal with left recursion, although at the same time it has a strong top-down nature, which allows early evaluation of arguments and therefore early rejection of incorrect derivations.

That cancellation parsing has a top-down nature can also be expressed by saying that the structure of the parser is very much related to the structure of the grammar. The advantages of this are that a minimal amount of analysis is needed to construct a parser, and that debugging of grammars is facilitated.

Because of these properties, we expect cancellation parsing to be particularly well suited for implementations of grammatical formalisms in environments which put particular constraints on the parsing algorithm, such as definite clause grammars in some Prolog environments.

Chapter 6

Efficient Decoration of Parse Forests

6.1 Introduction

Large subsets of natural languages can be described using context-free grammars extended with some kind of parameter mechanism, e.g. feature formalisms, affix grammars, attribute grammars, and definite clause grammars. This chapter deals with affix grammars over finite lattices (AGFLs). The parameters in AGFLs are called *affixes*. AGFLs are a simple formalism but have still been proved powerful enough for studying the description of various natural languages (see Chapter 7).

Context-free parsing for ambiguous languages can be implemented very efficiently by constructing *parse forests*. A parse forest is a concise representation of a number of parse trees for an ambiguous sentence. Parse forests can be constructed in polynomial time and require polynomial space for storage, measured in the length of the input sentence.

In this chapter, the extra parameters of extended context-free grammars are used to impose restrictions (called *context dependencies*) on the allowable parse trees. Context-free parsing must therefore be augmented to calculate the parameters. For a single parse tree, affix values for AGFLs can be evaluated efficiently because the domains are finite. The rejection of parse trees based on violated context dependencies is more complicated however if parse trees are merged into parse forests.

Because of the ambiguous nature of natural language, grammars describing natural languages are necessarily also ambiguous, and therefore context dependencies cannot determine a unique parse tree for every sentence. Instead, some human-computer interaction is necessary to find the intended parses of sentences among the syntactically correct ones.[1]

This chapter proposes to merge human-computer interaction for the disambiguation of sentences with the calculation of affixes. The computer determines an approximation of the affix values in a forest. All parts of the forest which cannot be part of any correct tree because of violated context dependencies are discarded automatically. The user performs a part of the disambiguation by rejecting some of the remaining parts of the parse forest.

[1]No explicit provisions have been made in our formalism for semantic or pragmatic disambiguation. One may however envisage a system similar to ours in which mechanized semantic and pragmatic analysis replaces the user in resolving the syntactic ambiguities. We will not go into this possibility since it is outside our range of interest. Neither do we address the issue of probabilistic processing (see [Car93, Section 6.3], which discusses the problem of selecting specific parse trees from a parse forest, using probabilities).

Because elimination of parts of the forest may trigger more accurate approximation of the affix values, the user-directed manipulation of the forest and the calculation of affix values may be performed alternately. This process ends with a unique parse tree decorated with affix values.

Because the constituent parse trees in a forest do not need to be investigated separately, our method provides a way to store and handle parse forests in a space-efficient and time-efficient way: at every moment only a polynomial amount of space is needed, whereas the number of parse trees treated may be exponential in the length of the input sentence.

A correct parse tree is found in linear time, measured in the size of the parse forest. The practical time costs are reduced by applying two related algorithms in order to achieve our end: the first, which has a relatively small time complexity, determines a crude approximation of the affix values; the second narrows down the affix values further so that they are as precise as is possible without further communication with the user.

The second algorithm has a high theoretical time complexity if it is applied without previously applying the first algorithm. The first algorithm however reduces the forest and its affix values considerably, so that in practical cases subsequent application of the second algorithm becomes feasible.

The structure of this chapter is as follows. In Section 6.2 we define AGFLs. Section 6.3 discusses the structure of parse forests and Section 6.4 deals with some methods of finding correct parse trees in a parse forest. The particular method we apply is discussed in more detail in Section 6.5, and its time complexity is investigated in Section 6.6. We end this chapter in Section 6.7 with a justification for the use of our method in the AGFL project.

6.2 Affix grammars over finite lattices

Affix grammars over finite lattices (AGFLs) are a restricted form of affix grammars [Kos71, Kos91a]. An affix grammar can be seen as a context-free grammar of which the productions are extended with *affixes* (cf. parameters, attributes, or features) to express agreement between parts of the production. The characteristic of AGFLs is that the domains of the affixes are given by a restricted form of context-free grammar, the *meta grammar*, in which each right-hand side consists of one terminal.

Formally, an AGFL G is a 7-tuple $(A_n, A_t, A_p, G_n, G_t, G_p, S)$ with the following properties.

The disjoint sets A_n and A_t and the function A_p together form the *meta grammar* of G, where

- A_n is the finite set of *affix nonterminals*;
- A_t is the finite set of *affix terminals*;
- A_p is a function from A_n to subsets of A_t. The fact that A_p maps some affix nonterminal N to some set of affix terminals $\{x_1, \ldots, x_m\}$ is written as $N :: x_1; \ldots; x_m$.

  We call the set $\{x_1, \ldots, x_m\}$ the *domain* of N, denoted by $dom(N)$.

The elements from A_n occur in the productions from G_p, which we define shortly, in the so called *displays*. A display is a list of elements from A_n, separated by "," and enclosed in

brackets "(" and ")". Distinguished positions in a display are called *affix positions*. Displays with zero positions are omitted.

The set of lists of zero or more elements from some set D is denoted by D^*. Thus, the set of all displays is formally described as ${A_n}^*$.

For the second part of an AGFL G we have

- G_n is the finite set of *nonterminals*. Each nonterminal has a fixed arity, which is a non-negative integer;
- G_t is the finite set of *terminals* ($G_n \cap G_t = \emptyset$);
- G_p is the finite set of *productions* $\subseteq (G_n \times {A_n}^*) \times ((G_n \times {A_n}^*) \cup G_t)^*$. Productions are written in the form
$$H : H_1, \ldots, H_m.$$
where $m \geq 0$, and H is a nonterminal followed by a display, and each member H_i, $1 \leq i \leq m$, is either a terminal, or a nonterminal followed by a display. The number of affix positions in a display which follows a nonterminal should correspond with the arity of the nonterminal;
- The start symbol $S \in G_n$ has arity 0.

Example 6.2.1 The following is a small AGFL to illustrate some of the definitions above. This example incorporates a number of shorthand constructions, which are explained below.

```
PER ::  1; 2; 3.

NUM ::  sing; plur.

simple sentence:
     pers pron (PER, NUM), to be (PER, NUM), adjective.

pers pron (1, sing):  "I".
pers pron (1, plur):  "we".
pers pron (2, NUM ):  "you".
pers pron (3, sing):  "he"; "she"; "it".
pers pron (3, plur):  "they".

to be (1,   sing):  "am".
to be (PER, NUM ):  "are", [PER = 2  ;  PER = 1 | 3, NUM = plur].
to be (3,   sing):  "is".

adjective:  "funny".
```

The affix terminal "`sing`" in the first alternative of "`pers pron`" constitutes a so called *affix expression*, which is to be seen as an anonymous affix nonterminal, the domain of which is the singleton set {`sing`}. In general, an affix expression consists of one or more affix terminals, separated by "|", and denotes an anonymous affix nonterminal, the domain of which is the set of all specified affix terminals. E.g. the affix expression "`1 | 3`" denotes an affix nonterminal with domain {`1`, `3`}.

In the fourth line of the definition of "`pers pron`", three alternatives sharing the same left-hand side are merged.

Another example of shorthand is the *guard* in the second alternative of "`to be`", which specifies under what conditions a form of "`to be`" is `"are"`. The comma is to be seen as "and" and the semicolon as "or". The operator "`=`" denotes unification of the values of affixes.

Within AGFLs more shorthand constructions are possible. The full syntax of AGFLs is defined in [DKNvZ92].

In the remaining part of this chapter we abstract as much as possible from shorthand constructions and follow the formal definition of AGFLs. □

In the following we use the general term *affix value* for any mathematical object which assigns values to the affixes in a derivation.

From the formal definition it may be concluded that the class of AGFLs is a restricted kind of feature formalism: the atomic values in AGFLs are the affix terminals; complex values however are not possible, due to the absence of constructors such as feature names (or feature symbols) in other feature formalisms, or functors in definite clause grammars. Therefore, the domains of affix nonterminals are finite and consequently, the descriptional power of AGFLs does not transcend the class of context-free languages, contrary to most kinds of feature formalisms.

The grammar consisting of the productions of an AGFL G from which the displays have been eliminated is called the *underlying context-free grammar* of G. A parse tree according to the underlying context-free grammar of an AGFL G is called a *context-free parse tree* according to G.

One approach to decorating context-free parse trees with affix values declares that two parse trees which incorporate the same context-free part represent the same derivation, although the associated affix values may be different. In other words, affix values are not an inherent part of a derivation but they are merely used to accept or reject a derivation. The purpose of decorating a tree is to determine *all* possible affix terminals at each position at each node of the tree.

This view is made more explicit by the following exposition of how a tree can be decorated.

Each node in a decorated parse tree for an AGFL is labelled with a number of affix values. (This number corresponds with the arity of the nonterminal at that node.) In this context an affix value is a nonempty set of affix terminals and is restricted to be a subset of the domain of the affix nonterminal which occurs at the corresponding position in a production applied at that node or its father.

We use the term *variable* to refer to an object which is used to store affix values. A variable identified with a position at a node in a parse tree is said to be an *instance* of an affix nonterminal N if N occurs at the corresponding position in a production applied at that node or its father.

For example, a variable which is an instance of affix nonterminal "`PER`" holds a value which is a nonempty subset of $\{1, 2, 3\}$.

If an affix nonterminal occurs more than once in a single production, then the corresponding variables should always hold the same value. This is generally called *consistent substitution*. For instance, the first production in the running example states that in a

"`simple sentence`", the values of both "`NUM`" and "`PER`" should be the same for "`pers pron`" and "`to be`". In this manner, the principle of consistent substitution allows us to express *context dependency* or *agreement*.

Note that decorated parse trees may contain affix values which are not singleton sets. E.g. in the sentence "`you are funny`", the affix value corresponding to "`NUM`" is {`sing`, `plur`}.

The problem of decorating a context-free parse tree with affix values may now be informally stated as "Find the largest possible affix values for each of the variables such that the requirement of consistent substitution is satisfied."

A straightforward implementation first assigns to each variable the intersection of the domains of the one or two affix nonterminals it is an instance of. Subsequently, affix values are propagated to variables in adjacent parts of the tree (due to consistent substitution), which makes the values of those variables smaller (according to the normal subset ordering[2]). This is repeated until a stable situation is reached. If some variable eventually holds the empty set, then the derivation is rejected.

In the next sections we discuss the decoration of parse forests as opposed to parse trees. In Section 6.5.2 we explain why in parse forests we can no longer treat different affix positions at a node independently. Instead of variables for individual positions, we use tables to store sets of tuples of affix terminals, so that only one table is used for each node. In that context, affix values are sets of tuples of affix terminals rather than sets of affix terminals.

6.3 Parse forests

A *parse forest* is a directed acyclic graph which allows efficient representation of a number of parse trees deriving the same sentence. By some authors, parse forests are more specifically called *shared, shared-packed*, or *packed shared* (parse) forests. Our particular representation of parse forests has been adopted from [vH91].

There are three kinds of nodes:

1. symbol nodes,
2. packing nodes, and
3. sharing nodes.

Symbol nodes correspond to nodes in ordinary parse trees. Packing nodes serve to hold together a number of alternative derivations from one nonterminal. Sharing nodes indicate that certain derivations form part of a number of larger derivations.

All nodes are labelled with elements from $G_n \cup G_t$. We also identify a stub (i.e. a symbol node labelled with a nonterminal, together with its sons) with a fixed production. At a stub, the labels of the sons should correspond to the members in the right-hand side of the associated production.

Parse forests are furthermore subject to the following restrictions:

[2]Note that according to the subset ordering, the possible values of a variable together form a *finite lattice*; hence the name of the formalism. In many other formalisms, such as EAGs [Meij86] (see also DCGs, Chapter 5) variables have a value of exactly one affix terminal.

- There is exactly one node which has no incoming arrows. This node is labelled with S, the start symbol.
- Symbol nodes have at most one father.
- Symbol nodes labelled with a terminal have no sons.
- Packing nodes (indicated by $\triangle$) have one father and at least two sons. The sons have the same label as the packing node itself.
- Sharing nodes (indicated by $\bigtriangledown$) have one son and at least two fathers. The son has the same label as the sharing node itself.
- There are no arrows between sharing nodes and no arrows between packing nodes.
- There are no arrows from packing nodes to sharing nodes.
- No two sons of a symbol node have any descendants in common.

Usually, the strings derived by the sons of a packing node are pairwise the same. This is however not essential to the concept of "parse forest".

Note that a *packed node* as defined in [Tom86] is equivalent to a packing node together with its sons. There is no notion in [Tom86] equivalent to "sharing node"; instead, symbol nodes are allowed to have more than one father.

Parse forests bear a resemblance to disjunctive feature structures [Vér92].

Parse forests can be produced by generalized LR parsing [Tom86], by generalized LC parsing (Chapter 2), and by various tabular parsing algorithms as constructed according to the method of Lang [BL89]. The order of the number of nodes in a parse forest produced by these algorithms is optimal, viz. $\mathcal{O}(|w|^{r+1})$, where w is the input sentence and r the maximum number of members in productions of the grammar. (It is however not true that each of these algorithms always produce optimal parse forests, i.e. ones with the minimal number of nodes. Merely the *order* of the number of nodes is optimal.)

Note that the order of the number of nodes in a parse forest is the same as the order of the number of arrows, because of the restriction that there should not be any arrows from packing nodes to sharing nodes.

6.4 Finding correct parses

A context-free parse forest represents a number of context-free parse trees, some of which can be successfully decorated with affix values and are thereby considered correct. If we want to identify those correct parse trees in the forest, then the following problem arises. One node of the parse forest may represent one node in a number of different parse trees. In each of those trees, the affix values associated with the corresponding node may be different.

Therefore, the identification of correct parse trees in the parse forest is hindered by the fact that at every position at a node in the parse forest one must take into account a number of different affix values belonging to different decorated parse trees.

