VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

Flexible Table-Driven Parsing for Natural Language Processing

Linton Miller

Technical Report CS-TR-95/14
July 1995

Abstract

Ambiguity is a major difficulty for natural language processing (NLP) systems. The longer that ambiguities in a sentence remain unresolved, the more work an NLP system may perform in considering alternative interpretations of the sentence. Thus, for efficiency, an NLP system should resolve ambiguities as early as possible in processing.

This report describes L$*$ parsing—an algorithm for table-driven parsing, designed to permit efficient processing of natural language by facilitating the early resolution of ambiguity. The algorithm is a generalisation of GLR parsing that allows grammar rules to be used whenever they may provide useful syntactic information to an NLP system.

L$*$ parsing defines a general framework for specifying a variety of parser control strategies. Different control strategies can be expressed by specifying exactly when grammar rules are to be used. This report presents one possible control strategy, designed to provide syntactic information that enables useful semantic and pragmatic processing, and describes a method of compiling this strategy into a parse table.

Publishing Information

This report was also submitted to Victoria University of Wellington in partial fulfillment of the requirements of a Masters of Science in Computer Science.

Author Information

Linton Miller is a former Masters student in the Department of Computer Science at Victoria University of Wellington.

Acknowledgements

There are so many, many people to thank.

Firstly, special thanks to my supervisor, Eric Jones, for his guidance and energy throughout the project.

My deepest thanks go to my officemate, and fellow sufferer, Aaron Roydhouse, for his good nature, company, and friendship.

Also, thanks go to the other members of the AI group, in particular Peter Andreae, Paul Hosking, and Michael Norrish, for many wide-ranging and varied discussions—they may not have helped me finish, but at least they kept me sane.

A big thank you to the Department programming staff Mark Davies, Duncan McEwan, Bernd Gill and Julian Anderson for a great environment in which to work.

Thanks to the department secretaries Christine Polglase and Valerie McGillivray, who dealt with all those little administrative hassles, and gave me great company.

A heartfelt thank you to my friends and family who have supported me through the whole process, especially my father who waded through endless drafts of unintelligible jargon, and still didn't manage to teach me how to use the word "only".

And finally, a very special thank you to a very special person, without who the whole thing might never have been completed. For her never ending patience, support, and love, my deepest thanks go to Judi.

Contents

List of Figures

List of Tables

List of Grammars

Chapter 1

Introduction

Ambiguity is a major difficulty for natural language processing systems. Ambiguity can be found in all aspects of natural language, and can arise from many different sources. Even unambiguous sentences are constructed from ambiguous parts. Thus, a natural language processing (NLP) system must resolve ambiguity when processing a sentence.

To process natural language *efficiently*, ambiguity must be resolved as early as possible in processing. Ambiguities cause inefficiency because an NLP system must consider alternative interpretations of a sentence. The longer an ambiguity remains unresolved, the more work a system may perform in considering alternative interpretations.

Resolving ambiguity requires wide and varied types of knowledge, any of which can be called on to help resolve a particular ambiguity. Storing this knowledge in a number of modules is one possible organisation of an NLP system. A module consists of knowledge and algorithms relating to one aspect of natural language. A module take inputs from other parts of the system, applies the algorithms of the module, possibly building internal structures as part of the process, and produces outputs that can then be used by other parts of the system. Modularity is desirable in an NLP system because it allows the system to be flexible and expandable. A sentence can be analysed in different ways by different modules, according to the different knowledge and algorithms contained within.

Interaction between modules is vital because different knowledge sources are required for resolving ambiguity in different situations. The points when interaction occurs affect the efficiency of the NLP system as a whole, because they affect when ambiguities can be resolved. If a module performs all its processing before communicating with other parts of the system, not only will it consider more possibilities locally, but it will also present more possibilities for other parts of the system to process as well.

One type of knowledge used in natural language processing is syntactic knowledge. Syntax describes how words group into phrases, which in turn form larger phrases, and so on, forming the sentences of a language. Syntactic knowledge is usually encoded as a *grammar*—a set of rules describing the patterns used in the language. Most NLP systems contain a module for analysing a sentence according to a particular grammar. This module will be referred to as the *syntactic module*. The process of syntactic analysis is known as *parsing*, and the program that performs a syntactic analysis is called a *parser*. Input to the the parser is a sentence expressed as a string of words. Output from the parser is an analysis of the ways rules of the grammar can be applied to construct the sentence.

This report describes a framework for defining efficient parsers for an NLP system. These parsers can be tailored to improve the interaction of a syntactic module with other parts of an NLP system, allowing efficient resolution of ambiguity.

The rest of this chapter examines these issues in further detail. Section 1.1 examines the problem of ambiguity in more detail. Section 1.2 presents some evidence for interaction of modules in an NLP system. Section 1.3 examines the structure of the syntactic module and describes past work. Section 1.4 outlines the new method proposed by this report. Section 1.5 defines some terms and notation. Section 1.6 provides an overview of the rest of this report.

1.1 Ambiguity of natural language

Ambiguity in natural language comes in many different forms. A sentence may be *globally ambiguous*, where the entire sentence has more than one possible interpretation. A sentence may also contain *local ambiguity*, where a part of the sentence is ambiguous in isolation, but is not ambiguous as part of the complete sentence. Local ambiguities are a problem in NLP because they may cause a system to waste time investigating them.

There are many forms of global and local ambiguity. For example, the ambiguity may be structural, lexical, or referential. Structural ambiguities are exemplified in sentences 1–3.

(1) I saw a man with the telescope.

(2) I like fluffy cats and dogs.

(3) The oil filter pump was black.

Sentence 1 can mean that either I did the seeing with a telescope, or the man that I saw had a telescope. Similarly, sentence 2 is ambiguous because the word “fluffy” can refer to either the word “cats”, or the phrase “cats and dogs”. In sentence 3, the compound noun phrase “oil filter pump” can be analysed as either [[oil filter] pump], or [oil [filter pump]].

Lexical ambiguities arise from words having more than one possible interpretation, as in sentences 4–5b.

(4) Judi went to the bank.

(5a) The box is in the pen.

(5b) The pen is in the box.

Sentence 4 has two possible interpretations, depending on whether the word “bank” is taken to mean “a financial institution” or “the side of a river”. Similarly, sentences 5a and 5b are ambiguous because of the word “pen”, which can be interpreted as either “fenced area” or “writing instrument”.

Referential ambiguity is also common, as in sentences 6a–7.

(6a) The city council refused the demonstrators a permit because they feared violence.

(6b) The city council refused the demonstrators a permit because they advocated violence.

(7) Five students ate four slices of pizza.

In sentences 6a and 6b, “they” could conceivably refer either to the city council or the demonstrators. In sentence 7, it is ambiguous whether five students ate four slices of pizza each, or whether five students ate a total of four slices of pizza between them.

Ambiguities multiply with longer, more complex sentences. For example, sentence 8 has over a hundred syntactic parses (Jacobs, Krupka, and Rau, 1991).

(8) A form of asbestos once used to make Kent cigarette filters has caused a high percentage of cancer deaths among a group of workers exposed to it more than 30 years ago, researchers reported. (Wall St. Journal)

To resolve ambiguities requires many different forms of knowledge. To resolve the ambiguities in sentences 1–4 requires knowledge about the context of discourse. For example, if the fact that I was carrying a telescope has been previously established, then the likely meaning of sentence 1 is that through a telescope, I saw a man. The ambiguity in sentences 5a and 5b can be resolved by reasoning about the function of pens and boxes in the world, their size, and possible spatial relationships. In sentence 5a, the meaning of the word "pen" can be interpreted as "fenced area" because it is far more likely for a fenced area to contain boxes. In sentence 5b, the meaning of the word "pen" can be interpreted as "writing instrument" because only a writing instrument can be enclosed in a box. The ambiguity of sentences 6a and 6b can be resolved using the knowledge that city councils are generally fearful of demonstrators being violent, and not vice versa.

Local ambiguities are exemplified by sentences 9b–9a.

(9a) Flying planes are dangerous.

(9b) Flying planes is dangerous.

Having only read "Flying planes", there is a local ambiguity in whether this phrase is a noun phrase meaning "planes which are flying" (sentence 9a) or a verb phrase meaning "the act of flying planes" (sentence 9b).

Some sentences exhibit an extreme form of local ambiguity that causes people reading the sentence to assume one parse, and to backtrack when later information shows that this assumption was wrong. Sentences 10a–10b are examples of these so-called *garden path* sentences.

(10a) The officers taught at the academy were very demanding.

(10b) The courses taught at the academy were very demanding.

The garden path effect of sentence 10a results because the verb "taught" is ambiguous: it could be interpreted as a simple past tense verb, or as a past-participle. Under the simple past tense interpretation, "taught" is the main verb of the sentence, and the reader infers "The officers taught somebody at the academy". Under the past-participle interpretation, "taught" is the verb of a reduced relative clause, and the reader infers "The officers who were taught by somebody at the academy". Having processed only the words "The officers taught", the reader has insufficient information to choose between these two interpretations. In the absence of additional contextual clues, people tend to choose the first interpretation, even though it is incorrect, and change their decision only when they reach the real main verb "were" later in the sentence (Frazier, 1987). In contrast, if the subject noun "officers" is changed to "courses" as in sentence 10b, then the garden path effect is removed.