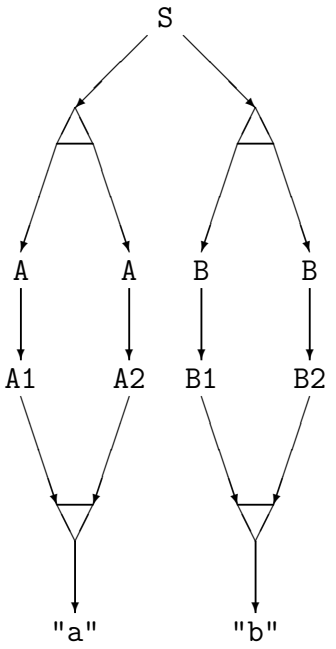


Figure 6.1: A parse forest representing four context-free parse trees

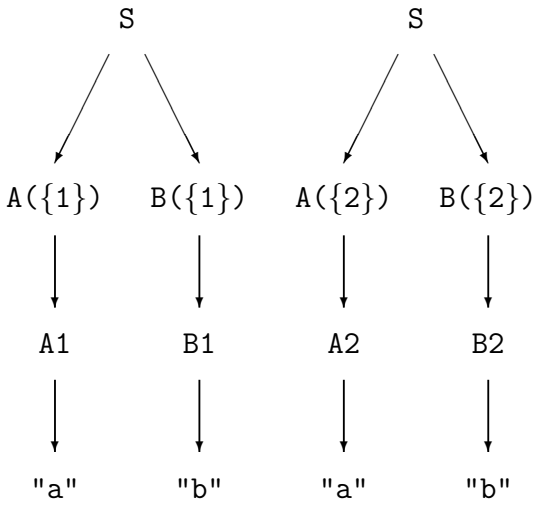


Figure 6.2: The two correctly decorated parse trees

Example 6.4.1 Figure 6.1 shows a context-free parse forest according to the following grammar.

```
NUM ::  1; 2.

S: A(NUM), B(NUM).

A(1):  A1.
A(2):  A2.

B(1):  B1.
B(2):  B2.

A1:  "a".
A2:  "a".
B1:  "b".
B2:  "b".
```

Four context-free parse trees are represented in this forest, one for every combination of choices for the productions of `A` and `B`. However, only the ones given in Figure 6.2 can be successfully decorated. □

There are roughly three ways to identify correct parses in a context-free parse forest:

(1) Evaluate the affix values during construction of the parse forest and abstain from introducing packing nodes if the affix values of subderivations differ. This approach

plays a role in the dynamic programming technique for Datalog Automata [Lan88b], in the combination of generalized LR parsing with semantic actions [PB91] and with attributes [Vos93], in the combination of Earley parsing with feature sets [Kar86] and with Prolog terms [PW83], and in the combination of bottom-up parsing with constraints [BS92].

A related approach can be applied to unification-based grammar formalisms [Per85]: structure sharing is applied as much as possible to the *complex phrase types* (comparable to affix values) which occur at different nodes in different derivations. If two alternative subderivations share their complex phrase types then those subderivations can be "packed" together.

A complementary approach has been applied in a parser for definite clause grammars [MS87]: the need for copying or renaming of values if a subderivation is shared is avoided by restricting the grammar formalism in such a way that the flow of data is from bottom to top.

(2) Decorate the context-free parse forest and restructure it in such a way that it only represents correct derivations. This amounts to expansion of the forest by relocating or eliminating packing and sharing nodes, and by multiplying symbol nodes.

This is similar to the lifting of a disjunction in a feature structure if unification with a second feature structure yields a different global result for both disjuncts [ED88].

The result of the above approach can be given in a more compact form by tagging packing nodes in such a way that correspondence between choices at distant packing nodes can be indicated [vH91]. This is similar to labelling disjunctions in feature terms with disjunction names [DE90].

(3) Compute optimistic affix values in the context of packing and sharing nodes. This causes all parts of the forest to be rejected which cannot be part of any correct parse tree, as far as is possible to determine without further communication with the user.

The user of the system may then reject more parts of the forest. Now more precise but still optimistic affix values are calculated. Again this may cause parts of the forest to be rejected. This process is repeated until all packing and sharing nodes are eliminated. The result is a single decorated parse tree.

In a more general setting, [MK93] discusses the first and second methods and compares their relative time requirements. In that paper, the first method is called *interleaved pruning* (context-free parsing is interleaved with *functional constraint solving*), and the second *non-interleaved pruning*. A related paper is [Nag92b], which discusses different kinds of interaction between the context-free parsing process and the unification process in unification-based parsing.

Because of the size of the domains of typical AGFLs, the first method is not practical for our purposes. The second method requires a very complicated algorithm and also produces very large parse forests.

In our AGFL research we have therefore chosen for the last method outlined above. This is the subject of the remaining part of this chapter.

6.5 Finding a single decorated parse tree

We now give a more explicit exposition of the last approach mentioned in the previous section.

The complete algorithm consists of two parts which are executed one after the other. The first does not take into account the relation between the different affix positions at each node, and therefore yields a crude approximation of the affix values. The second part narrows down the affix values more accurately by considering the relation between affix positions. This part is a costly procedure if executed independently. It is made feasible by previously applying the first part, which reduces the affix values considerably. The second part also handles disambiguation by the user.

We remark that splitting up the complete algorithm in two parts in this way is reminiscent of some methods of query evaluation. For example, [LY91] mentions how unary filters can be used for approximating sets of tuples in a top-down part, whereas the bottom-up part subsequently evaluates the exact tuples.

In the following, we discuss both parts of our complete algorithm independently.

6.5.1 The first part

Before presenting the algorithm of the first part, we first discuss some important aspects.

6.5.1.1 Cells

The main complication in decorating parse forests lies in the presence of packing and sharing nodes. To be able to simplify this process we divide a parse forest into *cells*.[3] A cell is a maximal connected subgraph of the parse forest which contains

- symbol nodes,
- packing nodes, such that the sons are not in the same cell, and
- sharing nodes, such that the fathers are not in the same cell.

Note that packing and sharing nodes form the interfaces between cells.

The evaluation of affix values within a cell can be implemented as follows. Consistent substitution imposes the equality of the values of a number of variables in the cell. These variables are unified, i.e. an administration is set up to make sure that from then on the value of one variable is always the same as that of one of the others. The initial common value of such a group of unified variables is the intersection of the domains of the affix nonterminals of which they are instances. Figure 6.3 gives an example.

Two variables which are unified will act from then on as one and the same object. Unification may be realised according to any of the methods known from the implementation of Prolog.[4]

[3] Cells are already present, although very implicitly, in Section A.4 of [Lan88c].

[4] However, most implementations of Prolog use a non-linear algorithm. In Section 6.6 we assume a linear unification algorithm, such as the one in [dC86].

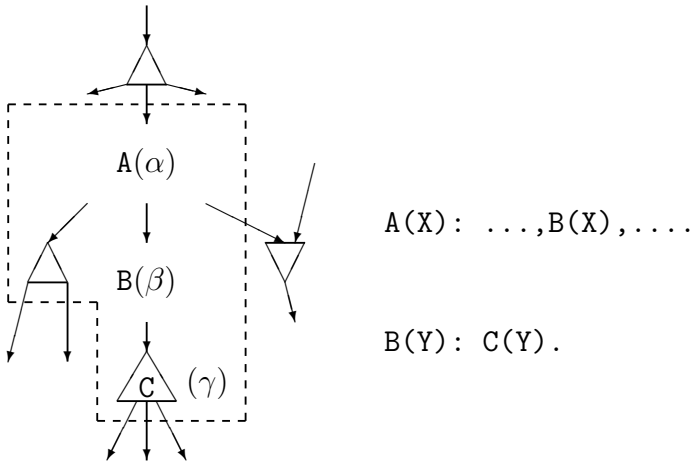


Figure 6.3: A cell in a parse forest. α, β, and γ denote variables. Because of consistent substitution, these are unified. Their initial common value is $dom(X) \cap dom(Y)$.

6.5.1.2 Restricted interfaces to other cells

As opposed to the relation between affix values within a cell, the relation between the affix values at a packing node and those at its sons can only be approximated. We first consider an approximation which does not take into account the relation between the different affix positions at each node.

Assume the packing node in Figure 6.4. When eventually the parse forest has been transformed into a decorated parse tree, then the packing node has been eliminated and one of the sons of the packing node, say the j-th, has become a son of the father of the packing node. Additionally, the variables α and α_j are merged into one.

At this moment however, we do not know yet which son of the packing node is going to be selected, but instead of saying that $\alpha = \alpha_j$ for one particular unknown j, we can say that $\alpha_i \subseteq \alpha$, for all i such that $1 \leq i \leq m$, and that $\alpha \subseteq \alpha_1 \cup \ldots \cup \alpha_m$.

If the arity of A had been larger than 1, then a separate set of equations would have been introduced for the second, third, etc. affix positions. Similar equations can be made

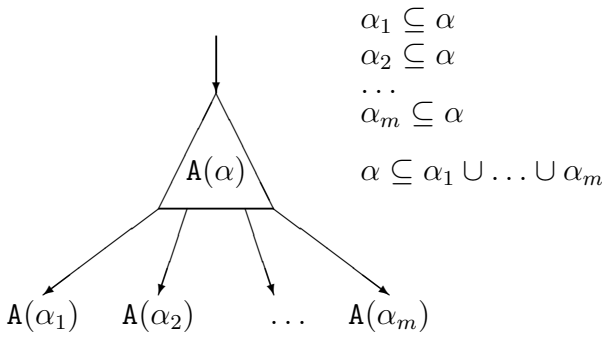


Figure 6.4: The equations for approximating the affix values at a packing node

for sharing nodes. (The situation there is mirrored.)

These equations together with the unification of variables and the initial values, as discussed in Section 6.5.1.1, can be solved by a greatest fixed-point computation. The resulting affix values are optimistic, i.e. a value at a node is at least as large as the value at each corresponding node in parse trees represented by the parse forest.

6.5.1.3 Elimination of packing and sharing nodes

Variables should never hold as value the empty set because the empty set indicates that a parse is incorrect. If a variable in a cell of the forest holds $\emptyset$ then we know that that cell cannot be part of any successfully decorated parse tree. Therefore, it is safe to eliminate that cell.

A cell with the property that it cannot be part of any correct tree is suggestively called *dead wood*.

If dead wood includes a packing node (or sharing node, respectively) then the cells containing the sons (or fathers) of that node are also dead wood and must also be eliminated.

A more complicated situation arises if dead wood includes a node which is the son of a packing node in another cell (or conversely, the father of a sharing node in another cell; this case is completely isomorphic to the above case and will therefore not be discussed separately). If at that moment the packing node has more than two sons, then the node is preserved. Because it loses a son, the equations introduced in Section 6.5.1.2 become more determined. This may trigger renewed greatest fixed-point computations to obtain more exact affix values throughout the forest (which may again lead to more dead wood to be eliminated, etc.).

If however the packing node has only two sons exactly one of which is part of dead wood, then more work has to be done. This situation is illustrated in Figure 6.5. The packing node is eliminated and its former son in cell 3, which is not dead wood, takes over the role as son of the father of the packing node. The variables at the corresponding positions in β_1 and β_3 are pairwise unified. The values of the unified variables are formed by the intersection of the values of the original variables. Cells 1 and 3 now become one cell.

Also in this case, renewed greatest fixed-point computations may be triggered, because of the unification of the variables in β_1 and β_3.

The alternation of the propagation of affix values with the elimination of dead wood is reminiscent of constraint propagation [Mar92, Nag92a].

6.5.1.4 The complete first part

We now present the algorithm of the first part in pseudo code. During execution of the algorithm, the cells are marked with exactly one of the labels *'alive'*, *'dead'*, and *'removed'*. Initially, each cell is marked with *'alive'* provided none of the unifications in that cell has failed (i.e. none of the unified variables holds the empty set), otherwise it is marked with *'dead'*. Variables may or may not be marked with the label *'changed'*.

The main procedure is given below.

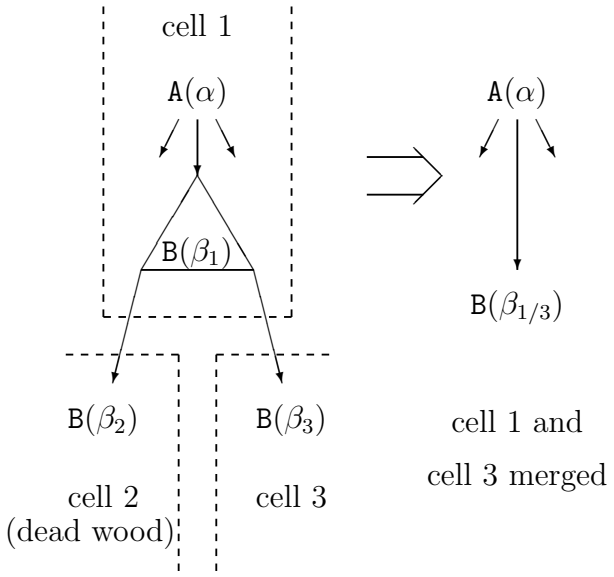


Figure 6.5: Eliminating a packing node. Cell 2 is eliminated as dead wood. Cells 1 and 3 are merged by eliminating the packing node. The corresponding variables in β_1 and β_3 are unified, resulting in $\beta_{1/3}$. (β_1, β_2, β_3, and $\beta_{1/3}$ represent lists of zero or more variables.)

proc *reduce the forest*:
unify and initialise the variables within each cell (Section 6.5.1.1);
determine the approximating equations at the packing and sharing nodes (Section 6.5.1.2);
make every variable 'changed';
make the forest consistent.

The realisation of the first three actions of this procedure is straightforward and is not discussed further. The iterative process of the calculation of affix values and the elimination of dead wood is expressed by the following.

proc *make the forest consistent*:
while *there are 'changed' variables or 'dead' cells*
do *update variables*;
remove dead wood
end.

The evaluation of affix values is a simple greatest fixed-point calculation with respect to the approximating equations and the current values of the variables.

proc *update variables*:
while *there is a 'changed' variable β in an 'alive' cell*
do *make β not 'changed'*;
for each *equation $\alpha \subseteq \alpha_1 \cup \ldots \cup \alpha_m$*
such that *β is some α_i, where $1 \leq i \leq m$, and α is contained in an 'alive' cell*
do *re-evaluate* ($\alpha \subseteq \alpha_1 \cup \ldots \cup \alpha_m$)
end
end.

proc *re-evaluate* ($\alpha \subseteq \alpha_1 \cup \ldots \cup \alpha_m$):
  $\alpha' := \alpha \cap (\alpha_1 \cup \ldots \cup \alpha_m)$;
  if $\alpha' = \emptyset$
  then *make the cell to which* α *belongs 'dead'*
  elseif $\alpha' \neq \alpha$
  then $\alpha := \alpha'$;
    make α *'changed'*
  end.