1.2 Interaction of syntax and semantics in an NLP system

There is psycholinguistic evidence for the interaction of syntax and semantics (Crain and Steedman, 1985; Stowe, 1991; Taraban and McClelland, 1988; Tyler and Marslen-Wilson, 1977). Crain and Steedman (1985) performed a number of experiments with humans to show that context could affect the interpretation of garden path sentences. In one experiment, subjects were presented with sentence pairs such as 11a and 11b.

(11a) The teachers taught by the Berlitz method passed the test.

(11b) The children taught by the Berlitz method passed the test.

Subjects judged sentence 11b as grammatical significantly more often than sentence 11a. Crain and Steedman claimed that the semantic difference between the two sentences (namely that teachers are more likely to teach, and children are more likely to be taught) accounts for these judgments.

Stowe (1991) performed similar experiments by measuring the word-by-word reading times of sentences such as 10a and 10b. These experiments established that semantic information such as the animate nature of the subject influenced the results of syntactic analysis.

The practical benefit of interaction between syntax and semantics is demonstrated empirically by NLP systems such as SCISOR (Rau and Jacobs, 1988), PUNDIT (Lang and Hirschman, 1988), and KERNEL (Palmer, Passonneau, Weir, and Finin, 1993). For example, Lang and Hirschman describe the SPQR module of the PUNDIT text-processing system. The purpose of SPQR is to use domain-specific knowledge to improve the accuracy and efficiency of the parser by ruling out syntactically-correct but domain-inconsistent parses. Their experimental results show that parsing with SPQR reduces the average number of parses found by 30% and reduces the average parse time by 35%.

1.3 Past work on syntactic analysis

The syntactic module of an NLP system consists of two elements: a grammar and a parsing algorithm. The grammar describes the sentences of a language. The parsing algorithm describes a computational mechanism for analysing a sentence according to a grammar.

1.3.1 Grammars of natural language

Much research in linguistics is concerned with encoding the syntax of natural language as a grammar. Many types of grammar have been suggested for this purpose. A popular class of grammars are phrase structure grammars, which describe a language by a set of rules. These rules describe how larger phrases are formed from sub-phrases. Examples of phrase structure grammars include context-free grammars (CFG), linguistic string grammars (Sager, 1981), generalised phrase structure grammars (GPSG) (Gazdar, Klein, Pullum, and Sag, 1985), and head-driven phrase structure grammars (Pollard and Sag, 1987). Perrault (1984) provides a good survey of a number of the main types of grammar.

Context-free grammars have been widely used as a basis for many computational systems, because, even though they cannot model all features of natural language (Shieber, 1987), they can model a useful subset of language. Context-free grammars have the advantages that they are simple, they provide an analysis of parse structure in an explicit manner, and there is a large amount of knowledge and experience in dealing with them. A common method to improve CFGs is to augment them with constraints, such as functional unification grammar (FUG) (Kay, 1982), PATR (Shieber, 1992), and DCG (Pereira and Warren, 1980).

1.3.2 Parsing algorithms

There have been many different parsing algorithms proposed for parsing with different grammars. A particularly popular method for parsing phrase structure grammars has been chart parsing (Kay, 1986), one of the earliest versions of which was created by Earley (1970). Chart parsing allows a number of different control strategies to be implemented, and has a worst case performance of $O(n^3)$ where n is the length of the input sentence. The basic data structure of a chart parser is the *active chart*, which represents all the partial parses it has created as *edges* in the chart. Control strategies of a chart parser are dictated by the order that new edges are added to the chart, and the order in which existing edges are processed. An agenda provides general mechanism for controlling the processing of edges (Kaplan, 1973). The fact that the chart stores all past derivations the parser has constructed allows the parser to avoid unnecessarily repeating work. Introductions to chart parsing can be found in Winograd (1983) and Gazdar and Mellish (1989).

A problem of chart parsers is the overhead of dynamically constructing the chart while parsing. A promising line of research in an attempt to resolve this problem is table-driven parsing, which seeks to reduce the overhead in parsing by pre-compiling a parse table (Lang, 1974; Schabes, 1991; Leermakers, 1989; Nederhof, 1993).

One particular method that has been designed for use in natural language processing is the generalised LR (GLR) parser (Tomita, 1985, 1986, 1987a, 1988; Tomita and Ng, 1991). The GLR parser has been shown to be efficient in comparison with other parsing algorithms (Shann, 1991), and is used as part of the machine translation project at CMU (Nirenburg, Carbonell, Tomita, and Gooman, 1992; Tomita, 1987b). However, the GLR parser suffers because of its strict bottom-up control strategy that does not allow effective interaction with other modules of an NLP system. The parser only reduces when all constituents of a grammar rule have been seen in the input.

1.4 The L$*$ parsing algorithm

This report describes the new L$*$ parsing algorithm for context-free grammars, which is designed to improve the efficiency of an NLP system by facilitating the early resolution of ambiguity. It does this by allowing grammar rules to be used wherever so doing is likely to allow useful semantic or pragmatic processing.

The L$*$ parser is a table-driven parser, for reasons of efficiency. It is implemented as an extension to the GLR algorithm, by introducing *eager reductions*. An eager reduction is similar to a bottom-up reduction except it is carried out before all members of the right hand side of a rule have been parsed. The eager reduction creates an incomplete derivation, because the derivation may be missing some of its children. These children may then be derived from later input and *combined* into the incomplete derivation. A basic familiarity with LR parsing is assumed in the description of the L$*$ parser. An extensive description of LR parsing can be found in (Aho and Ullman, 1977).

L$*$ parsing defines a general framework for creating parsers with different control strategies. Control strategies are expressed by specifying the circumstances in which eager reductions are to be performed, in a method similar to the announce points of Abney and Johnson (1991).

The L$*$ algorithm has been implemented in Lisp (Steele Jr., 1990), and tested on a wide range of grammars specifically designed to exercise the features of the algorithm. Diagrams of the parser stack presented throughout this report have been generated automatically from the implemented L$*$ parser. The research described in this report has also been presented in number of other places (Miller and Jones, 1992; Jones and Miller, 1992, 1993, 1994a, 1994b; Miller, 1994).

1.5 Definitions and Notation

This section introduces definitions and notation used throughout the rest of this report.

A *string* is a sequence of symbols. For example, if *a, b, c* and *d* are symbols, then $abcd$ and bdc are strings. The *empty string* is written ϵ. The *length* of a string s (written as $|s|$) is the number of symbols in the string. For example $|abcd| = 4$.

A *prefix* of a string is any number of leading symbols in the string. For example, the string abc has prefixes ϵ, a, ab, and abc.

An *alphabet* is a finite set of symbols.

A *language* is a possibly infinite set of strings of symbols from an alphabet.

A *context-free grammar* (CFG) describes the strings of a language recursively in terms of a set of primitive symbols called *terminals* and a set of variables called *nonterminals*. Nonterminals are defined by a set of *grammar rules*, which specify that a nonterminal can be formed by taking the concatenation of a sequence of terminals and nonterminals. Grammar rules are of the form $X \rightarrow \alpha$, where the left-hand side (LHS) X is a nonterminal and the right-hand side (RHS) α is a (possibly empty) sequence of terminals and nonterminals.

Formally, a context-free grammar is denoted $G = (N, T, R, S)$, where N is a set of nonterminal symbols, T is a set of terminals (such that $N \cap T = \{\}$), R is a set of grammar rules, and the nonterminal S is the start symbol.

A grammar rule describes how a nonterminal can be expanded as a sequence of terminals and nonterminals. The expansion of the nonterminal X by the grammar rule X $\rightarrow \gamma$ is written αX$\beta \Longrightarrow \alpha\gamma\beta$. A sequence of these expansions form a *derivation*: $\alpha_1 \stackrel{*}{\Longrightarrow} \alpha_n$ means that α_1 derives α_n by 0 or more expansions. An *incomplete (rightmost) derivation* is an expansion αX $\stackrel{i}{\Longrightarrow} \alpha\beta$ where X $\rightarrow \beta\gamma$ is a grammar rule.

In natural language, the individual words of the language are categorised into a number of syntactic classes known as *preterminals*. For example, "John" is a noun. It is possible for a word to be more than one type of preterminal. For example, the word "book" can be either a verb or a noun. The CFG is written with these preterminals as the terminal symbols of the grammar, although individual words can still also appear as terminals in the grammar rules.

A special *sentinel* symbol $\$ \notin (N \cup T)$ is added to the end of the input string to facilitate easy end-of-string processing.

An example of a CFG is the following:

(1)	S	$\rightarrow$	NP VP	Grammar 1.1
(2)	NP	$\rightarrow$	Det N	
(3)	NP	$\rightarrow$	N	
(4)	VP	$\rightarrow$	V NP	

This CFG is a very simple grammar that generates sentences such as “Judi ate the apple”. The words of this sentence are pre-classified as preterminals, making the sentence “N V Det N”. The nonterminals of this grammar are S, NP, and VP, with S being the start symbol. Rules 2 and 3 describe two different ways an NP may be constructed, either as the expansion “N” or “Det N”.

Throughout this report, examples will be given to explain and illustrate points being made. The examples will involve a number of different context-free grammars, some of which will be arbitrary and have no immediate relation to natural language processing. These CFGs are presented because they are simple. They do, however, exemplify situations that arise in more complex grammars of natural language.

The following conventions in writing arbitrary grammars apply:

- The letters S, V, W, X, Y, and Z denote nonterminals, with S being the start symbol.
- The letters A, B, C, D, E, and F denote terminal symbols.
- The italic letters K, L, and M denote symbols that can be either terminals or nonterminals.
- The lowercase greek letters α, β, γ, and δ denote strings of symbols (either terminals or nonterminals).

1.6 Outline of this report

- Chapter 2 provides an introduction to GLR parsing.
- Chapter 3 describes the new L$*$ parsing framework and a method for compiling L$*$ parse tables for parsers defined within this framework.
- Chapter 4 examines the basic actions of L$*$ parsing and demonstrates their use with a simple example.
- Chapter 5 examines extensions to the basic L$*$ algorithm to make it a practical algorithm for natural language processing.
- Chapter 6 addresses the problem of compactly representing the parses of an L$*$ parser.
- Chapter 7 presents a summary of the new L$*$ parsing method, and concludes with a discussion of ongoing and possible future work.

Chapter 2

GLR Parsing

This chapter provides an introduction to GLR parsing. Section 2.1 provides an overview of the GLR parsing process. Sections 2.2 and 2.3 examine the data structures used in GLR parsing. Section 2.4 illustrates the GLR algorithm by tracing the execution of the algorithm on a simple example. Finally, section 2.5 presents a formal specification of the GLR algorithm.

2.1 Overview of GLR parsing

GLR parsing is an extension of LR parsing that can cope with arbitrary context-free grammars. The approach was developed by Tomita (1986) specifically for efficient parsing of natural language. A GLR parser is a *shift-reduce* parser: elements of the right-hand side of a grammar rule are *shifted* one by one onto a stack as they are recognised in the input. Complete right-hand sides are then *reduced*, or replaced by their corresponding left-hand sides. A GLR parser is also a *table-driven* parser. All actions for the parser to perform are pre-compiled and stored in a *parse table* which is then accessed during parsing to determine the parse actions to perform next.

A GLR parse table is constructed automatically from a context-free grammar by the same method used to construct an LR parse table (DeRemer, 1971). However, the restriction that each cell of an LR parse table contain only a single parse action is removed. Instead, a cell of a GLR parse table can contain multiple parse actions. This enables a GLR parser to parse with ambiguous grammars.

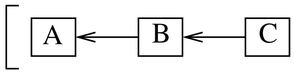
A GLR parser processes input from left to right, one word at a time, executing reduce and shift actions until an accept or error action is reached. If more than one action is specified at any point in a parse, any reduce actions are executed before shifts. All possible parses are pursued in parallel, using two main data structures: a *graph-structured stack* and a *packed shared forest*. To avoid confusion, an element of the graph-structured stack will be referred to as a *vertex*, and an element of the packed shared forest is referred to as a *node*.

2.2 The Graph-Structured Stack

The graph-structured stack is an extension of an LR stack that allows a GLR parser to deal with nondeterminism in a parse. Nondeterminism results from multiple actions in a cell of the parse table. In addition to the standard push and pop operations of an LR stack, the graph-structured stack has two new operations to handle nondeterminism: splitting and rejoining.

2.2.1 Splitting

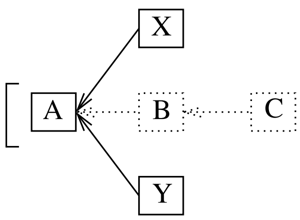
When there is more than one action to be performed at a stack top, the stack is split into multiple branches, one for each different result. For example, consider the following stack:



The stack grows from left to right, so A is the bottom of the stack, and C is the top of the stack. Given the grammar rules

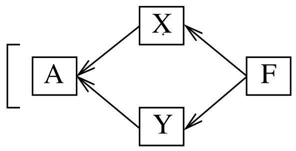
$$\begin{array}{lll} (1) & X \rightarrow & B\ C \\ (2) & Y \rightarrow & B\ C \end{array} \qquad \text{Grammar 2.1}$$

the B C at the top of the stack can be reduced to an X by rule 1 or a Y by rule 2. Upon performing these reductions, the parser splits the stack, with one branch for each alternative. The resulting stack after the reductions has two new stack tops X and Y.



2.2.2 Rejoining

If at some stage in the parse, the same vertex is to be pushed onto more than one stack top, then the corresponding branches of the stack are rejoined. For example, if F is shifted onto both stack tops X and Y from the previous figure, the resulting stack is



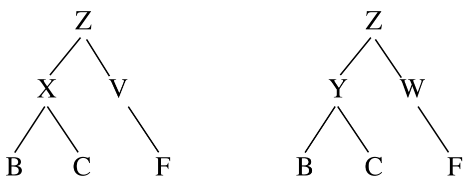
2.3 The Packed Shared Forest

A packed shared forest compactly represents all the possible parses of an input sentence. The forest is constructed so that all common subtrees are *shared*, and are represented only once in the forest. Subtrees that represent different parses of the same input substring are *packed* into a common root node that has multiple lists of children, one for each parse. This operation is called *local ambiguity packing.*

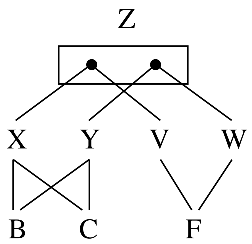
Continuing the example from the previous figure, if grammar 2.1 also contains the four rules

(3)	V	$\rightarrow$	F	
(4)	W	$\rightarrow$	F	Grammar 2.2
(5)	Z	$\rightarrow$	X V	
(6)	Z	$\rightarrow$	Y W	

then there are two possible derivations of Z, both of which cover the string “B C F”, but which are structured differently:

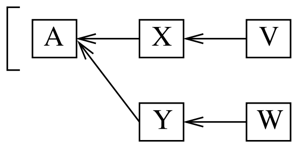


These two parses of Z can be *packed* together into a single node because the two parses both cover the same input substring. The forest node Z is referred to as a *packed node*, and has two lists of children, one corresponding to the expansion by rule 5, and the other corresponding to the expansion by rule 6. Also observe that B, C, and F are all shared among different parses—they have multiple parents in the forest.

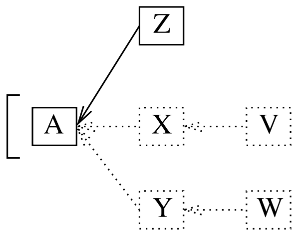


The decision to perform local ambiguity packing can be made using the configuration of the graph-structured stack. Local ambiguity packing is appropriate when two or more reductions to the same nonterminal are specified at the same input word and result in the same vertex. In this case the derivations resulting from the two reductions should be packed together in a single forest node.

For the above example, the stack before the reduction to Z by rules 5 and 6 will look like this:



Both reductions create a vertex labelled Z. Therefore the two derivations created by the reductions are packed together in a single forest node, and the resulting stack will look like this:



STATE	ACTION								
	N	Det	Prep	V	$	PP	NP	S	VP
0	s1	s3					g4	g2	
1			r3	r3	r3				
2			s5		acc	g12			
3	s11								
4			s5	s6		g8			g7
5	s1	s3					g10		
6	s1	s3					g9		
7			r1		r1				
8			r5	r5	r5				
9			s5, r7		r7	g8			
10			s5, r6	r6	r6	g8			
11			r4	r4	r4				
12			r2		r2				

Table 2.1: SLR(1) parse table for grammar 2.3.

2.4 An example of parsing with the GLR algorithm

To illustrate the GLR algorithm, this section presents a trace of the GLR algorithm parsing a simple sentence using grammar 2.3.

(1) S $\rightarrow$ NP VP
(2) S $\rightarrow$ S PP
(3) NP $\rightarrow$ N
(4) NP $\rightarrow$ Det N   Grammar 2.3
(5) NP $\rightarrow$ NP PP
(6) PP $\rightarrow$ Prep NP
(7) VP $\rightarrow$ V NP

This grammar is ambiguous because a PP may attach either to an S (by rule 2), or to an NP (by rule 5). For example, the sentence "John saw a man in the park" has two possible parses, corresponding to the two different attachments of the PP "in the park."

(1) `[John saw [a man [in the park]`$_{PP}$`]`$_{NP}$`]`$_{S}$

(2) `[John saw [a man]`$_{NP}$`[in the park]`$_{PP}$`]`$_{S}$

The two parses give rise to different interpretations. Parse 1 means that a man who was in the park was seen by John. Parse 2 means that John, who was in the park, saw a man.

Table 2.1 shows a parse table for grammar 2.3. The table is indexed by state number *st* and grammar symbol *L*. An entry ACTION[*st*, *L*] is a set of parse actions. A blank entry represents a parse error. Actions are written as follows:

- "`s` *n*" means shift and go to state *n*.
- "`r` *n*" means reduce by the *n*-th grammar rule.
- "`g` *n*," means go to state *n*.
- "`acc`" means accept.

To illustrate the GLR algorithm in detail, the state of the parser at each step is shown in a diagram with three components:

- *The graph-structured stack.*

  The graph-structured stack is a collection of vertices, each of which has an an associated state (an integer) and a parse forest node. Vertices are drawn with their state number inside them, and their associated parse forest node above them. Vertices that represent stack tops are drawn as circles. All other stack vertices are drawn as squares. The stack grows from left to right. Pending parse actions are entries from the parse table, and are placed to the right of the stack tops.

- *The parse forest.*

  Nodes in the parse forest are labelled with a grammar symbol and a subscript to distinguish different nodes with the same grammar symbol. Packed nodes are drawn as a box containing the different possible parses for the symbol of the node.