The elimination of dead wood is given by:

proc *remove dead wood*:
  while *there is a 'dead' cell c*
  do for each *'alive' cell* c' *which*
    • *contains a son of a packing node in c; or*
    • *contains a father of a sharing node in c*
    do *make* c' *'dead'*
    end;
    for each *packing node* N' *in an 'alive' cell* c' *which has a son N in c*
    do *reduce the number of sons of* N'
    end;
    for each *sharing node* N' *in an 'alive' cell* c' *which has a father N in c*
    do *... the converse of the above ...*
    end;
    make c 'removed';
    make all variables in c not 'changed';
  end.

which uses the following two local routines:

reduce the number of sons of N':
  if N' *has exactly one other son* N'' *in an 'alive' cell* c''
  then *merge* c' *and* c'' *into a new 'alive' cell*
  elseif N' *has more than one other son in 'alive' cells*
  then *replace equations*
  end.

replace equations:
  for each *equation* $\alpha \subseteq \alpha_1 \cup \ldots \cup \alpha_m$ *induced by* N'
  do let $\alpha'_1, \ldots, \alpha'_{m'}$ **be** *the variables among* $\alpha_1, \ldots, \alpha_m$ *which occur in 'alive' cells*;
    replace $\alpha \subseteq \alpha_1 \cup \ldots \cup \alpha_m$ *by* $\alpha \subseteq \alpha'_1 \cup \ldots \cup \alpha'_{m'}$;
    re-evaluate $(\alpha \subseteq \alpha'_1 \cup \ldots \cup \alpha'_{m'})$
  end.

The process of merging two cells ("*merge* c' *and* c'' *into a new 'alive' cell*") has already been described to some extent in Section 6.5.1.3. In the context of the above, it must be mentioned that the new cell is made *'dead'* right away if unification of the variables at the

corresponding positions at N'' and N' yields a variable with the value $\emptyset$. Otherwise the unified variables are made *'changed'* if one of the original ones was or if unification has led to a more restricted value than one of the original two values.

The algorithm above abstracts from the order in which *'changed'* variables are handled by the procedure "*update variables*". To minimize the number of times variables have to be visited, practical implementations of "*update variables*" should exhibit the following behaviour.

First, the forest is to be visited from bottom to top. That is, *'changed'* variables in cells which occur further from the root of the forest should be treated before those in cells which occur nearer to the root. After that, the forest is to be visited from top to bottom. At this moment, none of the variables in *'alive'* cells is *'changed'*. Therefore, complete execution of "*update variables*" can be achieved with at most one re-evaluation of each equation. (We assume that a call to "*re-evaluate* ($\alpha \subseteq \alpha_1 \cup \ldots \cup \alpha_m$)" is postponed until every *'changed'* variable α_i ($1 \leq i \leq m$) has been treated.)

Note that the number of times "*update variables*" is called is bounded by the number of times "*remove dead wood*" is called, which is bounded by the number of cells in the initial forest.

If the context-free forest is constructed by a bottom-up parsing algorithm, it may be advantageous to combine the first bottom-to-top pass of "*update variables*" with the context-free parsing, which establishes a synthesis with the first method discussed in Section 6.4. In this way, many incorrect derivations can be discarded at an early stage.

6.5.2 The second part

The algorithm of the first part does not take into account the relation between the different affix positions at each node. The consequence is that not all dead wood can be detected and removed.

Example 6.5.1 Assume the following grammar.

```
SMALL ::  1; 2.
BIG   ::  3; 4.

S: A(SMALL,BIG), B(SMALL,BIG).

A(1,3):  A1.
A(2,4):  A2.

B(2,3):  B1.
B(1,4):  B2.

A1:  "a".
A2:  "a".
B1:  "b".
B2:  "b".
```

A parse forest for the input "`ab`" is given in Figure 6.6. The affix values indicated are those resulting from the algorithm in the previous section. It is however clear that no

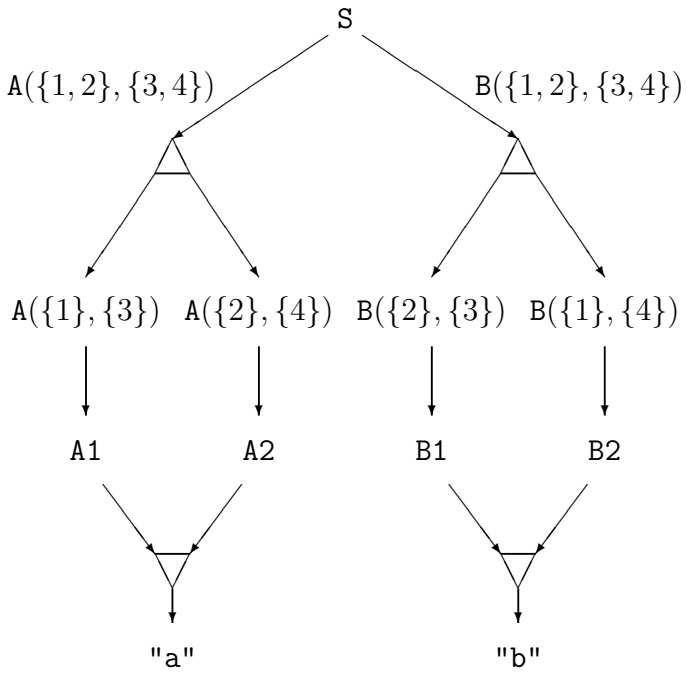


Figure 6.6: The affix values computed by the algorithm from Section 6.5.1

correctly decorated parse tree can be found; the complete forest is dead. Yet the algorithm does not succeed in detecting any dead wood, because it does not take into account the relation between the first and second affix positions of `A` and `B`. □

In [DNS92] we discussed the situation in which the algorithm from the previous section is combined with human-computer interaction in order to find a single decorated tree in a forest. Because the algorithm cannot find all dead wood by itself, more communication with the user is required than it would be by an algorithm which calculated the affix values taking into account the relation between the different affix positions at each node.

In this section we introduce such an algorithm. It calculates sets of *tuples* of affix terminals at each node, instead of sets of affix terminals at each position at each node. The former calculation is of course much more expensive than the latter one. The new algorithm can however be made feasible by reducing the amount of tuples it has to operate on. This is done by previously executing the algorithm from Section 6.5.1. For what correctness is concerned the new algorithm can however be executed independently.

Before presenting this algorithm, we first discuss some important aspects.

6.5.2.1 Tables within a cell

We define an *interface node* of a cell to be a node in that cell which has a son or father in an adjacent cell. In the algorithm of Section 6.5.1, the only internal property of cells relevant to the determination of affix values was the unification of variables at the different interface nodes.

In the algorithm we have set out to define, we also need the relation between the different affix positions at each interface node. The combinations of affix terminals which are possible

Figure 6.7: The tables associated with the nodes in a stub, and the extra *central* table associated with the stub itself. Consistency of the tables at the nodes is achieved via the central table in which all variables in the stub are represented. (The associated production may be of the form `A (X,Y,Z): B (X,Y), C (Y,P,Q), D (Z,P).`; $\alpha, \beta, \ldots$ denote variables; tuples such as (α, β, γ) denote tables containing those tuples of affix terminals which constitute possible combinations of values.)

at an interface node are represented by a set of tuples of affix terminals. We will call such a set a *table*. The algorithm we discuss in this section does not propagate affix terminals from one variable to the other, but tuples of affix terminals from one table to the other.

Because the relationship between different interface nodes in a cell may be very complicated, we rely on the internal structure of the cell. This means that we associate a table to each node in the cell, not only to the interface nodes, and that the propagation of affix values between different tables of a stub is guided by the associated production. To simplify the communication between tables of a stub, we introduce one extra *central* table at each stub.

As an example consider the stub in Figure 6.7 which occurs somewhere within a cell. At each of the nodes of the stub we have one table containing tuples of affix terminals, each of which represents one allowed combination of values for the corresponding variables. In the central table, all variables occurring in the stub are represented.

Initially, each table consists of the Cartesian product of the values of the corresponding variables. These values may result directly from the unification process, described in Section 6.5.1.1, or from the complete first part, described in Section 6.5.1.

Communication between the tables at the nodes of a stub is achieved via the central table. For each node in the stub, we have two equations relating the set of tuples in the associated table with those in the central table.

In the running example we have for instance the two equations

$$\begin{aligned} (\alpha, \beta, \gamma) &\subseteq \{(x, y, z) \in A_t^3 \mid \exists_{pq}[(x, y, z, p, q) \in (\alpha, \beta, \gamma, \delta, \eta)]\} \\ (\alpha, \beta, \gamma, \delta, \eta) &\subseteq \{(x, y, z, p, q) \in A_t^5 \mid (x, y, z) \in (\alpha, \beta, \gamma)\} \end{aligned}$$

which relate the table at the node labelled `A` with the central table. (Here, (α, β, γ) and $(\alpha, \beta, \gamma, \delta, \eta)$ denote the tables containing all allowed tuples of affix terminals for the corresponding variables.)

Figure 6.8: The tables after the optimizations. The dotted lines labelled with "=" indicate that the three tables labelled (α, β) are actually implemented by just one table.

The tables and the equations relating them may be constructed more efficiently as follows. If two nodes in a stub are associated with the same set of variables, then only one table is needed for those two nodes. (Similarly, the extra central table is not needed if one of the nodes in the stub is associated with all the variables in the stub.)

The tables and the equations relating them can also be optimized by splitting tables: a table representing a number of variables can be split into a number of smaller tables such that the variables represented by those new tables constitute a partition of the variables represented by the old table.

The basis for this optimization is a partition into classes of all variables in a cell. This partition should be the finest partition such that variables which occur at the same interface node are in the same class. This means that variables which are in different classes are unrelated by the interface nodes. Each variable at a node is now represented in one of a number of tables at that node, according to its class. (Similarly, each variable occurring in a stub is represented in one of a number of central tables, according to its class.)

For the running example this has the following implications. If we assume that the pair α and β, the pair δ and η, and the pair γ and δ occur at the same interface nodes, and that no other connections exist between these five variables, then the partition consists of the classes $\{\alpha, \beta\}$ and $\{\gamma, \delta, \eta\}$ and both optimizations above lead to the situation in Figure 6.8.

The *width* of tables, i.e. the number of variables they represent, plays an important role in the efficiency of the algorithm. The reason we use the internal structure of cells to build a structure consisting of tables and equations is that this method guaranties an upper bound for the largest width, viz. the largest number of affix nonterminals in a production.

It is an open problem whether there is a polynomial algorithm which determines the optimal structure ("optimal" in terms of the minimal widths) of tables and equations given the interface nodes and possibly the internal structure of the cell.

6.5.2.2 Interfaces to other cells

Tables in one cell are kept consistent with tables in other cells via equations similar to those presented in Section 6.5.1.2. The difference is that for the algorithm in this section, the equations range over tables containing sets of tuples instead of over variables containing

sets of affix terminals.

6.5.2.3 Interaction with the user

Variables or tables holding the empty set are not the only means by which dead wood can be created. It is also possible that the user selects one or more sons of a packing node (or fathers of a sharing node) and thereby artificially declares the cells containing the other sons to be dead wood.

This interaction is required to eliminate all the packing and sharing nodes from the forest, a task which cannot be done merely by means of the calculation of greatest fixed-points and the elimination of dead wood. The human-computer interaction and the mechanized manipulation of the forest alternate until a single decorated parse tree remains.[5]

In actual implementations the selection of sons of packing nodes and fathers of sharing nodes may be realised by providing a graphical representation of some part of the parse forest containing such a node. The user may then select one or more sons or fathers, respectively. There are many possible ways to determine the order in which the different packing and sharing nodes are to be treated, either directed by the user or by mechanized heuristics.

An alternative to a graphical representation of parts of the parse forest is the use of textual queries, one for each packing and sharing node. The form of these queries is determined by attaching extra information to each production of the grammar, as was suggested in [Tom86].

Working systems using human-computer interaction for disambiguation have been described in [BN90, Mar92]. An alternative to human-computer interaction is mechanical preference calculation [Nag92a].

6.5.2.4 The complete second part

We now present the complete algorithm of the second part, which is very similar to the first part. Again, cells are marked with exactly one of the labels *'alive'*, *'dead'*, and *'removed'*. Tables may or may not be marked with the label *'changed'*.

proc *find a single parse tree in a forest*:
  initialise the tables within each cell and create the equations between them (Section 6.5.2.1);
  determine the approximating equations at the packing and sharing nodes (Section 6.5.2.2);
  make every table 'changed';
  make the forest consistent;
  while *there are packing or sharing nodes left in 'alive' cells*
  do *let the user select sons of a packing node or fathers of a sharing node*;
    make the cells containing the other sons or fathers 'dead';
    make the forest consistent
  end.

[5] An exception may be made for the packing nodes which result from disjunction in guards, as in Example 6.2.1, because such packing nodes do not represent actual ambiguity.

The iterative process of updating tables and eliminating dead wood is expressed by the following.

proc *make the forest consistent*:
  while *there are 'changed' tables or 'dead' cells*
  do *update tables*;
    remove dead wood
  end.

The evaluation of tuples in tables is a straightforward greatest fixed-point calculation.

proc *update tables*:
  while *there is a 'changed' table T in an 'alive' cell*
  do *make T not 'changed'*;
    for each *equation $T' \subseteq E$*
      such that *expression E contains T and table T' is contained in an 'alive' cell*
    do *re-evaluate* $(T' \subseteq E)$
    end
  end.

proc *re-evaluate* $(T \subseteq E)$:
  $T' := T \cap E$;
  if $T' = \emptyset$
  then *make the cell to which T belongs 'dead'*
  elseif $T' \neq T$
  then $T := T'$;
    make T 'changed'
  end.

The elimination of dead wood is almost the same as in the algorithm of the first part (Section 6.5.1.4), apart from the fact that in this section the equations involve tables instead of variables.

Similar to "*update variables*" in Section 6.5.1.4, "*update tables*" can be implemented by one bottom-to-top pass followed by a top-to-bottom pass through the forest. If "*remove dead wood*" is combined with the updating of tables, then even the complete procedure "*make the forest consistent*" can be implemented by a single pair of passes (contrary to "*make the forest consistent*" in Section 6.5.1.4).

In this case, such a pair of passes is needed every time the user discards a collection of cells.

6.6 Complexity of the algorithm

The complete algorithm for finding a decorated parse tree in a context-free parse forest as described in Section 6.5 requires $\mathcal{O}(n)$ time, where n is the number of arrows in the parse forest (or alternatively, the number of nodes, which is of the same order as argued in Section 6.3). This can be explained as follows.

6.6.1 Complexity of the first part

The algorithm of the first part, as given in Section 6.5.1, is the less costly of the two parts. The most costly calculation is that of the greatest fixed-points. The sum of the lengths of the right-hand sides of equations of the form $\alpha \subseteq \alpha_1 \cup \ldots \cup \alpha_m$, where $m > 1$, or of the form $\alpha_i \subseteq \alpha$ is $\mathcal{O}(p \times n)$, where p is the largest arity of all nonterminals.

A variable holds at most q different affix values during execution of the algorithm, where q is the size of the largest domain. (Note that the values can only get further restricted.) We assume that affix terminals in a variable can be accessed in a constant amount of time.

If we implement the current values of the right-hand sides $\alpha_1 \cup \ldots \cup \alpha_m$ using bags, then the evaluation of a new value after the value of say α_i $(1 \leq i \leq m)$ is reduced by one element takes a constant amount of time.

We may conclude that the time complexity for the calculation of the greatest fixed-points is $\mathcal{O}(q \times p \times n)$.

The time complexities of the construction and merging of cells, including unification of variables, and the elimination of dead wood are trivially also $\mathcal{O}(q \times p \times n)$. Therefore the total time complexity of the first part is $\mathcal{O}(q \times p \times n)$, or $\mathcal{O}(n)$ not considering properties of the AGFL.

6.6.2 Complexity of the second part

The algorithm of the second part, as given in Section 6.5.2, is more costly.

In this algorithm, equations are of the form $T \subseteq T_1 \cup \ldots \cup T_m$, where $m > 1$, $T_i \subseteq T$, or $T \subseteq \{(x_1, \ldots, x_m) \in A_t^m \mid (y_1, \ldots, y_{m'}) \in T'\}$, where one of the sets of variables $\{x_1, \ldots, x_m\}$ and $\{y_1, \ldots, y_{m'}\}$ is a subset of the other. The sum of the lengths of (the number of tables in) the right-hand sides of these equations is $\mathcal{O}(n)$. This is of course under the assumption that none of the optimizations from Section 6.5.2.1 for the construction of tables and equations has been applied.

Tables contain at most $\mathcal{O}(q^{p'})$ tuples, where p' is the largest number of affix nonterminals in a production, and therefore their values may change at most $\mathcal{O}(q^{p'})$ times.

We assume that tuples in a table can be accessed in a constant amount of time and that the current values of the right-hand sides $T_1 \cup \ldots \cup T_m$ are implemented using bags. The time complexity related to equations of the first and second forms above is therefore $\mathcal{O}(q^{p'} \times n)$.

More serious to the overall time complexity are the equations of the form $T \subseteq \{(x_1, \ldots, x_m) \in A_t^m \mid (y_1, \ldots, y_{m'}) \in T'\}$. Calculation of the tuples by which T is reduced from a tuple by which T' is reduced costs $\mathcal{O}(q^{|m-m'|})$ time, which is $\mathcal{O}(q^{p'-1})$.

The total time complexity is therefore $\mathcal{O}(q^{p'-1} \times q^{p'} \times n) = \mathcal{O}(q^{2p'-1} \times n)$, or $\mathcal{O}(n)$ not considering properties of the AGFL.

If the proposed two-pass implementation of "*make the forest consistent*" is applied, then we can give a different time complexity. During each execution of "*make the forest consistent*" the equations are re-evaluated at most once. Together this costs $\mathcal{O}(q^{p'} \times n)$ time. The number of times "*make the forest consistent*" has to be executed is bounded by the number of times the user discards one or more cells, which is $\mathcal{O}(n)$ times.

The total time complexity is therefore also $\mathcal{O}(n \times q^{p'} \times n) = \mathcal{O}(q^{p'} \times n^2)$.

6.6.3 Practical handling of sets of tuples

The time complexity for the second part as presented in the previous section does not look promising for realistic purposes. The second part operates however on affix values which have already been reduced considerably by the first part. Furthermore, the two optimizations from Section 6.5.2.1 reduce the practical time requirements.

One more optimization should be mentioned in the context of calculating with tuples. The essential observation is that in a parse forest without packing or sharing nodes, all tables contain a Cartesian product of a number of sets. In a parse forest with a small number of packing and sharing nodes, each table contains a union of a small number of Cartesian products.

A smart representation for a Cartesian product of m sets requires $\mathcal{O}(m \times q)$ space for storage, although $\mathcal{O}(q^m)$ tuples may be represented. Even for a union of Cartesian products a smart representation may be found which requires less space than the straightforward representation using separate storage for each of the represented tuples.

In the Grammar Workbench for AGFLs (Chapter 7) these considerations have influenced the design of the grammar verifier. Sets of tuples are represented as sets of tuples of sets, which represent unions of Cartesian products. For example, the set of tuples

$$\begin{array}{rcl} \{(a,b,d) & , & (a,b,e)\ , \\ (a,c,d) & , & (a,c,e)\ , \\ (p,x,y) & , & (q,x,y)\} \end{array}$$

can be represented by the set of tuples of sets

$$\{(\{a\},\{b,c\},\{d,e\}) \quad , \quad (\{p,q\},\{x\},\{y\})\}$$

This form can be obtained by first taking the following set of tuples of singleton sets:

$$\begin{array}{rcl} \{(\{a\},\{b\},\{d\}) & , & (\{a\},\{b\},\{e\})\ , \\ (\{a\},\{c\},\{d\}) & , & (\{a\},\{c\},\{e\})\ , \\ (\{p\},\{x\},\{y\}) & , & (\{q\},\{x\},\{y\})\} \end{array}$$

and by merging pairs of tuples. E.g. $(\{a\},\{b\},\{d\})$ and $(\{a\},\{b\},\{e\})$ can be merged into $(\{a\},\{b\},\{d,e\})$. Repeated application of merging leads to the desired form.

The union of two sets represented in this form is complicated by the fact that two tuples may overlap. E.g. if we want to compute the union of $\{(X,Y,Z)\}$ and $\{(P,Q,R)\}$, where X,Y,Z,P,Q,R denote sets of affix terminals, then we may have that $X \cap P \neq \emptyset$ and $Y \cap Q \neq \emptyset$ and $Z \cap R \neq \emptyset$. A solution is to mould e.g. $\{(P,Q,R)\}$ into a different form:

$$\begin{array}{l} \{(P-X\ ,\ Q\ \ \ \ \ \ ,\ R\ \ \ \ \ )\ , \\ (P\cap X\ ,\ Q-Y\ ,\ R\ \ \ \ \ )\ , \\ (P\cap X\ ,\ Q\cap Y\ ,\ R-Z)\ , \\ (P\cap X\ ,\ Q\cap Y\ ,\ R\cap Z)\} \end{array}$$

Possibly some of the tuples in this form may be discarded because one or more of their parts may be the empty set.

The union of the two sets is now represented by:

$$\begin{array}{l}\{(X \quad , Y \quad , Z \quad) , \\ \ (P - X \ , Q \quad , R \quad) , \\ \ (P \cap X \ , Q - Y \ , R \quad) , \\ \ (P \cap X \ , Q \cap Y \ , R - Z)\}\end{array}$$

Our representation of sets of tuples has proved to be of considerable use in the analysis of large grammars. Many of the sets of tuples which have been calculated by the Grammar Workbench would not even fit into the memory of the computer that it operates on if all tuples of affix terminals were represented by a separate unit of memory.

What makes this representation particularly suitable to the decoration of parse forests is that the sets of tuples of sets do not have to be constructed by merging sets of tuples of singleton sets. Instead, the initialisation of the tables yields the desired representation straightforwardly.

A problem with this representation is that it can become fragmented considerably because of the union and subtraction operations, as indicated above. In other words, the union of two sets consisting of k and m tuples of sets, respectively, may yield a set consisting of more than $k + m$ tuples of sets, in general less than or equal to $k + m \times t^k$ tuples, where t is the width of the tuples.

To prevent quick degeneration of this representation to sets of tuples of singleton sets, it is necessary to regularly apply the merge operation, which reduces the number of tuples (possibly after deliberately fragmenting the representation further to allow more extensive merging).

We have found in the Grammar Workbench that the costs of merging tuples of sets do not outweight the high costs of operating on tuples of affix terminals. It required some experimenting however to determine how often the expensive merging operation should be applied to improve instead of deteriorate the calculation time. In some cases, allowing temporary overlapping of tuples in a set proved to be advantageous to the time costs.

A representation of sets of tuples similar to ours is proposed in [ZLC94]. This representation, called *sharing trees*, may be more compact than ours, and has the advantage that a unique sharing tree exists for each set of tuples, which simplifies the test for equivalence. However, sharing trees were developed with a certain application in mind (analysis of synchronized automata), and it is not clear yet whether for our case sharing trees would behave any better than the above representation, using sets of tuples of sets. This is because the analysis of AGFLs requires different operations (such as subtraction), some of which are more complicated for sharing trees than for our representation.

6.7 Discussion

The goal of our research is to invent a reasonably efficient system which can assist a linguist in finding the intended parse for each sentence in some corpus. The parse trees should be constructed according to a fixed AGFL.

As natural languages are inherently ambiguous, AGFLs describing natural languages are also ambiguous. Therefore, no automated system can perform the task of finding the intended parse of a sentence, based on an AGFL alone.

We have described part of an implementation which takes a context-free parse forest produced by some context-free parsing algorithm and turns it into a single decorated parse tree. During this process the system consults the user for the resolution of ambiguities.[6]

For reasons of efficiency, our implementation consists of two parts. The second part constitutes the actual calculation of the affixes, and the resolution of ambiguities by means of interaction with the user. The first part merely serves to filter out the affix values which can easily be found inconsistent, thereby saving work for the second, more complicated part.

We have proposed some optimizations, which turn our method into a very efficient algorithm for the decoration of parse forests according to AGFLs and related formalisms: it is efficient in storage costs because at every moment the amount of space needed is linear in the size of the forest. It is efficient in time costs because it makes optimal use of the sharing and packing established by the context-free parsing algorithm.

Future research will show how the algorithm can be fine-tuned. In particular, the special representation of sets of tuples will need to be investigated further to determine how often and in what way the merge operation should be applied.

It is an intriguing question whether a polynomial algorithm exists to determine the optimal structure in a cell consisting of tables and equations relating them.

[6]One may consider the possibility that the resolution of ambiguities in our system could equally well be supported by *mechanized* semantic and pragmatic analyses.

Chapter 7

A customized grammar workbench

In this chapter we describe the ideas behind the Grammar Workbench (GWB). The GWB is one of a series of tools for the development of AGFLs (affix grammars over finite lattices) for natural languages. Its functions comprise a specialised editor, computation of properties, a random generator of sentences, and special functions to provide an overview of a grammar.

This chapter discusses the functions of the GWB, the grammatical formalism of AGFL, and the AGFL project. We also discuss the relationship between the complete development environment for AGFLs and other development environments, both for other grammatical formalisms and for computer programs.

7.1 Introduction

Formal grammars for natural languages tend to become unmanageable as they get larger. Similar problems occur in the development of large computer programs. To overcome these problems with large programs, a number of techniques have been invented in the field of software engineering to

- structure programs so as to permit decomposition into smaller, more manageable units (modules, subroutines, etc.), and to
- allow properties of programs to be mechanically determined (static analysis, symbolic execution, integration testing, etc.)

to mention just a few. An early paper which discusses how tools and modularity can alleviate the complex task of developing large programs is [Win84].

In the AGFL project, some of the techniques from software engineering have been incorporated in the grammatical formalism of AGFL and its tools, and especially in the Grammar Workbench:

- AGFLs can be divided into modules with a clearly defined interface to other modules (Section 7.2.2);
- the GWB allows analysis of AGFLs in various ways (Section 7.5);

- the GWB allows random sentences to be generated from a nonterminal. The purpose of this function is to provide some intuition of the meaning of the nonterminal and to reveal overgeneration (Section 7.6);
- future versions of the GWB will have the possibility to trace the parsing process (Section 7.7). Furthermore, a tool is currently available which allows the investigation of tree structures (see the discussion below).

The complete implementation of AGFLs is meant to consist of three tools:

1. the Grammar Workbench (GWB)
2. the Parser Generator (GEN)
3. the Linguistic Database (LDB)

The Linguistic Database of van Halteren and van den Heuvel [vHvdH90] is a tool for storing and investigating large numbers of tree structures of various kinds. Although the LDB has many purposes, the one we are interested in here is the ability to store and investigate parse trees which have resulted from the parsing of corpora according to AGFLs.

Two variants of the Parser Generator have been under investigation. The first constructs backtrack parsers from AGFLs. The second constructs two cooperating programs: a *generalised LC* parser (Chapter 2), which produces parse forests, and a program which decorates parse forests (Chapter 6).

The Grammar Workbench is a tool for editing and investigating AGFLs. Its functions are the main subject of this chapter.

In Figure 7.1 we outline how the GWB, GEN, and LDB are intended to interact in the development of AGFLs. The development of grammars is an iterative process. Investigation of the result of parsing a corpus yields the information needed to improve the grammar in such a way that in the next iteration the grammar fits more closely to the corpus.

Other development environments for various grammatical formalisms have been constructed. TAGDevEnv [Sch88] is a development environment for tree adjoining grammars. An environment for developing augmented phrase structure grammars, called METAL, is described in [Whi87].

Two development environments have been constructed for formalisms related to generalised phrase structure grammars. One is called ProGram [Eva85], the other is called GDE [BGBC87, BCBG88, Bog88].

D-PATR [Kar86] is a development environment for PATR, which is a formalism suitable for encoding a wide variety of grammars, especially unification-based grammars. A development environment called ud is described in [JR89] and implements a formalism related to PATR.

The TFS (typed feature structure) constraint language is implemented by a number of tools described in [Gri92]. The ATN programming environment (APE) [HG86] implements augmented transition network grammars.

In [Wah84] some general remarks on the concept of a workbench for linguists can be found.

The structure of this chapter is as follows. A general discussion of the GWB can be found in Section 7.3. Sections 7.4 to 7.7 discuss the various functions of the Grammar Workbench

Figure 7.1: The interaction between GWB, GEN, and LDB.

and the way these functions are realised in the other development environments discussed above. In the next section we first define the grammatical formalism of AGFL.

For a more extensive discussion of the Grammar Workbench see [NKDvZ92, DKNvZ92].

7.2 Affix grammars over finite lattices

Affix grammars over finite lattices (AGFLs) have already been described in Section 6.2 in a simplified form. In this section we describe some aspects not mentioned before.

7.2.1 More shorthand constructions

In Section 6.2 we stated that the rhs of a rule of the meta grammar consists of one affix terminal. In general AGFLs, each rhs may also consist of an affix *non*terminal. The domain of an affix nonterminal N defined by $N :: X_1; \ldots; X_m.$ is now defined to be the union of

the set of affix terminals in $\{X_1, \ldots, X_m\}$ and the domains of all affix nonterminals in $\{X_1, \ldots, X_m\}$.

An example of a shorthand construction in productions is that a number of alternatives may be grouped together as a single member in a larger alternative.

7.2.2 Modularity in AGFLs

Modular programming is a well established and essential concept in software engineering. A large piece of software can only be kept manageable if it is split up into parts in such a way that each of the parts can be understood without knowledge of details of other parts.

In a similar way, and following e.g. [AFL87, VB85, Gri92], we have introduced the concept of modularity into a grammatical formalism. An AGFL is physically composed of a number of files. Each file contains one module and begins with a module heading. This heading contains among other things

- a list of all modules from which objects are imported,
- a list of all objects which are exported from the module.

The objects in the second list can be written in the form of a left side of a production, i.e. with a display. The purpose of the display is twofold. First, the export-list now provides more explicit information on the interfaces between modules. Second, an expression $N(A_1, \ldots, A_i)$ in an export-list can be seen as a statement that it is possible to derive some string from $N(a_1, \ldots, a_i)$ for every list of affix terminals $a_1, \ldots, a_i$, such that a_j is in the domain of A_j, for $1 \leq j \leq i$, and conversely, that it is not possible to derive some string from $N(a_1, \ldots, a_i)$ if a_j is not in the domain of A_j, for some j such that $1 \leq j \leq i$.