- *The current word.*

  The word the parser is currently processing is shown in a box at the right hand edge of the diagram.

The remainder of this section presents a detailed trace of the GLR algorithm parsing the sentence "John saw a man in the park".

Initially the parse forest is empty, and the stack contains only the single vertex with the start state of 0. The first word is "John," which is an N, so the first set of parse actions is ACTION[0, N] = {`s1`} (first diagram of figure 2.1). The parser therefore performs a shift action

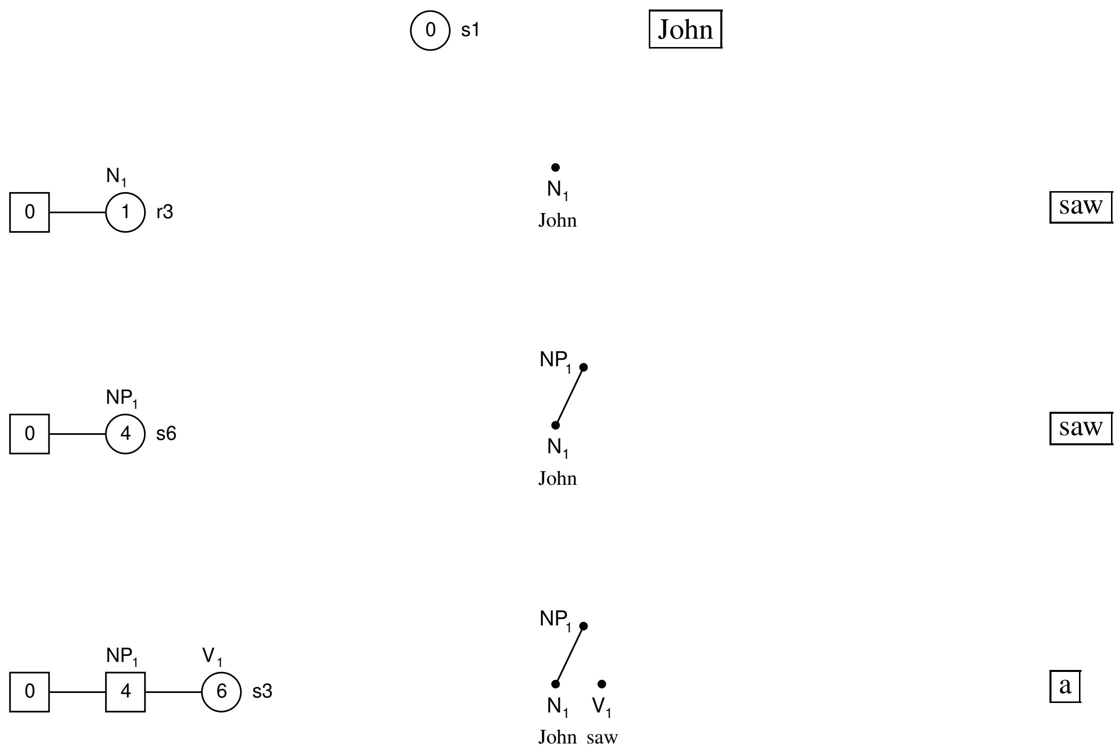


Figure 2.1: Trace of the GLR algorithm parsing "John saw a man in the park".

that creates a new stack vertex with state 1 and forest node N_1. The newly created vertex is pushed onto the stack, becoming the new stack top; hence it is drawn as a circle. The original vertex is redrawn as a square (second diagram of figure 2.1).

The next word to process is "saw," whose lexical category is V. The parse actions to perform are determined by the table entry ACTION[1, V] = $\{\mathtt{r3}\}$, so the parser performs a reduction by rule 3. The vertex labelled with N_1 at the top of the stack corresponds to the RHS of rule 3, and is popped off, to be replaced by a new vertex with state 4. The state of this new vertex is determined by ACTION[0, NP] = $\{\mathtt{g4}\}$. The parser uses state 0 to index the ACTION table because it is at the top of the stack after N_1 has been popped off. A new forest node NP_1 is also created, corresponding to the LHS of rule 3, and has as its child the single node N_1 that was popped off the stack (third diagram of figure 2.1). At the new vertex with state 4, the next actions to perform are given by ACTION[4, V] = $\{\mathtt{s6}\}$. Consequently, the verb "saw" is shifted onto the stack, creating a new vertex with state 6, and a new forest node V_1 (fourth diagram of figure 2.1).

The following words "a" and "man" are shifted onto the stack as shown in the first and second

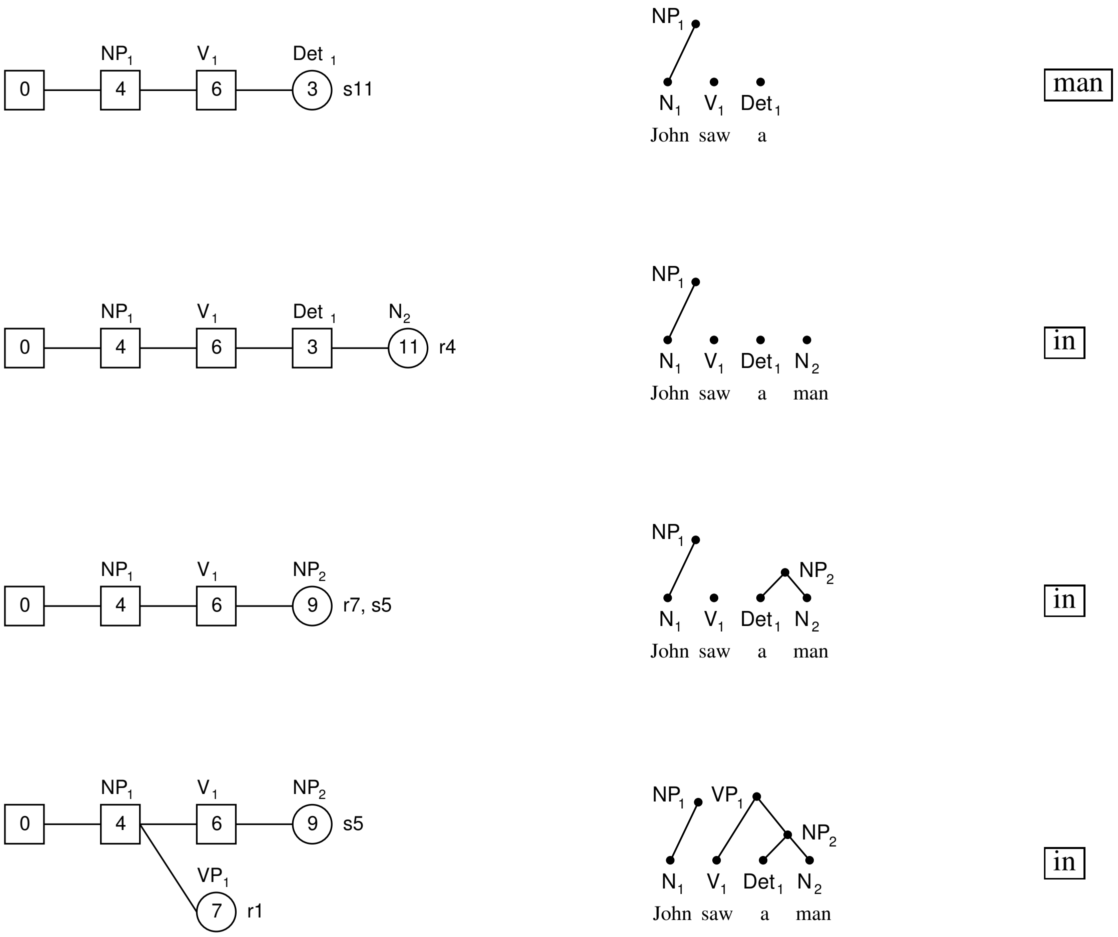


Figure 2.2: Trace of the GLR algorithm parsing "John saw a man in the park" (cont.).

diagram of figure 2.2, making "in", with lexical category Prep, the next word to process. The next action is ACTION[11, Prep] = {`r4`}, so Det_1 and N_2 are reduced to NP_2 by rule 4, creating a new stack vertex with state 9 and associated forest node NP_2 (third diagram of figure 2.2). At this point there are multiple parse actions to perform, because ACTION[9, Prep] = {`r7`, `s5`}. The GLR algorithm specifies that all reductions are processed before any shifts at a word, so the parser executes a reduce by rule 7, leaving the vertex with state 9 still active with an unprocessed shift action. The reduction causes the stack to split, creating a new stack top with state determined by ACTION[4, VP] = {`g7`} (fourth diagram of figure 2.2).

The parse actions to perform at this new vertex are ACTION[7, Prep] = {`r1`}. Again, the parser processes this reduction immediately, before performing the shift. The reduction by rule 1 creates the new forest node S_1 from NP_1 and VP_1, and a new stack vertex with state 2 (first diagram

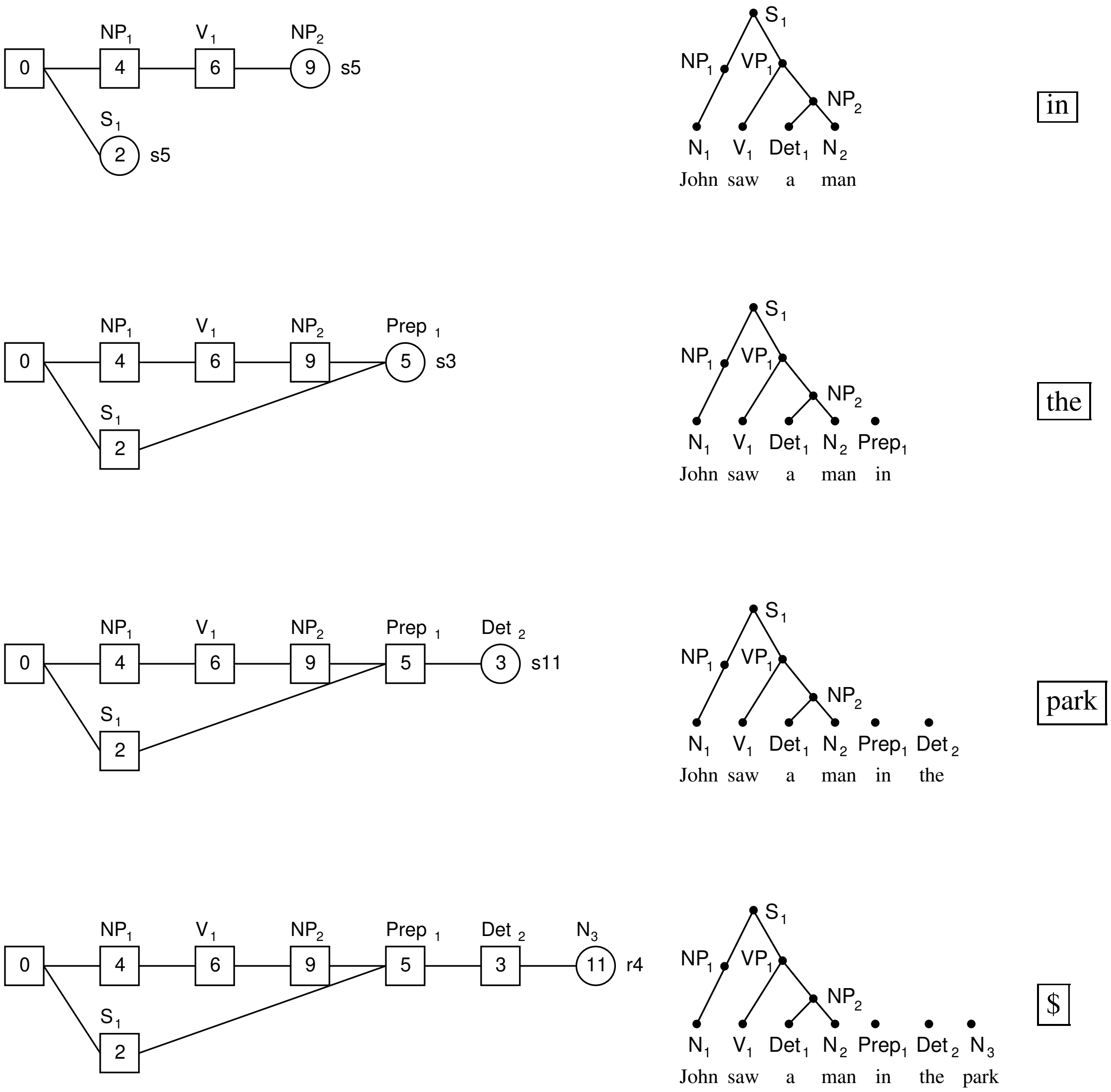


Figure 2.3: Trace of the GLR algorithm parsing "John saw a man in the park" (cont.).

of figure 2.3). At this new vertex, ACTION[2, Prep] = $\{\mathtt{s5}\}$. The only outstanding actions are now the s5 actions at the vertices with states 2 and 9, so the parser shifts the Prep "in" onto the stack. Both the shift actions create a vertex with the same state (being state 5) and associated forest node (being Prep_1), so the stack is rejoined and only a single new stack vertex with state 5 and associated forest node Prep_1 is created (second diagram of figure 2.3).

Next, the words "the" and "park" are shifted onto the stack (third and fourth diagrams of figure 2.3). This brings the parser to the end of the input sentence, so the next input word becomes

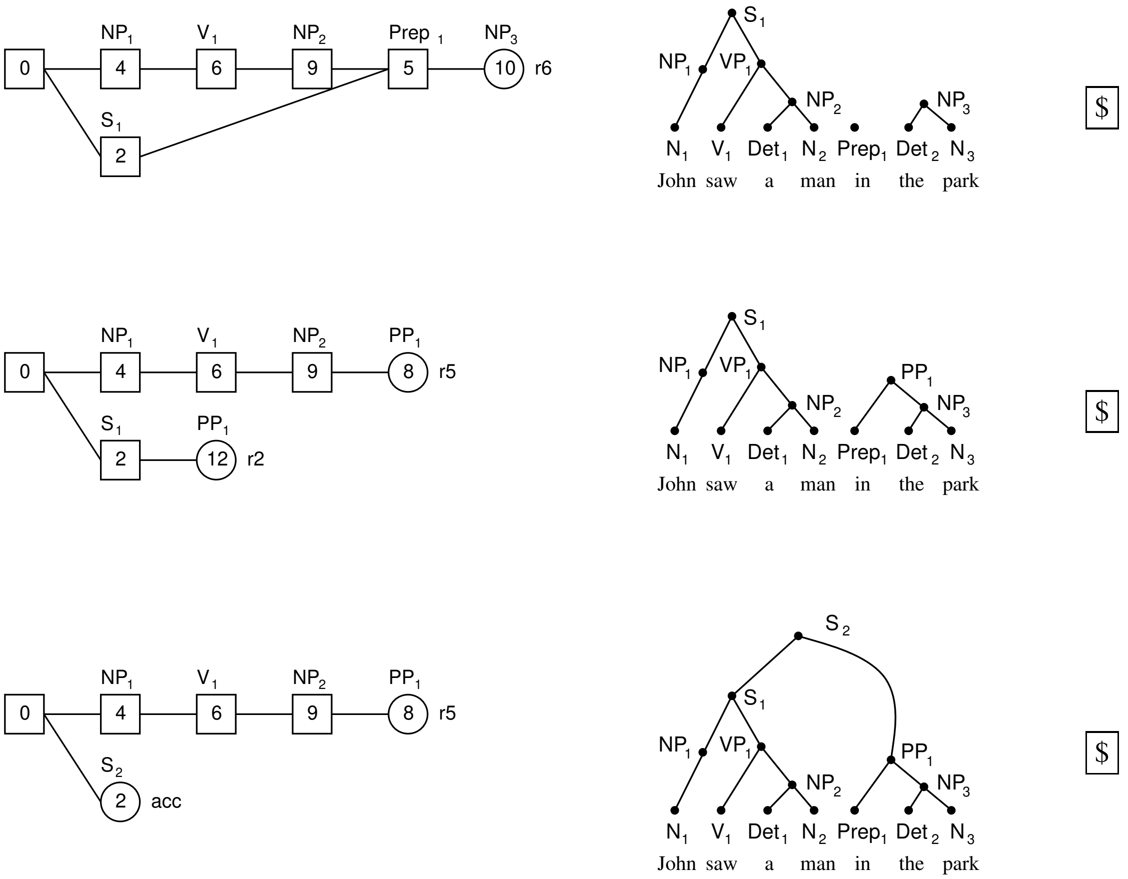


Figure 2.4: Trace of the GLR algorithm parsing “John saw a man in the park” (cont.).

the sentinel $\$$. A reduction by rule 4 creates NP_3 from Det_2 and N_3, leaving the stack as shown in the first diagram of figure 2.4. The next action to perform is $\text{ACTION}[10, \$] = \{\texttt{r6}\}$. There is only one path of length 2 starting from NP_3 in the stack (Prep_1 NP_3), so only the single new parse forest node PP_1 is created. However, there are two stack vertices joined to the end of this path, namely the vertices with states 2 and 9. As the goto values for these two states differ ($\text{ACTION}[9, \text{PP}] = \{\texttt{g8}\}$ and $\text{ACTION}[2, \text{PP}] = \{\texttt{g12}\}$), two new stack vertices are created. The vertex with state 8 is pushed onto the branch of the stack with the vertex with state 9 at the end. Similarly, the new vertex with state 12 is pushed onto the vertex with state 2. The same forest node PP_1 is associated with both these new vertices (second diagram of figure 2.4).