This completeness and consistency condition can be mechanically checked, and this is implemented in the GWB. Except in the case of predicates, if a proper display $(A_1, \ldots, A_i)$ cannot be found such that the above condition holds, then this usually indicates a mistake in the design of the grammar.

7.2.3 Describing languages using AGFLs

Extended affix grammars (EAGs), which where already mentioned above, have been used in three projects describing natural languages:

- English [Oos91]
- Spanish [Hal90]
- Modern Standard Arabic [Dit92]

EAGs can describe all languages of type 0 of the Chomsky hierarchy, whereas AGFLs can only describe context-free languages because of the finiteness of the domains.

Thus EAGs provide a much stronger formalism than AGFLs. It is however shown in practice that the greater power is seldom (if ever) used to describe aspects of a natural language which are not context-free.

Instead, the greater power of EAGs (and equally powerful formalisms) causes many grammar writers to conceive of the formalism as a programming language, and consequently the structure of their grammars may be motivated more by procedural considerations than by linguistic ones. Following [Kap87] we argue that such grammars are hard to understand, because the actual information which a grammar is supposed to convey, viz. the structure of a natural language, is clouded by irrelevant information of a procedural nature. Furthermore, powerful formalisms such as EAGs do not allow as much flexibility in the parsing scheme as for instance the formalism of AGFL does.

At the moment, the above-mentioned EAGs are all being rewritten into AGFLs. There have also been some feasibility studies concerning the development of original AGFLs:

- Hungarian [FKKN91]
- Turkish [KW91]
- Greek [KM92]

More advanced than these projects is the development of an AGFL for Dutch [vB84, Cop87].

7.3 The ideas behind the Grammar Workbench

The Grammar Workbench is designed for two purposes: to support development of large grammars, and to support the training of students in compiler construction and computational linguistics.

Portability and machine-independence is of paramount importance to a tool which is intended to be used for such diverse applications. Therefore, we have deliberately avoided the use of graphic displays or windows as applied in TAGDevEnv, D-PATR, and APE. However, we follow the example of METAL and GDE to make it possible to extend the system with graphic displays or windows. (See [Tei84] for a discussion of the usefulness of display-oriented tools for the development of programs.) To compensate for the lack of windows, we practice economic use of the screen, and instead of using graphic representation of tree structures we use indentation. This has also been applied in METAL.

Portability of the GWB is enhanced by the implementation language Elan, which is translated into C by a compiler developed at the University of Nijmegen.

As the GWB is also meant to be an educational tool, the user interface is very simple. All commands are activated by pressing only one key. At all times exactly one object is the distinguished object, the *focus*, which is the implicit parameter to most commands. The principle of 'focus' has also been applied in GDE.

In the GWB, modular development is supported by a design of its functions which ensures that they perform correctly in case a grammar is incomplete. Therefore it is possible to use all functions of the GWB in early stages of grammar development when only some modules or parts of modules have been completed. This kind of *incremental* development is also possible with GDE and the TFS tools.

A more primitive function which reflects the incremental spirit can be found in TAGDevEnv: the lexicon can be extended during parsing, if some word in the input cannot be found by the lexical analyser.

7.4 Changing of grammars

An important requirement to any development environment for grammars is the comfortable editing of grammars and parts of grammars.

The GWB allows editing of the definition of the focused object. Similar to GDE, the GWB performs incremental syntax checks, i.e. if a grammar definition which is being read from a file or which has just been edited contains a syntactic error, then the editor is automatically invoked. The cursor points to the offending position, so that the mistake can be corrected immediately.

Often a new production is similar to an existing one. For this purpose the GWB allows 'input by analogy': new productions can be inserted by selecting alternatives from existing definitions. Copies of these productions may subsequently be edited and inserted into the grammar. Similar functions are implemented in TAGDevEnv and METAL.

Besides these simple ways to change grammars, also some standard *transformations* are available in the GWB which preserve the generated language. Some examples are (un)folding and left-factoring. These transformations are useful for pedagogical purposes and can help to transform grammars into a form which allows more efficient parsers.

TAGDevEnv, D-PATR, and APE incorporate graphic editors. To compensate the lack of graphic displays, the GWB incorporates an excellent pretty-printer. It makes sure that the indentation and the places where a production is broken up into more than one line reflect the hierarchical structure. Note that the hierarchical structure of AGFLs may be complicated because of nested alternatives.

We think that a skilfully pretty-printed production provides as much insight into its structure as any graphic representation. Furthermore, editing a pretty-printed text does not require any practice, in contrast to manipulating graphic structures on a graphic display.

7.5 Analysis of grammars

One of the most demanding requirements of an environment for the development of grammars is that it provides as much information as a user might need to keep track of a grammar during its development.

The information provided by generation of random sentences and by tracing the parsing process is discussed in Sections 7.6 and 7.7. In this section we restrict ourselves to static information.

GDE allows monitoring the effects of grammar expansion and compilation. TAGDevEnv and METAL possess functions which compute certain properties of a grammar and give diagnostics if any of the properties violates the contextual requirements of the grammatical formalism.

The information provided by the GWB is more extensive and can be divided into three classes:

- *inconsistencies*, i.e. aspects of a syntactically correct grammar which are considered to be in conflict with various well-formedness conditions;
- properties such as reachability, starters, the LL(1) property, left recursion, and the nonfalse property; and

- information on the structure of the grammar as a whole, such as the *is-called-by* relation.

Some examples of inconsistencies are missing definitions, wrong use of objects, and violation of the permitted and necessary affix values as specified in the export-list (Section 7.2.2).

A high-level view of the grammar is provided by a function which provides the is-called-by hierarchy of nonterminals in a compact form, using appropriate indentation. In a similar way, there is a function which outputs a minimal collection of paths needed to explain the non-LL(1)-ness of a nonterminal. The output is very clear and therefore useful for educational purposes.

The computation of properties such as starters is realised by first deriving equations from the productions. The properties are then determined by the least solution to these equations. In the GWB this solution is computed by an *incremental* algorithm. This means that if the grammar is changed, only those properties which are affected are recomputed.

This incremental algorithm, described in [NKDvZ92], is relatively simple: the initial evaluation proceeds by iteratively propagating values through the variables in the equations until a stable situation arises. Whenever an equation is added, this process is resumed, so that the new solution is obtained by taking advantage of the already computed results. Removing an equation however has more serious consequences: all directly or indirectly affected variables first have to be reinitialised to the empty set before the new solution can be computed by renewed iterations of the propagation of values.

Although this is a very simple way of performing incremental evaluation, we have found that it has been advantageous to the usefulness of the GWB: a system which regularly remains incommunicable for a long time while computing requested information puts a heavy strain on the patience of the user. Therefore, a small gain in response time already constitutes an important increase in the usefulness of an interactive system.

Whether more sophisticated methods of incremental evaluation, such as described in e.g. [Zad84], may lead to faster results is the subject of further research.

A related topic is the caching applied in GDE, which optimizes re-expansion of meta-grammars.

7.6 Automatic generation of sentences

A writer of a grammar is often confronted with the problem of finding out whether the language his grammar generates is what he intended it to be.

An obvious way to explore some aspects of the grammar is to run a parser generated from the grammar on a corpus of sentences. Sentences which are unexpectedly rejected may reveal *undergeneration* (the grammar generates fewer sentences than it should). The location of the imperfections of the grammar can easily be found if some kind of tracing of the parsing process is supported (Section 7.7).

Contrary to Evans [Eva85] however, we feel that this process is not appropriate to investigate *overgeneration* (the grammar generates too many sentences). Evans suggests that the grammar developer can detect overgeneration by supplying a parser with incorrect sentences and then observing that the sentences are accepted by the parser. In practice it

is however much too difficult for the user to write a sufficient number of incorrect sentences such that a meaningful part of the overgeneration comes to light.

To allow detection of overgeneration we have therefore included a random generator of sentences and sentence fragments in the GWB, similar to the generator in GDE. A generator to test functional unification grammars has been described in [KK85].

In the GWB, the generator derives a string from the focused nonterminal, making quasi-random choices. Special care is taken to make sure that the generated sentences are long enough to allow interesting linguistic constructs, and short enough to allow insight into the sentence.

Actual generation is preceded by an analysis of the grammar which is performed only once. After the analysis, the generator proceeds without having to resort to backtracking. Therefore, the time needed to derive a single sentence is negligible.

The details of the generation algorithm and the required grammar analysis are discussed in [Ned94a].

7.7 Tracing the parsing process

If a sentence is found to be erroneously generated by a grammar, it is easy to identify the source of the overgeneration: one may simply investigate a mechanically constructed parse tree annotated with affix values which derives that sentence, and locate the incorrect production(s).

If a correct sentence is found to be rejected, then the language generated by the grammar needs to be enlarged by adding new productions or by relaxing some constraints in the form of guards. To find out exactly why the current grammar does not generate some particular sentence may be very difficult without mechanical support. For this purpose most grammar development environments are equipped with some form of tracer which allows interactive parsing.

The kind of interaction possible in these tracers differs considerably. E.g. TAGDevEnv incorporates a tracer where no search paths are investigated autonomously; all decisions have to be made by the user. (Conversely, the parser in TAGDevEnv allows no interaction.)

The tracer in METAL merely prints information on the rule applications, without allowing the user to guide the process.

The tracer in GDE is more flexible. It allows construction of a derivation both guided by the user and autonomously by the generator, already mentioned above. Furthermore, the generator can be parameterised such that the user has a variable amount of influence over the kind of derivation that will be constructed. The tracer in ProGram has similar properties, except that a parser is used instead of a generator, that is, the autonomous tracing mode is guided by input. Parameterisation of the parsing behaviour is also possible in ud.

The tracer in APE provides the user with a graphic representation of the intermediate results of autonomous tracing. At any moment the user may interrupt this process and change the intermediate results. Thereupon autonomous tracing can be reactivated.

If no graphic displays are available, the design of a useful tracer is very difficult. Evans [Eva85] observes that understanding the behaviour of the tracer in ProGram requires some effort. Evans blames this primarily on the intricate compilation process of a generalized

phrase structure grammar which precedes parsing. Our assumption however is that the behaviour of the tracer is so hard to follow because no useful layout has been incorporated in the output of the tracer.

We have considered different kinds of tracers, one of which is to be permanently incorporated in the GWB. An important example to us is the tracer in the Elan programming environment [Kos91b]: in the interactive tracing mode, a call to a procedure is reflected on the screen by (among other things) the name of the procedure and the values of its parameters. This message is indented more than the corresponding message of the calling procedure on the previous line.

The unique property of this tracer is that upon leaving a procedure, the message of the procedure is erased by moving the cursor up and writing the message of the next procedure called by the same calling procedure. Therefore, the position on the screen invariably determines the depth in the calling hierarchy.

The same strategy can be used for tracing the parsing process. In case of top-down parsing, the position on the screen invariably determines the distance from the root of the parse tree.

An extra complication with tracing backtrack parsing is that the difference between backtracking and moving up to process right sisters of the mother node should be reflected graphically. We propose that the messages telling which nonterminals are called with what affix values are not preceded by indentation, but that instead the left margin consist of a primitive reflection of the parse tree constructed so far. Pieces of that diagram are removed only upon backtracking. A message concerning an instance of a nonterminal with certain affix values is printed just to the right of the marker reflecting the corresponding node in the parse tree.

The tracer should also be able to autonomously derive specified parts of the input from specified instances of nonterminals. To secure termination of autonomous parsing, it may be more appropriate to apply bottom-up parsing in the tracer. (Backtracking parsing strategies which always terminate are suggested in Chapters 2, 4 and 5.)

A very simple kind of tracer for AGFLs has been described in [FT93].

7.8 Future developments

At the moment, the GWB implementation consists of about 12000 lines of Elan. Its robustness has been demonstrated by its successful use by students, linguists, and in our own language theoretic research. Ongoing developments will lead to a growing number of available functions. Apart from implementation of the tracer discussed in the previous section and possibly a parser generator, the collection of available transformations will be enlarged.

The current implementation may still be called a prototype. Therefore, many improvements are still waiting to be realised. For example, the need for faster processing of grammars may in the future require the transition to a different implementation language. The current implementation takes between 2 and 3 minutes to calculate most properties of a grammar of about 1000 productions on a SPARC MP670 with moderate load. Most of this time is spent on the calculation of starters, enders and followers, and on the check whether each call of a nonterminal matches some production in its definition and vice versa.

A second possible improvement consists of an extension to the system, which may be activated if the screen supports windows. It has been shown in practice that requested information is often pushed from the screen by subsequently requested information before it loses its interest to the user. Windows may be linked to the GWB in order to raise the lifetime of requested information on the screen.

Bibliography

[AD89] H. Abramson and V. Dahl. *Logic Grammars*. Springer-Verlag, 1989.

[AFL87] L. Appelo, C. Fellinger, and J. Landsbergen. Subgrammars, rule classes and control in the Rosetta Translation System. In *Third Conference of the European Chapter of the ACL* [6], pages 118–133.

[AHU68] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. Time and tape complexity of pushdown automaton languages. *Information and Control*, 13:186–206, 1968.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[AJU75] A.V. Aho, S.C. Johnson, and J.D. Ullman. Deterministic parsing of ambiguous grammars. *Communications of the ACM*, 18(8):441–452, August 1975.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[AU72] A.V. Aho and J.D. Ullman. *Parsing*, The Theory of Parsing, Translation and Compiling, volume 1. Prentice-Hall, 1972.

[Aug93] A. Augusteijn. *Functional Programming, Program Transformations and Compiler Construction*. PhD thesis, Eindhoven University of Technology, 1993.

[Bar93] F. Barthélemy. *Outils pour l'Analyse Syntaxique Contextuelle*. PhD thesis, University of Orléans, 1993.

[BC88] D.T. Barnard and J.R. Cordy. SL parses the LR languages. *Computer Languages*, 13(2):65–74, 1988.

[BCBG88] B. Boguraev, J. Carroll, T. Briscoe, and C. Grover. Software support for practical grammar development. In *COLING '88* [9], volume 1, pages 54–58.

[Bea83] J. Bear. A breadth-first parsing model. In *Proc. of the Eighth International Joint Conference on Artificial Intelligence*, volume 2, pages 696–698, Karlsruhe, West Germany, August 1983.

[BGBC87] T. Briscoe, C. Grover, B. Boguraev, and J. Carroll. A formalism and environment for the development of a large grammar of English. In *IJCAI-87* [12], volume 2, pages 703–708.

[BHPS64] Y. Bar-Hillel, M. Perles, and E. Shamir. On formal properties of simple phrase structure grammars. In Y. Bar-Hillel, editor, *Language and Information: Selected Essays on their Theory and Application*, chapter 9, pages 116–150. Addison-Wesley, 1964.

[Bir80] R.S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–417, December 1980.