The actions to perform now are $\text{ACTION}[8, \$] = \{\texttt{r5}\}$ and $\text{ACTION}[12, \$] = \{\texttt{r2}\}$. The reduction by rule 2 is processed next, creating a new stack vertex with state 2 and new forest node S_2 (third diagram of figure 2.4). Because $\text{ACTION}[2, \$] = \{\texttt{acc}\}$, the node S_2 represents

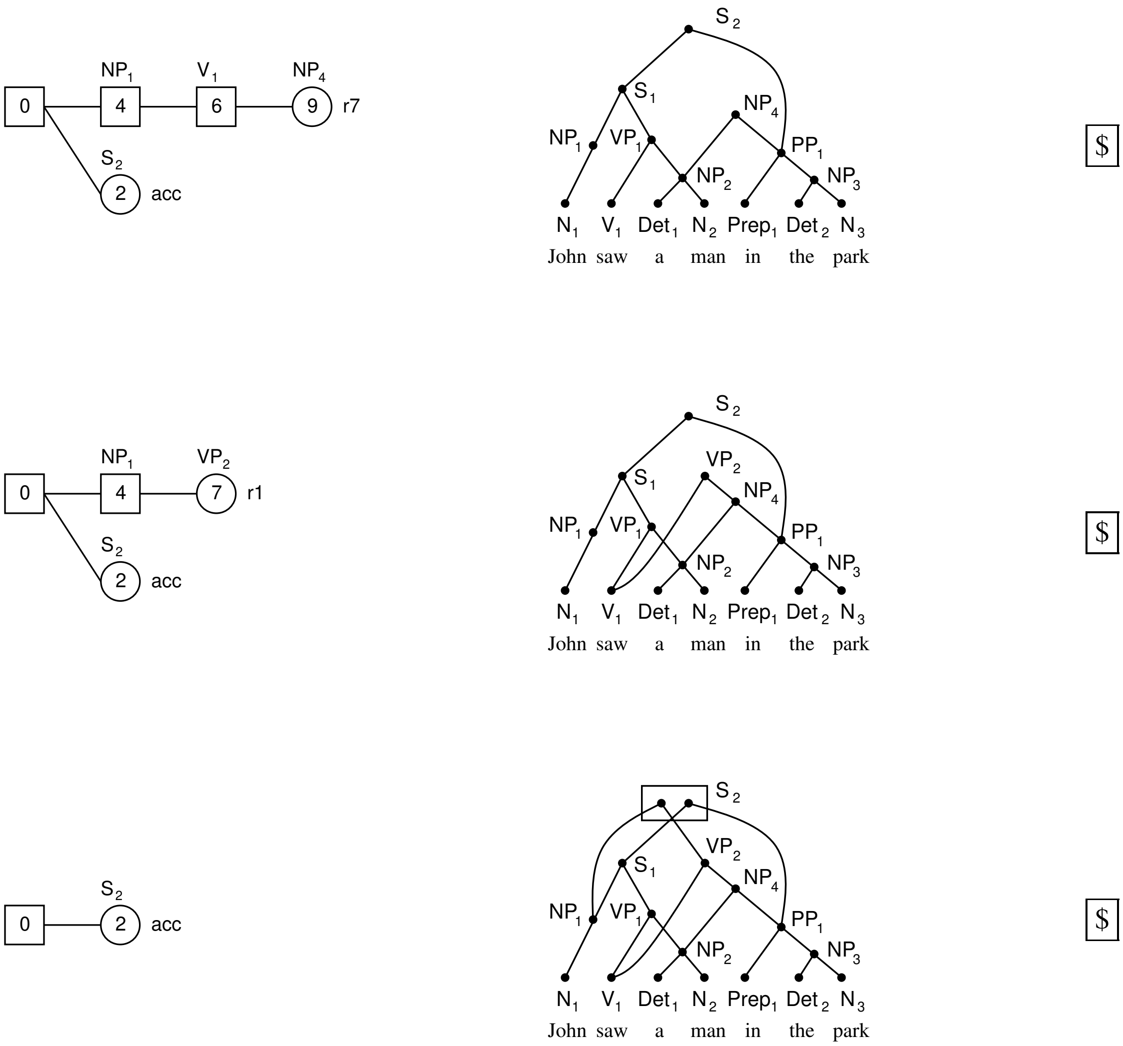


Figure 2.5: Trace of the GLR algorithm parsing "John saw a man in the park" (cont.).

a complete parse of the sentence "John saw the man." The parse is not yet finished, however, because there is still an outstanding reduction by rule 5 to be processed. Performing this reduction creates the new node NP_4, and new vertex with state 9 (first diagram of figure 2.5). This is followed by a reduction by rule 7, creating VP_2 (second diagram of figure 2.5). At this point, the next action is ACTION[7, \$] = {`r1`}. Executing this reduction creates an S from NP_1 and VP_2. The table entry ACTION[0, S] = {`g2`} determines the state of the vertex resulting from the reduction. There is already a stack top with state 2 and associated parse forest node S, so instead of creating a new forest node, the parser performs local ambiguity packing, and adds the new derivation from this reduction into forest node S_2. The only remaining action now is `acc`, so the parse is finished, and the final parse forest is rooted at node S_2 (third diagram of figure 2.5).

2.5 The Formal GLR Algorithm

Input

A parse table ACTION[*state*, *symbol*] for the context-free grammar $G = \langle N, T, R, S\rangle$ and an input string $z \in T^*$. Entries in the parse table are sets of parsing actions. Each action has the form "`shift` s", "`reduce` r", "`goto` s", or "`accept`". N is a set of nonterminals, T is a set of terminals, R is a set of grammar rules of the form $X \rightarrow \alpha$, where $X \in N$ and $\alpha \in (N \cup T)^+$, and S is the start symbol. The state s_0 is designated as the start state.

Output

The root node of a packed shared parse tree for z if $z \in L(G)$, otherwise an error indication.

Data Structures

A *vertex* in the graph-structured stack is a tuple $\langle s, n, \mathcal{S}\rangle$, where s is a state, n is a forest node, and $\mathcal{S}$ is the set of successor vertices in the stack. For notational convenience, the elements of a vertex v can be referenced using the functions $\mathsf{State}(v)$, $\mathsf{Node}(v)$, and $\mathsf{Successors}(v)$.

A *node* in the packed shared forest is a tuple $\langle X, \mathcal{D}\rangle$, where $X \in N$, and $\mathcal{D}$ is a set of lists of child forest nodes. The elements of a node n can be referenced using the functions $\mathsf{Nonterm}(n)$ and $\mathsf{Derivs}(n)$.

A *path* is a contiguous sequence of vertices $v_1, \ldots, v_k$ in the stack. That is, for $i = 2, \ldots, k$, $v_i \in \mathsf{Successors}(v_{i-1})$.

FRONTIER stores a set of pairs $\langle v, a\rangle$, where v is a vertex and a is a parse action yet to be performed at v. The vertices within the pairs of this list form the active stack tops.

CURRENT-V is a set of all vertices created by shifts or normal reductions while parsing an individual word, providing the set of vertices to check for packing.

ω denotes the current input symbol.

Main Loop

- Add a terminator symbol \$ to the end of the input string z
- $\omega \leftarrow$ The first symbol of z
- $v_0 \leftarrow \langle s_0, \text{NIL}, \{\}\rangle$ — Initialise the stack
- CURRENT-V $\leftarrow \{v_0\}$
- Call **Schedule**(v_0, ω)
- Loop — Perform reductions followed by shifts until acceptance or rejection.
  - Call **Reduce**()
  - Call **Shift**()
  - If FRONTIER = $\{\langle v, \text{“}\texttt{accept}\text{”}\rangle\}$ then halt and return $\mathsf{Node}(v)$
  - If FRONTIER = $\{\}$ then halt and signal an error

Reduce()

Perform all reductions licensed by an individual word.

- While $\exists x \in$ FRONTIER of the form $\langle v, \text{“}\texttt{reduce}\ X \rightarrow \alpha\text{”}\rangle$:
  - Remove x from FRONTIER
  - $\mathcal{P} \leftarrow \{p \mid p$ is a path of length $|\alpha|$ starting at v in the stack$\}$
  - $\forall p \in \mathcal{P}$, call **Reduce-Path**$(p, X \rightarrow \alpha)$

Reduce-Path$(v_1, \ldots, v_k, X \rightarrow \alpha)$

Perform a reduction by the rule $X \rightarrow \alpha$ along the path $v_1, \ldots, v_k$ in the stack, creating a new derivation consisting of the nodes of vertices $v_1, \ldots, v_k$.

- $d \leftarrow \langle(\mathsf{Node}(v_k), \ldots, \mathsf{Node}(v_1))\rangle$
- $\Pi \leftarrow$ A partition of $\mathsf{Successors}(v_k)$ by goto value on symbol X
- If $\exists v \in$ CURRENT-V s.t. $\mathsf{Successors}(v) \in \Pi \wedge \mathsf{Nonterm}(\mathsf{Node}(v)) = X$ — Pack using an existing vertex in CURRENT-V whose nonterminal and left-context match the current reduction.
  - Add d to $\mathsf{Derivs}(\mathsf{Node}(v))$

  Else
  - $n' \leftarrow \langle X, \{d\}\rangle$ — Create a new forest node
  - $\forall \pi_s \in \Pi$ — Create new vertices containing this node, one for each element of Π
    - $v' \leftarrow \langle s, n', \pi_s\rangle$
    - Add v' to CURRENT-V
    - Call **Schedule**(v', ω)

Shift()

Shift the next terminal symbol onto all the stack tops, creating a new node for it in the parse forest, and schedule any actions triggered by the shift.

- $n \leftarrow \langle \omega, \{\} \rangle$
- $\omega \leftarrow$ The next symbol of the input string
- CURRENT-V $\leftarrow \{\}$
- $\mathcal{S} \leftarrow \{\langle v, a \rangle \in$ FRONTIER $\mid a =$ "`shift` s"$\}$
- FRONTIER $\leftarrow$ FRONTIER $- \mathcal{S}$
- $\Pi \leftarrow$ A partition of $\mathcal{S}$ according to goto state of the `shift` actions
- $\forall \pi_s \in \Pi$
  - $\mathcal{V} \leftarrow \{v \mid \langle v, a \rangle \in \pi_s\}$
  - $v_s \leftarrow \langle s, n, \mathcal{V} \rangle$
  - Add v_s to CURRENT-V
  - Call **Schedule**(v_s, ω)

- Create a new node for the shifted input symbol.
- $\mathcal{S}$ is the set of all shift actions to perform.
- Each $\pi_s \in \Pi$ consists of elements of the form $\langle v,$ "`shift` s" $\rangle$.
- Create a new vertex for each member of Π.