[BKR91] L. Banachowski, A. Kreczmar, and W. Rytter. *Analysis of Algorithms and Data Structures*. Addison-Wesley, 1991.

[BL89] S. Billot and B. Lang. The structure of shared forests in ambiguous parsing. In *27th Annual Meeting of the ACL* [3], pages 143–151.

[BN90] R.D. Brown and S. Nirenburg. Human-computer interaction for semantic disambiguation. In *COLING-90* [10], volume 3, pages 42–47.

[Bog88] B. Boguraev. A natural language toolkit: Reconciling theory with practice. In U. Reyle and C. Rohrer, editors, *Natural Language Parsing and Linguistic Theories*, pages 95–130. D. Reidel Publishing Company, Dordrecht, Holland, 1988.

[BPS75] M. Bouckaert, A. Pirotte, and M. Snelling. Efficient parsing algorithms for general context-free parsers. *Information Sciences*, 8:1–26, 1975.

[BR87] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proc. of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 269–283, 1987.

[BS92] H.U. Block and L.A. Schmid. Using disjunctive constraints in a bottom-up parser. In *KONVENS 92* [1], pages 169–177.

[BSS84] D.R. Barstow, H.E. Shrobe, and E. Sandewall, editors. *Interactive Programming Environments*. McGraw-Hill, 1984.

[BVdlC92] F. Barthélemy and E. Villemonte de la Clergerie. Subsumption-oriented push-down automata. In *Programming Language Implementation and Logic Programming, 4th International Symposium*, Lecture Notes in Computer Science, volume 631, pages 100–114, Leuven, Belgium, August 1992. Springer-Verlag.

[BvN93] G. Bouma and G. van Noord. Head-driven parsing for lexicalist grammars: Experimental results. In *Sixth Conference of the European Chapter of the ACL* [7], pages 71–80.

[Car93] J.A. Carroll. Practical unification-based parsing of natural language. Technical Report No. 314, University of Cambridge, Computer Laboratory, England, 1993. PhD thesis.

[Car94] J. Carroll. Relating complexity to practical performance in parsing with wide-coverage unification grammars. In *32nd Annual Meeting of the ACL* [5], pages 287–294.

[CCMR91] M. Chytil, M. Crochemore, B. Monien, and W. Rytter. On the parallel recognition of unambiguous context-free languages. *Theoretical Computer Science*, 81:311–316, 1991.

[CH87] J. Cohen and T.J. Hickey. Parsing and compiling using Prolog. *ACM Transactions on Programming Languages and Systems*, 9(2):125–163, April 1987.

[Cop87] P.A. Coppen. Het AMAZON-algoritme voor werkwoordelijke eindclusters. *GRAMMA*, 11:153–167, 1987.

[CS70] J. Cocke and J.T. Schwartz. *Programming Languages and Their Compilers — Preliminary Notes*, pages 184–206. Courant Institute of Mathematical Sciences, New York University, second revised version, April 1970.

[dC86] D. de Champeaux. About the Paterson-Wegman linear unification algorithm. *Journal of Computer and System Sciences*, 32:79–90, 1986.

[DE90] J. Dörre and A. Eisele. Feature logic with disjunctive unification. In *COLING-90* [10], volume 2, pages 100–105.

[Dem77] A.J. Demers. Generalized left corner parsing. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 170–182, Los Angeles, California, January 1977.

[Den94] Y. Den. Generalized chart algorithm: an efficient procedure for cost-based abduction. In *32nd Annual Meeting of the ACL* [5], pages 218–225.

[Die87] S.W. Dietrich. Extension tables: memo relations in logic programming. In *Proc. 1987 Symposium on Logic Programming*, pages 264–272, San Francisco, August 1987.

[Dit92] E. Ditters. *A Formal Approach to Arabic Syntax — The Noun Phrase and the Verb Phrase*. PhD thesis, University of Nijmegen, 1992.

[DKNvZ92] C. Dekkers, C.H.A. Koster, M.-J. Nederhof, and A. van Zwol. Manual for the Grammar WorkBench Version 1.5. Technical Report no. 92–14, University of Nijmegen, Department of Computer Science, July 1992.

[DMAM94] J. Dowding, R. Moore, F. Andry, and D. Moran. Interleaving syntax and semantics in an efficient bottom-up parser. In *32nd Annual Meeting of the ACL* [5], pages 110–116.

[DNS92] C. Dekkers, M.J. Nederhof, and J.J. Sarbo. Coping with ambiguity in decorated parse forests. In *Coping with Linguistic Ambiguity in Typed Feature Formalisms*, Proceedings of a Workshop held at ECAI 92, pages 11–19, Vienna, Austria, August 1992.

[dV93] J.P.M. de Vreught. *Parallel Parsing*. PhD thesis, Delft University of Technology, 1993.

[Dym92] M. Dymetman. A generalized Greibach normal form for definite clause grammars. In *COLING-92* [11], volume 1, pages 366–372.

[Ear70] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

[Ear75] J. Earley. Ambiguity and precedence in syntax description. *Acta Informatica*, 4:183–192, 1975.

[ED88] A. Eisele and J. Dörre. Unification of disjunctive feature descriptions. In *26th Annual Meeting of the ACL* [2], pages 286–294.

[Eva85] R. Evans. ProGram — a development tool for GPSG grammars. *Linguistics*, 23:213–243, 1985.

[FG92] J. Fortes Gálves. Generating LR(1) parsers of small size. In *Compiler Construction, 4th International Conference*, Lecture Notes in Computer Science, volume 641, pages 16–29, Paderborn, FRG, October 1992. Springer-Verlag.

[FKKN91] E. Farkas, C.H.A. Koster, P. Köves, and M. Naszódi. Towards an affix grammar for the Hungarian language. In *Conference on Intelligent Systems*, pages 223–236, Verszprém, Hungary, September 1991.

[Fos68] J.M. Foster. A syntax improving program. *The Computer Journal*, 11:31–34, 1968.

[Fro92] R.A. Frost. Guarded attribute grammars — top down parsing and left recursive productions. *SIGPLAN Notices*, 27(6):72–75, 1992.

[FT93] C.P. Fotinaki and M.A. Theodoropoulou. Tools for natural language processing — profiling and tracing agfls. Master thesis, University of Nijmegen, Department of Computer Science, 1993.

[Gar87] G. Gardarin. Magic functions: a technique to optimize extended Datalog recursive programs. In *Proc. of the Thirteenth International Conference on Very Large Data Bases*, pages 21–30, Brighton, England, September 1987.

[GHR80] S.L. Graham, M.A. Harrison, and W.L. Ruzzo. An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, 2(3):415–462, July 1980.

[Gri92] R.Z. Gril. Towards computer-aided linguistic engineering. In *COLING-92* [11], volume 2, pages 827–834.

[Hal90] J. Hallebeek. *Een Grammatica voor Automatische Analyse van het Spaans*. PhD thesis, University of Nijmegen, 1990.

[Ham74] M. Hammer. A new grammatical transformation into LL(k) form. In *Conference Record of the Sixth Annual ACM Symposium on Theory of Computing*, pages 266–275, 1974.

[Har78] M.A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, 1978.

[Hei81] S. Heilbrunner. A parsing automata approach to LR theory. *Theoretical Computer Science*, 15:117–157, 1981.

[Hei85] S. Heilbrunner. Truly prefix-correct chain-free LR(1) parsers. *Acta Informatica*, 22:499–536, 1985.

[Hes92] W.H. Hesselink. LR-parsing derived. *Science of Computer Programming*, 19:171–196, 1992.

[HG86] H. Haugeneder and M. Gehrke. A user friendly ATN programming environment (APE). In *Coling '86* [8], pages 399–401.

[HS91] S. Heilbrunner and L. Schmitz. An efficient recognizer for the Boolean closure of context-free languages. *Theoretical Computer Science*, 80:53–75, 1991.

[IF83] K. Inoue and F. Fujiwara. On LLC(k) parsing method of LR(k) grammars. *Journal of Information Processing*, 6(4):206–217, 1983.

[JM92] E.K. Jones and L.M. Miller. Eager GLR parsing. In *First Australian Workshop on Natural Language Processing and Information Retrieval*, Melbourne, Australia, 1992.

[JM94] E.K. Jones and L.M. Miller. L$*$ parsing: A general framework for syntactic analysis of natural language. In *Proceedings Twelfth National Conference on Artificial Intelligence*, Seattle, Washington, 1994.

[Joh91] M. Johnson. The computational complexity of GLR parsing. In Tomita [Tom91], chapter 3, pages 35–42.

[Joh94] M. Johnson. Computing with features as formulae. *Computational Linguistics*, 20(1):1–25, 1994.

[JPSZ92] W. Janssen, M. Poel, K. Sikkel, and J. Zwiers. The primordial soup algorithm: A systematic approach to the specification of parallel parsers. In *COLING-92* [11], volume 1, pages 373–379.

[JR89] R. Johnson and M. Rosner. A rich environment for experimentation with unification grammars. In *Fourth Conference of the European Chapter of the Association for Computational Linguistics, Proceedings of the Conference*, pages 182–189, Manchester, England, April 1989.

[KA88] F.E.J. Kruseman Aretz. On a recursive ascent parser. *Information Processing Letters*, 29(4):201–206, November 1988.

[KA89] F.E.J. Kruseman Aretz. A new approach to Earley's parsing algorithm. *Science of Computer Programming*, 12:105–121, 1989.

[Kap73] R.M. Kaplan. A general syntactic processor. In Rustin [Rus73], pages 193–241.

[Kap87] R.M. Kaplan. Three seductions of computational psycholinguistics. In P. Whitelock, M.M. Wood, H.L. Somers, R. Johnson, and P. Bennett, editors, *Linguistic Theory and Computer Applications*, pages 149–188. Academic Press, 1987.

[Kar86] L. Karttunen. D-PATR: A development environment for unification-based grammars. In *Coling '86* [8], pages 74–80.

[Kay73] M. Kay. The MIND system. In Rustin [Rus73], pages 155–188.

[Kay86] M. Kay. Algorithm schemata and data structures in syntactic processing. In B.J. Grosz, K. Sparck Jones, and B.L. Webber, editors, *Readings in Natural Language Processing*, pages 35–70. Morgan Kaufmann, 1986. Originally, report CSL-80-12, Xerox Corporation, 1980.

[Kay89] M. Kay. Head-driven parsing. In *International Parsing Workshop '89* [13], pages 52–62.

[Kil84] J. Kilbury. Chart parsing and the Earley algorithm. In U. Klenk, editor, *Kontextfreie Syntaxen und verwandte Systeme*, Linguistische Arbeiten, volume 155, pages 76–89, Ventron (Vogesen), October 1984. Max Niemeyer Verlag.

[Kip91] J.R. Kipps. GLR parsing in time $\mathcal{O}(n^3)$. In Tomita [Tom91], chapter 4, pages 43–59.

[KK85] L. Karttunen and M. Kay. Parsing in a free word order language. In D.R. Dowty, L. Karttunen, and A.M. Zwicky, editors, *Natural language parsing: Psychological, computational, and theoretical perspectives*, pages 279–306. Cambridge University Press, 1985.

[KM92] H.V. Kontouli and M.-A. G. Mountzia. Morphological and syntactical description of the Greek language. Master thesis, University of Nijmegen, Department of Computer Science, 1992.

[Knu71] D.E. Knuth. Top-down syntax analysis. *Acta Informatica*, 1:79–110, 1971.

[Kos71] C.H.A. Koster. Affix grammars. In J.E.L. Peck, editor, *ALGOL68 Implementation*, pages 95–109. North Holland Publishing Company, Amsterdam, 1971.

[Kos74] C.H.A. Koster. A technique for parsing ambiguous languages. Interner Bericht Nr. 74 – 27, FB 20 der TU Berlin, October 1974.

[Kos91a] C.H.A. Koster. Affix grammars for natural languages. In *Attribute Grammars, Applications and Systems, International Summer School SAGA*, Lecture Notes in Computer Science, volume 545, pages 358–373, Prague, Czechoslovakia, June 1991. Springer-Verlag.

[Kos91b] C.H.A. Koster. *Systematisch Leren Programmeren, Deel 2: Bottom-Up programmering*. Academic Service, Schoonhoven, Holland, 1991.

[KR88] P.N. Klein and J.H. Reif. Parallel time $O(\log n)$ acceptance of deterministic CFLs on an exclusive-write P-RAM. *SIAM Journal on Computing*, 17(3):463–485, 1988.

[Kun65] S. Kuno. The predictive analyzer and a path elimination technique. *Communications of the ACM*, 8(7):453–462, July 1965.

[KW91] C.H.A. Koster and R. Willems. Towards an affix grammar for Turkish. In *Proc. of the Sixth International Symposium on Computer and Information Sciences*, volume 2, pages 1067–1076, Side, Antalya, Turkey, October–November 1991.

[LAKA92] R. Leermakers, L. Augusteijn, and F.E.J. Kruseman Aretz. A functional LR parser. *Theoretical Computer Science*, 104:313–323, 1992.

[Lan74] B. Lang. Deterministic techniques for efficient non-deterministic parsers. In *Automata, Languages and Programming, 2nd Colloquium*, Lecture Notes in Computer Science, volume 14, pages 255–269, Saarbrücken, 1974. Springer-Verlag.

[Lan88a] B. Lang. Complete evaluation of Horn clauses: An automata theoretic approach. Rapport de Recherche 913, Institut National de Recherche en Informatique et en Automatique, Rocquencourt, France, November 1988.

[Lan88b] B. Lang. Datalog automata. In *Proc. of the Third International Conference on Data and Knowledge Bases: Improving Usability and Responsiveness*, pages 389–401, Jerusalem, June 1988.

[Lan88c] B. Lang. Parsing incomplete sentences. In *COLING '88* [9], volume 1, pages 365–371.

[Lan88d] B. Lang. The systematic construction of Earley parsers: Application to the production of $\mathcal{O}(n^6)$ Earley parsers for tree adjoining grammars. Unpublished paper, December 1988.

[Lan89] B. Lang. A generative view of ill-formed input processing. In *ATR Symposium on Basic Research for Telephone Interpretation*, Kyoto, Japan, December 1989.

[Lan91a] B. Lang. Towards a uniform formal framework for parsing. In M. Tomita, editor, *Current Issues in Parsing Technology*, chapter 11, pages 153–171. Kluwer Academic Publishers, 1991.

[Lan91b] M. Lankhorst. An empirical comparison of generalized LR tables. In R. Heemels, A. Nijholt, and K. Sikkel, editors, *Tomita's Algorithm: Extensions and Applications*, Proc. of the first Twente Workshop on Language Technology, pages 87–93. University of Twente, September 1991. Memoranda Informatica 91-68.

[Lan92] B. Lang. Recognition can be harder than parsing. Unpublished paper, April 1992.

[Lav94] A. Lavie. An integrated heuristic scheme for partial parse evaluation. In *32nd Annual Meeting of the ACL* [5], pages 316–318.