Schedule(v, L)

Add to FRONTIER all possible actions to be performed at vertex v.

- $\forall a \in$ ACTION[State$(v), L$]
  - Add $\langle v, a \rangle$ to FRONTIER

□

Chapter 3

Building Tables for L∗ Parsing

Close interaction between modules of an NLP system is important for the efficient resolution of ambiguity. This chapter examines a method of building parsers to facilitate this interaction. Section 3.1 describes how top-down and bottom-up parsers act in a modular NLP system. Section 3.2 then describes the new L∗ parsing framework for defining table-driven parsers, and presents a particular parser within the framework that is designed to be efficient in a modular NLP system. Section 3.3 describes building tables for LR parsing, to provide a background for section 3.4, which describes a method of compiling L∗ parse tables. Section 3.5 addresses the problem of left-recursion. Section 3.6 presents a formal algorithm for building L∗ parse tables.

3.1 Parsing in a modular system

The ambiguity of natural language means that an NLP system must search through a large set of partial interpretations of a sentence for one that is syntactically, semantically, and pragmatically plausible. For this search to be efficient, it must be as focused as possible, using all available information as early as possible in processing (Birnbaum, 1986). In a syntactic context, this means it is desirable that a parser generate syntactic hypotheses (partial parses) as early as possible in processing. These syntactic hypotheses can then help guide the system's search for an interpretation of the sentence. Hence, the accuracy of the syntactic hypotheses directly affects the efficiency of the search. In particular, incorrect hypotheses will misdirect the system and cause it to perform extra useless work. There is a trade-off between the desire to generate syntactic hypotheses early, and the desire to make them accurate. The earlier that hypotheses are generated, the less information they are based on and therefore the more likely they are to be incorrect.

There are two standard styles of parsing—top-down and bottom-up (Aho and Ullman, 1977). A top-down parser generates syntactic hypotheses very early in processing a sentence because it expands grammar rules before parsing the symbols of the rule. The hypotheses are not strongly based on the actual input to the parser, however, so they may be inaccurate and therefore misdirect the search.

In contrast, a bottom-up parser generates more accurate syntactic hypotheses, because it only generates these hypotheses once all input covered by the hypotheses has been processed. Hence a bottom-up parser does not allow all available information to be applied as early as possible in processing. For example, consider the sentences

(1a) The officers taught at the academy were very demanding.

(1b) The courses taught at the academy were very demanding.

Sentence 1a is a garden path sentence: humans initially assume the verb "taught" to be the main verb of the sentence, and only change that assumption when they read "were" later in the sentence. In contrast, with the subject noun changed from "officers" to "courses", as in sentence 1b, the garden path effect is removed. This is because of the knowledge that courses can only be taught and cannot teach. In the interests of efficiency, this knowledge should be applied as soon as the verb "taught" is read, so that only the reduced relative clause parse is pursued, however, a bottom-up parser does not do this.

(1)	S	$\rightarrow$	NP VP	
(2)	RCl	$\rightarrow$	VP	
(3)	NG	$\rightarrow$	Det N	
(4)	NP	$\rightarrow$	NG	
(5)	NP	$\rightarrow$	NG RCl	Grammar 3.1
(6)	VP	$\rightarrow$	V PP	
(7)	VP	$\rightarrow$	V Adv Adj	
(8)	PP	$\rightarrow$	Prep NG	

Consider how a bottom-up parser processes sentence 1b according to grammar 3.1. First, it reads "the courses" as a Det and N, which it reduces to an NG by rule 3, then an NP by rule 4. Next it reads "taught" as a V. At this point, enough input has been processed to use the knowledge that courses cannot teach to rule out the interpretation that has "taught" as the main verb of the sentence. This knowledge can be applied when the parser attempts to construct a complete sentence with “taught” as the main verb, by a reduction using rule 1. However, before reducing by rule 1, a bottom-up parser must first parse “at the academy” as a PP and attach it to the VP by a reduction using rule 6. It follows that the knowledge that courses cannot teach is not applied as early as possible in processing, and therefore causes the system more work.

The deficiencies of top-down and bottom-up parsers described above arise because of the strict control strategy of the parsers. A top-down parser always proposes application of a rule before any members of the rule have been processed. A bottom-up parser only proposes reduction by a grammar rule once all constituents of the rule have been parsed.

3.2 Eager reduction

A more general method of parsing is to allow reduction by a grammar rule when the first k symbols of the rule have been processed. The value of k can differ for different grammar rules. This allows rules to be reduced at the point which produces syntactic hypotheses that best focus the search of the NLP system.

This approach defines a framework in which a variety of control strategies for a parser can be specified, depending on when grammar rules are applied. This framework covers both top-down and bottom-up parsers. If every rule is applied before any RHS symbols have been parsed, the result is a top-down parser. If every rule is applied only after all RHS symbols have been parsed, the result is a bottom-up parser.

One way of implementing this approach is to modify a bottom-up parser to allow partial or *eager* reductions. An eager reduction is similar to a bottom-up or *full* reduction except it is carried out before all members of the RHS of a rule have been parsed. The eager reduction creates an *incomplete* derivation, where the derivation resulting from the reduction is missing some of its children. These children may then be derived from later input and *combined* into the incomplete derivation. The result is an *L∗ parser*, so called because the parser reads input from *L*eft-to-right, but allows derivations to be constructed in any fashion—the “∗” represents a wildcard.

An eager reduction introduces a predictive component into a bottom-up parser. Having seen some portion of the RHS of a grammar rule, an L∗ parser will predict that the rule is applicable, and will reduce without waiting to see the complete RHS of the rule as a bottom-up parser would.

Any method that reduces by a rule after only seeing a prefix of the rule may propose syntactic analyses that are inconsistent with the larger sentence and that are not completely justified by the input, as in top-down parsing. Thus in an entirely syntactic context, such a method can never perform better than a bottom-up parser. However, in the context of a larger NLP system, this method has the potential for greater efficiency because it allows the earlier resolution of ambiguity on semantic or pragmatic grounds.

Because eager reductions introduce more hypotheses into an NLP system, it is important they are proposed only when they are likely to provide useful information that can improve the efficiency of the system. In particular, eager reductions should be performed when the resulting syntactic hypotheses are likely to afford the greatest semantic leverage, that is the earliest points at which semantic processing can perform useful work with the resulting syntactic hypotheses.

3.2.1 Semantic heads and semantic attachment

One approach for determining the useful points for eager reduction in a grammar is to consider when useful semantic work happens and how this can be triggered by syntactic processing.

A common method for representing the semantics of a phrase is to use a frame-based system. A frame has a number of slots, each of which has a number of constraints on the values that may fill that slot.

For example, consider the following frame:

```
teach is-a action with
  teacher: human
  student: human
  subject-matter: course
```

This frame represents the action of teaching. It specifies that “teach” is an action. There are three things associated with teaching—the teacher, the student, and the subject matter. These are the slots of the teach frame. Both the teacher and the student are constrained to be humans, and the subject matter is constrained to be a course.

Individual instances of the teaching frame are created to represent specific situations. For example, the following frame might represent the semantics of the sentence "John teaches the officers marching":

```
teach-5 inst-of teach with
  teacher := john-7
  student := officers-2
  subject-matter := marching-101
```

The frame `teach-5` is an instance of the teach frame, with the teacher slot having the value `john-7`, the student slot being `officers-2`, and the subject-matter being `marching-101`. Each of the slot fillers is itself an instance of a frame. For example, `john-7` may be an instance of the `male` frame.

Whenever a slot of a frame is filled with a particular value, checks are performed to determine that the value meets all the constraints on the slot. For example, when `john-7` fills the `teacher` slot of `teach-5`, the constraint that `john-7` is `human` should be checked. If these constraints are violated, then the particular meaning can be discarded. Thus useful semantic work can be done when the slots of a frame are filled.

The *semantic head* of a phrase is the word whose lexical entry is used to construct the top-level frame in a representation of the meaning of the phrase. In the above example, "teaches" is the semantic head of the sentence "John teaches the officers marching" because `teach-5` is the top-level frame of the representation for this sentence.

Words whose lexical entries specify values that fill slots of a frame are said to *semantically attach* to the head. For example, "the officers" semantically attaches to "teaches" because the value `officers-2` fills the `student` slot of `teach-5`. Useful semantic work occurs when semantic attachments are established, because constraints on the slot are checked when a semantic attachment is established. Hence one case where an eager reduction can be beneficial is when it causes a semantic attachment.