[LCVH93] B. Le Charlier and P. Van Hentenryck. A general fixpoint algorithm for abstract interpretation. Technical Report 93-22, Institute of Computer Science, University of Namur, Belgium, June 1993.

[Lee89] R. Leermakers. How to cover a grammar. In *27th Annual Meeting of the ACL* [3], pages 135–142.

[Lee91] R. Leermakers. Non-deterministic recursive ascent parsing. In *Fifth Conference of the European Chapter of the Association for Computational Linguistics, Proceedings of the Conference*, pages 63–68, Berlin, Germany, April 1991.

[Lee92a] R. Leermakers. A recursive ascent Earley parser. *Information Processing Letters*, 41(2):87–91, February 1992.

[Lee92b] R. Leermakers. Recursive ascent parsing: from Earley to Marcus. *Theoretical Computer Science*, 104:299–312, 1992.

[Lee93] R. Leermakers. *The Functional Treatment of Parsing*. Kluwer Academic Publishers, 1993.

[Lei90] H. Leiss. On Kilbury's modification of Earley's algorithm. *ACM Transactions on Programming Languages and Systems*, 12(4):610–640, October 1990.

[Leo91] J.M.I.M. Leo. A general context-free parsing algorithm running in linear time on every LR(k) grammar without using lookahead. *Theoretical Computer Science*, 82:165–176, 1991.

[Llo84] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.

[LRS76] P.M. Lewis II, D.J. Rosenkrantz, and R.E. Stearns. *Compiler Design Theory*. Addison-Wesley, 1976.

[LT93] A. Lavie and M. Tomita. GLR* – an efficient noise-skipping parsing algorithm for context free grammars. In *IWPT 3* [14], pages 123–134.

[LY91] V.S. Lakshmanan and C.H. Yim. Can filters do magic for deductive databases? In *3rd UK Annual Conference on Logic Programming*, pages 174–189, Edinburgh, April 1991. Springer-Verlag.

[M$^+$83] Y. Matsumoto et al. BUP: A bottom-up parser embedded in Prolog. *New Generation Computing*, 1:145–158, 1983.

[Mah87] U. Mahn. *Attributierte Grammatiken und Attributierungsalgorithmen*, Informatik-Fachberichte, volume 157. Springer-Verlag, 1987.

[Mar92] H. Maruyama. JAUNT: A constraint solver for disjunctive feature structures. In *COLING-92* [11], volume 4, pages 1162–1166.

[Meij86] H. Meijer. *Programmar: A Translator Generator*. PhD thesis, University of Nijmegen, 1986.

[MK93] J.T. Maxwell III and R.M. Kaplan. The interface between phrasal and functional constraints. *Computational Linguistics*, 19(4):571–590, 1993.

[MS87] Y. Matsumoto and R. Sugimura. A parsing system based on logic programming. In *IJCAI-87* [12], volume 2, pages 671–674.

[MW88] D. Maier and D.S. Warren. *Computing with Logic*. The Benjamin/Cummings Publishing Company, 1988.

[Nag92a] K. Nagao. A preferential constraint satisfaction technique for natural language analysis. In *10th European Conference on Artificial Intelligence*, pages 522–527, Vienna, Austria, August 1992.

[Nag92b] M. Nagata. An empirical study on rule granularity and unification interleaving toward an efficient unification-based parsing system. In *COLING-92* [11], volume 1, pages 177–183.

[NB94] M.-J. Nederhof and E. Bertsch. Linear-time suffix recognition for deterministic languages. Technical Report CSI-R9409, University of Nijmegen, Department of Computer Science, August 1994.

[Ned93a] M.J. Nederhof. Generalized left-corner parsing. In *Sixth Conference of the European Chapter of the ACL* [7], pages 305–314.

[Ned93b] M.J. Nederhof. A multidisciplinary approach to a parsing algorithm. In Sikkel and Nijholt [SN93], pages 85–98.

[Ned93c] M.J. Nederhof. A new top-down parsing algorithm for left-recursive DCGs. In *Programming Language Implementation and Logic Programming, 5th International Symposium*, Lecture Notes in Computer Science, volume 714, pages 108–122, Tallinn, Estonia, August 1993. Springer-Verlag.

[Ned94a] M.-J. Nederhof. Efficient generation of random sentences. Technical Report CSI-R9408, University of Nijmegen, Department of Computer Science, August 1994.

[Ned94b] M.J. Nederhof. An optimal tabular parsing algorithm. In *32nd Annual Meeting of the ACL* [5], pages 117–124.

[NF89] R. Nozohoor-Farshi. Handling of ill-designed grammars in Tomita's parsing algorithm. In *International Parsing Workshop '89* [13], pages 182–192.

[NF91] R. Nozohoor-Farshi. GLR parsing for ε-grammars. In Tomita [Tom91], chapter 5, pages 61–75.

[Nij80] A. Nijholt. *Context-Free Grammars: Covers, Normal Forms, and Parsing*, Lecture Notes in Computer Science, volume 93. Springer-Verlag, 1980.

[Nil86] U. Nilsson. AID: An alternative implementation of DCGs. *New Generation Computing*, 4:383–399, 1986.

[NK92] M.J. Nederhof and K. Koster. A customized grammar workbench. In J. Aarts, P. de Haan, and N. Oostdijk, editors, *English Language Corpora: Design, Analysis and Exploitation*, Papers from the thirteenth International Conference on English Language Research on Computerized Corpora, pages 163–179, Nijmegen, 1992. Rodopi.

[NK93] M.J. Nederhof and C.H.A. Koster. Top-down parsing for left-recursive grammars. Technical Report no. 93–10, University of Nijmegen, Department of Computer Science, June 1993.

[NKDvZ92] M.J. Nederhof, C.H.A. Koster, C. Dekkers, and A. van Zwol. The Grammar Workbench: A first step towards lingware engineering. In W. ter Stal, A. Nijholt, and H.J. op den Akker, editors, *Linguistic Engineering: Tools and Products*, Proc. of the second Twente Workshop on Language Technology, pages 103–115. University of Twente, April 1992. Memoranda Informatica 92-29.

[Nor91] P. Norvig. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1):91–98, 1991.

[NS93a] M.J. Nederhof and J.J. Sarbo. Efficient decoration of parse forests. In H. Trost, editor, *Feature Formalisms and Linguistic Ambiguity*, pages 53–78. Ellis Horwood, 1993.

[NS93b] M.J. Nederhof and J.J. Sarbo. Increasing the applicability of LR parsing. In *IWPT 3* [14], pages 187–201.

[NS93c] M.J. Nederhof and J.J. Sarbo. Increasing the applicability of LR parsing. Technical Report no. 93–06, University of Nijmegen, Department of Computer Science, March 1993.

[NS94] M.J. Nederhof and G. Satta. An extended theory of head-driven parsing. In *32nd Annual Meeting of the ACL* [5], pages 210–217.

[OL93] P. Oude Luttighuis. *Parallel algorithms for parsing and attribute evaluation*. PhD thesis, University of Twente, 1993.

[OLS93] P. Oude Luttighuis and K. Sikkel. Generalized LR parsing and attribute evaluation. In *IWPT 3* [14], pages 219–233.

[Oos91] N. Oostdijk. *Corpus Linguistics and the Automatic Analysis of English*. PhD thesis, University of Nijmegen, 1991.

[Pag70] D. Pager. A solution to an open problem by Knuth. *Information and Control*, 17:462–473, 1970.

[Par84] H. Partsch. Structuring transformational developments: a case study based on Earley's recognizer. *Science of Computer Programming*, 4:17–44, 1984.

[Par86] H. Partsch. Transformational program development in a particular problem domain. *Science of Computer Programming*, 7:99–241, 1986.

[Par90] H.A. Partsch. *Specification and Transformation of Programs: A Formal Approach to Software Development*. Springer-Verlag, 1990.

[PB81] P.W. Purdom, Jr. and C.A. Brown. Parsing extended LR(k) grammars. *Acta Informatica*, 15:115–127, 1981.

[PB91] M. Piastra and R. Bolognesi. An efficient context-free parsing algorithm with semantic actions. In *Trends in Artificial Intelligence, 2nd Congress of the Italian Association for Artificial Intelligence, AI*IA*, Lecture Notes in Artificial Intelligence, volume 549, pages 271–280, Palermo, Italy, October 1991. Springer-Verlag.

[Per85] F.C.N. Pereira. A structure-sharing representation for unification-based grammar formalisms. In *23rd Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 137–144, Chicago, Illinois, USA, July 1985.

[Per91] M. Perlin. LR recursive transition networks for Earley and Tomita parsing. In *29th Annual Meeting of the ACL* [4], pages 98–105.

[Pra75] V.R. Pratt. LINGOL - A progress report. In *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, pages 422–428, Tbilisi, Georgia, USSR, September 1975.

[Pur74] P. Purdom. The size of LALR (1) parsers. *BIT*, 14:326–337, 1974.

[PvE93] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[PW80] F.C.N. Pereira and D.H.D. Warren. Definite clause grammars for language analysis—a survey of the formalism and a comparison with the augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.

[PW83] F.C.N. Pereira and D.H.D. Warren. Parsing as deduction. In *21st Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 137–144, Cambridge, Massachusetts, July 1983.

[Ram88] R. Ramakrishnan. Magic Templates: a spellbinding approach to logic programs. In *Logic Programming, Proc. of the Fifth International Conference and Symposium*, pages 140–159, Seattle, 1988.

[Rek92] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

[RH88] P. Rangel Henriques. A semantic evaluator generating system in Prolog. In *PLILP '88* [15], Lecture Notes in Computer Science, volume 348, pages 201–218.

[RL70] D.J. Rosenkrantz and P.M. Lewis II. Deterministic left corner parsing. In *IEEE Conference Record of the 11th Annual Symposium on Switching and Automata Theory*, pages 139–152, 1970.

[RLK86] J. Rohmer, R. Lescoeur, and J.M. Kerisit. The Alexander Method — a technique for the processing of recursive axioms in deductive databases. *New Generation Computing*, 4:273–285, 1986.

[Rob88] G.H. Roberts. Recursive ascent: An LR analog to recursive descent. *SIGPLAN Notices*, 23(8):23–29, August 1988.

[Rob89] G.H. Roberts. Another note on recursive ascent. *Information Processing Letters*, 32(5):263–266, September 1989.

[Rus73] R. Rustin, editor. *Natural Language Processing.* Algorithmic Press, New York, 1973.

[Ryt82] W. Rytter. Time complexity of unambiguous path systems. *Information Processing Letters*, 15(3):102–104, October 1982.

[Ryt85] W. Rytter. Fast recognition of pushdown automaton and context-free languages. *Information and Control*, 67:12–22, 1985.

[Sai90] H. Saito. Bi-directional LR parsing from an anchor word for speech recognition. In *COLING-90* [10], volume 3, pages 237–242.

[SB90] J.J. Schoorl and S. Belder. Computational linguistics at Delft: A status report. Report WTM/TT 90–09, Delft University of Technology, Applied Linguistics Unit, 1990.

[Sch88] K. Schifferer. TAGDevEnv: Eine Werkbank für TAGs. In *Computerlinguistik und ihre theoretischen Grundlagen, Symposium*, Informatik-Fachberichte, volume 195, pages 152–171, Saarbrücken, March 1988. Springer-Verlag.

[Sch91] Y. Schabes. Polynomial time and space shift-reduce parsing of arbitrary context-free grammars. In *29th Annual Meeting of the ACL* [4], pages 106–113.

[SDR87] S. Steel and A. De Roeck. Bidirectional chart parsing. In J. Hallam and C. Mellish, editors, *Advances in Artificial Intelligence*, Proc. of the 1987 AISB Conference, pages 223–235, University of Edinburgh, 1987. John Wiley & Sons.

[Sei94] D. Seipel. An efficient computation of the extended generalized closed world assumption by support-for-negation sets. In *Logic Programming and Automated Reasoning, 5th International Conference*, Lecture Notes in Artificial Intelligence, volume 822, pages 245–259, Kiev, Ukraine, 1994. Springer-Verlag.

[Sek89] H. Seki. On the power of Alexander Templates. In *Proc. of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 150–159, 1989.

[Sha87] E. Shapiro, editor. *Concurrent Prolog.* Collected Papers. MIT Press, 1987. Two volumes.

[Sha91] P. Shann. Experiments with GLR and chart parsing. In Tomita [Tom91], chapter 2, pages 17–34.

[She76] B.A. Sheil. Observations on context-free parsing. *Statistical Methods in Linguistics*, 1976, pages 71–109.

[SI88] H. Seki and H. Itoh. A query evaluation method for stratified programs under the extended CWA. In *Logic Programming, Proc. of the Fifth International Conference and Symposium*, pages 195–211, Seattle, 1988.

[Sik90] K. Sikkel. Cross-fertilization of Earley and Tomita. Memoranda Informatica 90-69, University of Twente, November 1990.

[Sik93] K. Sikkel. *Parsing Schemata.* PhD thesis, University of Twente, 1993.

[SL92] K. Sikkel and M. Lankhorst. A parallel bottom-up Tomita parser. In *KONVENS 92* [1], pages 238–247.

[Slo81] J. Slocum. A practical comparison of parsing strategies. In *19th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 1–6, Stanford, California, June–July 1981.

[SN93] K. Sikkel and A. Nijholt, editors. *Natural Language Parsing: Methods and Formalisms*, Proc. of the sixth Twente Workshop on Language Technology. University of Twente, 1993.

[SodA92] K. Sikkel and R. op den Akker. Head-corner chart parsing. In *Computing Science in the Netherlands*, pages 279–290, Utrecht, November 1992.

[SodA93] K. Sikkel and R. op den Akker. Predictive head-corner chart parsing. In *IWPT 3* [14], pages 267–276.

[SS89] G. Satta and O. Stock. Head-driven bidirectional parsing: A tabular method. In *International Parsing Workshop '89* [13], pages 43–51.

[SS91] G. Satta and O. Stock. A tabular method for island-driven context-free grammar parsing. In *Proceedings Ninth National Conference on Artificial Intelligence*, volume 1, pages 143–148, July 1991.

[SS94] G. Satta and O. Stock. Bidirectional context-free grammar parsing for natural language processing. *Artificial Intelligence*, 1994. To appear.

[SSS90] S. Sippu and E. Soisalon-Soininen. *Parsing Theory, Vol. II: LR(k) and LL(k) Parsing*, EATCS Monographs on Theoretical Computer Science, volume 20. Springer-Verlag, 1990.

[SST88] E. Soisalon-Soininen and J. Tarhio. Looping LR parsers. *Information Processing Letters*, 26(5):251–253, January 1988.

[SSU79] E. Soisalon-Soininen and E. Ukkonen. A method for transforming grammars into LL(k) form. *Acta Informatica*, 12:339–369, 1979.

[Szp87] S. Szpakowicz. Logic grammars. *BYTE*, August 1987, pages 185–195.

[TDL91] H.S. Thompson, M. Dixon, and J. Lamping. Compose-reduce parsing. In *29th Annual Meeting of the ACL* [4], pages 87–97.

[Tei84] W. Teitelman. A display-oriented programmer's assistant. In Barstow et al. [BSS84], chapter 13, pages 240–287.

[Tho94] M. Thorup. Controlled grammatic ambiguity. *ACM Transactions on Programming Languages and Systems*, 16(3):1024–1050, May 1994. In press.

[TN91a] H. Tanaka and H. Numazaki. Parallel GLR parsing based on logic programming. In Tomita [Tom91], chapter 6, pages 77–91.

[TN91b] M. Tomita and S.K. Ng. The generalized LR parsing algorithm. In Tomita [Tom91], chapter 1, pages 1–16.

[Tom86] M. Tomita. *Efficient Parsing for Natural Language.* Kluwer Academic Publishers, 1986.

[Tom87] M. Tomita. An efficient augmented-context-free parsing algorithm. *Computational Linguistics*, 13:31–46, 1987.

[Tom88] M. Tomita. Graph-structured stack and natural language parsing. In *26th Annual Meeting of the ACL* [2], pages 249–257.

[Tom91] M. Tomita, editor. *Generalized LR Parsing.* Kluwer Academic Publishers, 1991.

[TR84] H. Thompson and G. Ritchie. Implementing natural language parsers. In T. O'Shea and M. Eisenstadt, editors, *Artificial Intelligence: Tools, Techniques, and Applications*, chapter 9, pages 245–300. Harper & Row, New York, 1984.

[TS86] H. Tamaki and T. Sato. OLD resolution with tabulation. In *Third International Conference on Logic Programming*, Lecture Notes in Computer Science, volume 225, pages 84–98, London, United Kingdom, 1986. Springer-Verlag.

[TSM79] H. Tanaka, T. Sato, and F. Motoyoshi. Predictive control parser: Extended LINGOL. In *Proc. of the Sixth International Joint Conference on Artificial Intelligence*, volume 2, pages 868–870, Tokyo, August 1979.

[UOKT84] K. Uehara, R. Ochitani, O. Kakusho, and J. Toyoda. A bottom-up parser based on predicate logic: A survey of the formalism and its implementation technique. In *1984 International Symposium on Logic Programming*, pages 220–227, Atlantic City, New Jersey, February 1984.

[Var83] G.B. Varile. Charts: a data structure for parsing. In M. King, editor, *Parsing Natural Language*, chapter 5, pages 73–87. Academic Press, 1983.

[vB84] J. van Bakel. *Automatic Semantic Interpretation — A Computer Model of Understanding Natural Language*. Foris Publications, 1984.

[VB85] B. Vauquois and C. Boitet. Automated translation at Grenoble University. *Computational Linguistics*, 11(1):28–36, 1985.

[VdlC93] E. Villemonte de la Clergerie. *Automates à Piles et Programmation Dynamique — DyALog: Une application à la Programmation en Logique*. PhD thesis, University Paris VII, 1993.

[Vér92] J. Véronis. Disjunctive feature structures as hypergraphs. In *COLING-92* [11], volume 2, pages 498–504.

[vH91] H. van Halteren. Efficient storage of ambiguous structures in textual databases. *Literary and Linguistic Computing*, 6(4):233–242, 1991.

[vHvdH90] H. van Halteren and T. van den Heuvel. *Linguistic Exploitation of Syntactic Databases*. Rodopi, 1990.

[Vie89] L. Vieille. Recursive query processing: the power of logic. *Theoretical Computer Science*, 69:1–53, 1989.

[Voi86] F. Voisin. CIGALE: A tool for interactive grammar construction and expression parsing. *Science of Computer Programming*, 7:61–86, 1986.

[Voi88] F. Voisin. A bottom-up adaptation of Earley's parsing algorithm. In *PLILP '88* [15], Lecture Notes in Computer Science, volume 348, pages 146–160.

[Vos93] T. Vosse. Robust GLR parsing for grammar-based spelling correction. In Sikkel and Nijholt [SN93], pages 169–181.

[VR90] F. Voisin and J.-C. Raoult. A new, bottom-up, general parsing algorithm. *BIGRE*, 70:221–235, September 1990.

[VSW93] K. Vijay-Shanker and D.J. Weir. The use of shared forests in tree adjoining grammar parsing. In *Sixth Conference of the European Chapter of the ACL* [7], pages 384–393.

[Wah84] W. Wahlster. Zur Rolle der Linguistik bei der Entwicklung natürlichsprachlicher KI-Systeme. In *8th German Workshop on Artificial Intelligence*, Informatik-Fachberichte, volume 103, pages 267–269, Wingst/Stade, October 1984. Springer-Verlag.

[War92] D.S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, March 1992.

[Whi87] J.S. White. The research environment in the METAL project. In S. Nirenburg, editor, *Machine translation: theoretical and methodological issues*, pages 225–246. Cambridge University Press, 1987.

[Win83] T. Winograd. *Language as a Cognitive Process, Volume 1: Syntax*. Addison-Wesley, 1983.

[Win84] T. Winograd. Breaking the complexity barrier (again). In Barstow et al. [BSS84], chapter 1, pages 3–18.

[Wir87] M. Wirén. A comparison of rule-invocation strategies in context-free chart parsing. In *Third Conference of the European Chapter of the ACL* [6], pages 226–233.

[Wir93] M. Wirén. Bounded incremental parsing. In Sikkel and Nijholt [SN93], pages 145–156.

[WR93] M. Wirén and R. Rönnquist. Fully incremental parsing. In *IWPT 3* [14]. hand-out, not included in the original proceedings.

[Zad84] F.K. Zadeck. Incremental data flow analysis in a structured program editor. *SIGPLAN Notices*, 19(6):132–143, 1984.

[ZLC94] D. Zampuniéris and B. Le Charlier. A yet more efficient algorithm to compute the synchronized product. Research Report 94/5, Institute of Computer Science - FUNDP Namur, Belgium, 1994.

[1] *1. Konferenz "Verarbeitung Natürlicher Sprache" (KONVENS 92)*, Nürnberg, October 1992. Springer-Verlag.

[2] *26th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, Buffalo, New York, June 1988.

[3] *27th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, Vancouver, British Columbia, Canada, June 1989.

[4] *29th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, Berkeley, California, USA, June 1991.

[5] *32nd Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, Las Cruces, New Mexico, USA, June 1994.

[6] *Third Conference of the European Chapter of the Association for Computational Linguistics, Proceedings of the Conference*, Copenhagen, Denmark, April 1987.

[7] *Sixth Conference of the European Chapter of the Association for Computational Linguistics, Proceedings of the Conference*, Utrecht, The Netherlands, April 1993.

[8] *11th International Conference on Computational Linguistics (Coling '86)*, Bonn, August 1986. University of Bonn.

[9] *Proc. of the 12th International Conference on Computational Linguistics (COLING '88)*, Budapest, August 1988.

[10] *Papers presented to the 13th International Conference on Computational Linguistics (COLING-90)*, 1990.

[11] *Proc. of the fifteenth International Conference on Computational Linguistics (COLING-92)*, Nantes, August 1992.

[12] *Proc. of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, Milan, August 1987.

[13] *International Workshop on Parsing Technologies*, Pittsburgh, 1989.

[14] *Third International Workshop on Parsing Technologies (IWPT 3)*, Tilburg (The Netherlands) and Durbuy (Belgium), August 1993.

[15] *Programming Languages Implementation and Logic Programming, International Workshop (PLILP '88)*, Lecture Notes in Computer Science, volume 348, Orléans, France, May 1988. Springer-Verlag.

Samenvatting

Betrouwbare ontwikkeling van computerprogramma's kan geschieden middels het principe van *transformationeel programmeren*. Het uitgangspunt hierbij is een formele specificatie van een taak die een beoogd computerprogramma moet verrichten. Deze specificatie kan ondergespecificeerd zijn (d.w.z. het hoeft niet eenduidig de te verrichten taak te beschrijven) en is in het algemeen niet direct uitvoerbaar (d.w.z. de gebruikte specificatietaal wordt niet door een computer begrepen). We kunnen dan de specificatie veranderen m.b.v. kleine stapjes (*transformaties*), waarvan al vast staat dat ze de betekenis van een specificatie niet anders kunnen maken dan bedoeld was. Geleidelijk wordt zo de specificatie omgezet in iets dat wel uitvoerbaar is, namelijk het beoogde computerprogramma. We mogen nu aannemen dat dit computerprogramma correct is t.a.v. de oorspronkelijke specificatie, omdat we hadden aangenomen dat de gebruikte transformaties correct waren.

Een praktisch probleem van transformationeel programmeren is echter dat het aantal transformaties waarover we moeten beschikken om een niet-triviaal programma te kunnen afleiden zeer groot kan zijn, en bovendien is het voor de ontwikkelaar van het programma niet altijd makkelijk te bepalen welke transformaties gebruikt moeten worden om een goed werkend programma te verkrijgen, d.w.z. een programma dat niet alleen de beoogde taak verricht maar ook efficiënt omspringt met rekentijd en geheugencapaciteit.

In dit proefschrift zullen we ons beperken tot een paar zeer simpele problemen, namelijk die van de herkenning en ontleding van zinnen. Het uitgangspunt hierbij zijn *grammatica's*. Grammatica's zijn beschrijvingen van talen. Deze talen kunnen computertalen zijn, maar ook natuurlijke talen, zoals het Nederlands of het Frans. Het probleem van herkenning wordt gegeven door: "Stel ik heb een taal, beschreven door een grammatica, en ik heb een zin. Bepaal nu of de zin in de taal zit." Het ontledingsprobleem is iets moeilijker: niet alleen moet de vraag worden beantwoord of de zin in de taal zit maar ook hoe, d.w.z. er wordt een antwoord verwacht dat aangeeft hoe de structuur van de zin eruitziet.

Het construeren van automatische herkenners en ontleders is een probleem waarvoor we transformationeel programmeren kunnen gebruiken. De specificaties zijn hier de grammatica's en de beoogde programma's zijn de herkenners en ontleders. Een klein aantal transformaties die hiervoor bruikbaar zijn wordt in dit proefschrift bestudeerd. Wegens mijn persoonlijke interesse in natuurlijke talen zullen vooral de aspecten van de transformaties die relevant zijn voor de verwerking hiervan worden besproken.

In hoofdstuk 1 wordt de literatuur behandeld van het *tabulair ontleden*. Een berekening in het algemeen is tabulair als berekeningen van tussenresultaten worden opgeslagen in een tabel, zodat eenzelfde tussenresultaat nooit meer dan eens hoeft te worden berekend. De tabulaire ontleedmethoden maken gebruik van tabellen om de ontledingen van elk stukje van de zin maar één maal te hoeven uitrekenen. De constructie van tabulaire ontleders kan gezien worden in het kader van transformationeel programmeren daar deze constructie vaak automatisch kan geschieden gegeven een niet-tabulaire ontleder. Met name zal een constructie van een tabulaire ontleder uit een ontleder geformuleerd als een stackautomaat een grote rol spelen.

In hoofdstuk 2 bespreken we een specifiek tabulair ontleedalgoritme. Dit algoritme is gebaseerd op left-corner ontleding, dat traditioneel een algoritme is dat werkt op een stack. We laten zien dat dit algoritme grote voordelen heeft t.o.v. een zeer beroemd tabulair

algoritme, namelijk Tomita's algoritme, dat gebaseerd is op LR ontleding.

Hoofdstuk 3 laat een hele familie van ontleedalgoritmen de revue passeren die werken op een stack. We laten dan zien dat deze familie af te beelden is op een familie van tabulaire algoritmen. Een interessant resultaat is dat één van deze tabulaire algoritmen optimaal is, zij het volgens zekere criteria en met enkele kunstgrepen.

Het al eerder genoemde LR algoritme is het onderwerp van hoofdstuk 4. LR ontleding zoals dat traditioneel gebruikt wordt kan sommige grammatica's niet aan: met name sommige eigenschappen van grammatica's voor natuurlijke talen maken dat een simpele (non-deterministische) LR ontleder niet termineert. We geven een aantal manieren om dit te verhelpen. Een van deze manieren is nieuw, en heeft grote voordelen boven de andere manieren, in termen van de grootte van de ontleder. Een grote rol spelen hier transformaties van grammatica's die uit een grammatica de eigenschappen kunnen verwijderen die terminatie van LR ontleding verhinderen.

Ook in hoofdstuk 5 speelt terminatie een belangrijke rol. We kijken hier naar top-down ontleding, die normaal geen links-recursieve grammatica's aankan. Door nu een kleine maar niet-vanzelfsprekende wijziging aan te brengen in het top-down algoritme verkrijgen we *cancellation parsing* (of *wegstreep ontleding*) dat wel in staat is om links-recursieve grammatica's te verwerken. Een opvallend aspect van dit hoofdstuk is dat alle ontleedalgoritmen worden beschreven m.b.v. een transformatie naar het formalisme van de *definite clause grammars*.

Tot hoofdstuk 6 beschouwen we voornamelijk *context-vrije* grammatica's, die zeer eenvoudig zijn en daarom met niet al te ingewikkelde ontleedalgoritmen te behandelen zijn. Bij interessante grammatica's echter zijn de regels vaak geparametriseerd, d.w.z. ze zijn uitgebreid met logische variabelen die samenhang aangeven tussen verschillende onderdelen van de regel. Een elegante klasse van grammatica's met geparametriseerde regels is het formalisme AGFL. Hierbij kunnen de logische variabelen maar een eindig aantal waarden aannemen. In hoofdstuk 6 laten we nu zien hoe ontleding kan worden gedaan door eerst een tabulair algoritme toe te passen dat de parameters negeert, en dan een ander algoritme toe te passen dat in de ontstane structuur (een zogenaamd *ontleedwoud*) de parameterwaarden berekent.

Doelmatige ontwikkeling van computerprogramma's moet worden ondersteund door automatische hulpmiddelen, die de programmaontwikkelaars helpen bij het controleren van verscheidene eigenschappen van de software en bij het aanbrengen van wijzigingen. In hoofdstuk 7 laten we zien dat ook de ontwikkeling van grammatica's doelmatiger kan geschieden indien ondersteund door geautomatiseerde hulpmiddelen. We bespreken zo'n hulpmiddel, de *Grammar Workbench* voor het AGFL formalisme, en vergelijken dat met andere hulpmiddelen voor de ontwikkeling van grammatica's.

Curriculum Vitae

23 oktober 1966
Geboren te Leiden.

31 mei 1985
VWO-diploma, *Newmancollege* te Breda.

27 april 1990
Doctoraaldiploma (cum laude), Informatica, Katholieke Universiteit Nijmegen.

1 mei 1990 — 1 mei 1994
Onderzoeker in opleiding in het project "Transformationeel Programmeren" (STOP – Specification and Transformation of Programs), projectnr. 00-62-518; Informatica, Katholieke Universiteit Nijmegen.