3.2.2 Syntactic heads and syntactic attachment

Unfortunately, a syntactic parser cannot determine whether a reduction will generate a semantic attachment, because the semantic representations necessary to make such a decision are not available to the parser. This problem can be solved, however, by defining a syntactic correlate to the semantic head and semantic attachment described in section 3.2.1. The *syntactic head* (or just *head*) of a grammar rule is an element of the RHS of that rule, chosen so as to dominate the semantic heads of as large a proportion of the phrases derivable from it as possible. A nonterminal Y of the rule X $\rightarrow \alpha \mathrm{Y} \beta$ *dominates* the semantic head a_i of a phrase $a_1, \ldots, a_n$ if

$$\begin{array}{rcl} \mathrm{X} & \Longrightarrow & \alpha \mathrm{Y} \beta \\ & \overset{*}{\Longrightarrow} & \alpha \alpha' a_i \beta' \beta \\ & \overset{*}{\Longrightarrow} & a_1, \ldots, a_n \end{array}$$

The head of a rule can be indicated by annotating the symbol in the rule with a subscripted *h*. For example, if VP is the head of the rule S $\rightarrow$ NP VP, then this is written S $\rightarrow$ NP VP_h. Each grammar rule has a unique head. There may be no explicit annotation on a grammar rule. In this case, the last element of the rule is treated as the *implicit head* of the rule.

Semantic attachments established by filling the slots of a frame can be mapped into the syntactic domain using the concept of *syntactic attachment*. All non-head elements of a grammar rule syntactically attach to the head of the rule. There are two kinds of attachment. For a grammar rule X $\rightarrow \alpha$ Y_h β, all symbols in α *left-attach* to Y, and all symbols in β *right-attach* to Y.

3.2.3 When to eagerly reduce

The definitions of syntactic head and syntactic attachment provides a purely syntactic notion of when a reduction is likely to facilitate semantic processing. Therefore, the parser should perform an eager reduction upon processing the syntactic head of a rule whenever doing so will cause a syntactic attachment with no further consumption of input.

For example, given the grammar rule

$$(1) \quad \mathrm{X} \rightarrow \mathrm{Y}\ \mathrm{Z}_h\ \alpha$$

the parser should perform an eager reduction after parsing Z, because doing so will generate an attachment between Y and Z. This differs from a bottom-up parser, which would delay until α had been completely parsed before performing a reduction by rule 1.

It may also be the case that an eager reduction only indirectly causes an attachment. For example, given the grammar rules

$$\begin{array}{lrcl} (1) & \mathrm{X} & \rightarrow & \mathrm{Y}\ \mathrm{Z}_h \\ (2) & \mathrm{Z} & \rightarrow & \mathrm{A}_h\ \beta \end{array}$$

there is a left-attachment of Y to Z, which can be established once the head of Z is parsed. Eagerly reducing by rule 2 once A has been parsed constructs a Z, which can then be used in a reduction by rule 1, to generate an attachment between Y and Z. Thus, eagerly reducing by rule 2 generates an attachment by rule 1.

Whether or not an eager reduction generates an attachment may depend on context. For example, consider the following grammar:

$$\begin{array}{lccl} (1) & \mathrm{S} & \rightarrow & \mathrm{X}\ \mathrm{Y}_h \\ (2) & \mathrm{Y} & \rightarrow & \mathrm{X}_h\ \mathrm{C} \\ (3) & \mathrm{X} & \rightarrow & \mathrm{A}_h\ \mathrm{B} \end{array}$$

The first expected symbol is an X, by rule 1. This X left-attaches to Y (the head of rule 1), but this attachment cannot be made until the head of Y is parsed. Hence an eager reduction by rule 3 does not generate an attachment between X and Y in this case, so rule 3 should be processed in normal bottom-up fashion, waiting until both A and B are parsed before performing a reduction. After X is parsed, the parser works towards parsing a Y. Eagerly reducing after seeing the head of rule 2 generates an attachment of X to Y by rule 1. In this case, an eager reduction by rule 3 creates an X which can be in turn eagerly reduced by rule 2 to create a Y, to which the first X left-attaches by rule 1. Thus an eager reduction by rule 3 should be carried out by the parser in this case. This demonstrates that in the context of parsing an S, eagerly reducing by rule 3 generates no attachment, whereas in the context of parsing a Y, an eager reduction by rule 3 will generate an attachment.

3.3 Building LR parse tables

The specification of when to eagerly reduce must be stated in a form that can be used in a table-driven parser. This means compiling the chosen eager reduction strategy into the parse table. To explain how this is done, it is first necessary to understand how an LR parse table is constructed.

3.3.1 LR finite automata

An LR parse table encodes a finite automaton that recognises the valid items for a viable prefix of a grammar. A viable prefix is a partial derivation that can always be extended into a complete parse. When used in conjunction with a stack, the result is a push-down automaton that parses the grammar. States of the automaton contain sufficient information to decide when to reduce.

Conceptually, a suitable finite automaton can be constructed in a two step process. Firstly, a nondeterministic finite automaton (NFA) is defined that recognises the viable prefixes of a grammar. Secondly, the subset construction method (Aho and Ullman, 1977) is applied to this NFA to define a deterministic finite automaton (DFA) from which the LR parse table can be constructed.

The NFA is constructed from a grammar $G = (N, T, R, S)$, where N is a set of non-terminal symbols, T is a set of terminals (such that $N \cap T = \{\}$), R is a set of grammar rules, and S is the start symbol. To allow detection of the completion of a parse, the grammar G is augmented by adding a new production $S' \to S$, creating the *augmented grammar* $G' = (N, T, R \cup \{S' \to S\}, S')$. S' is a new symbol not in $N \cup T$.

Formally, the NFA recognising viable prefixes of a grammar is defined by a 5-tuple $(Q, I, \delta, q_0, Q\prime)$, where Q is the set of states of the NFA, I is the set of input symbols, δ is the transition function, q_0 is the start state, and $Q\prime (\subseteq Q)$ is the set of accepting states. A finite automaton can be represented by a directed graph called a *transition diagram*. The states of the finite automaton are the vertices of the graph. For each element of the transition function $q_2 \in \delta(q_1,\ a)$, there is a directed arc from state q_1 to state q_2 labelled with symbol a.

The states of the NFA are the *items* of a grammar. An *LR(0) item* is defined to be a grammar rule, together with a dot at some position of the RHS of the rule. For example, the possible items for the grammar rule S $\to$ NP VP are

[S $\to$ $\cdot$ NP VP]
[S $\to$ NP $\cdot$ VP]
[S $\to$ NP VP $\cdot$]

Intuitively, an item represents how far the parser is through parsing a grammar rule. For example, the item [S $\to \cdot$ NP VP] indicates that the parser expects a string derivable from NP VP next in the input. Similarly, the item [S $\to$ NP $\cdot$ VP] represents the case where the parser has derived an NP from the input, and is expecting to find a VP next.

The NFA M recognising viable prefixes for a grammar G' is $M = (Q, N \cup T, \delta, q_0, Q)$, where the set of states Q of the NFA is the set of LR(0) items for G', the initial state q_0 is the item [S$'$ $\to \cdot$ S], and the transition function δ is defined as:

1. $\delta([\mathrm{X} \to \alpha \cdot\ L\, \beta],\ L) = \{[\mathrm{X} \to \alpha\ L\ \cdot\ \beta]\}$
2. $\delta([\mathrm{X} \to \alpha \cdot\ \mathrm{Y}\, \beta],\ \epsilon) = \{[\mathrm{Y} \to \cdot \gamma] \mid \mathrm{Y} \to \gamma \in R\}$.

Intuitively, rule 1 corresponds to the situation where the parser is in a state where it has so far recognised α from the grammar rule X $\rightarrow \alpha\ L\ \beta$. If L is then derived, the portion of this grammar rule recognised can be extended to αL, represented by the item where the dot has been moved over the L. Rule 2 corresponds to the case where the parser is expecting to see Y next in the input, and Y derives γ, so it must also be expecting to see γ next in the input.

For example, consider the following grammar:

(1)	S	$\rightarrow$	NP VP	
(2)	VP	$\rightarrow$	V NP	Grammar 3.2
(3)	NP	$\rightarrow$	N	

The transition diagram of the NFA recognising viable prefixes for grammar 3.2 is shown in figure 3.1. The arc from [S $\rightarrow \cdot$ NP VP] to [S $\rightarrow$ NP $\cdot$ VP] is an example of rule 1 of the transition function, with X = S, $\alpha = \epsilon$, L = NP, and β = VP. Similarly, the arc from [S $\rightarrow$ NP $\cdot$ VP] to [VP $\rightarrow \cdot$ V NP] is an example of rule 2 of the transition function, with X = S, α = NP, Y = VP, $\beta = \epsilon$, and γ = V NP.

A DFA that recognises viable prefixes can be constructed by applying the subset construction algorithm to the NFA. After inputs $a_1, \ldots, a_n$, the DFA is in a state that represents the set of possible states in the NFA that can be reached with input $a_1, \ldots, a_n$. For example, the DFA constructed from the NFA in figure 3.1 is shown in figure 3.2. The start state (state 0) of the DFA is constructed by taking the start state of the NFA [S$'$ $\rightarrow \cdot$ S] and adding the states [S $\rightarrow \cdot$ NP VP] and [NP $\rightarrow \cdot$ N], because both these states can be reached by ϵ-transitions from the start state of the NFA.

In practice, the two steps of creating the NFA and then applying the subset construction algorithm are combined into a single procedure for building a DFA.

There are more sophisticated finite automaton construction techniques that augment the definition of an LR(0) item with *lookahead*: items now include information about the terminal symbols that can possibly follow them. The number of symbols used for lookahead is described by the number of the item. An LR(0) item means the item includes 0 lookahead symbols. An LR(1) item includes one symbol of lookahead in each item, and is of the form [X $\rightarrow \alpha$, t], where t is a

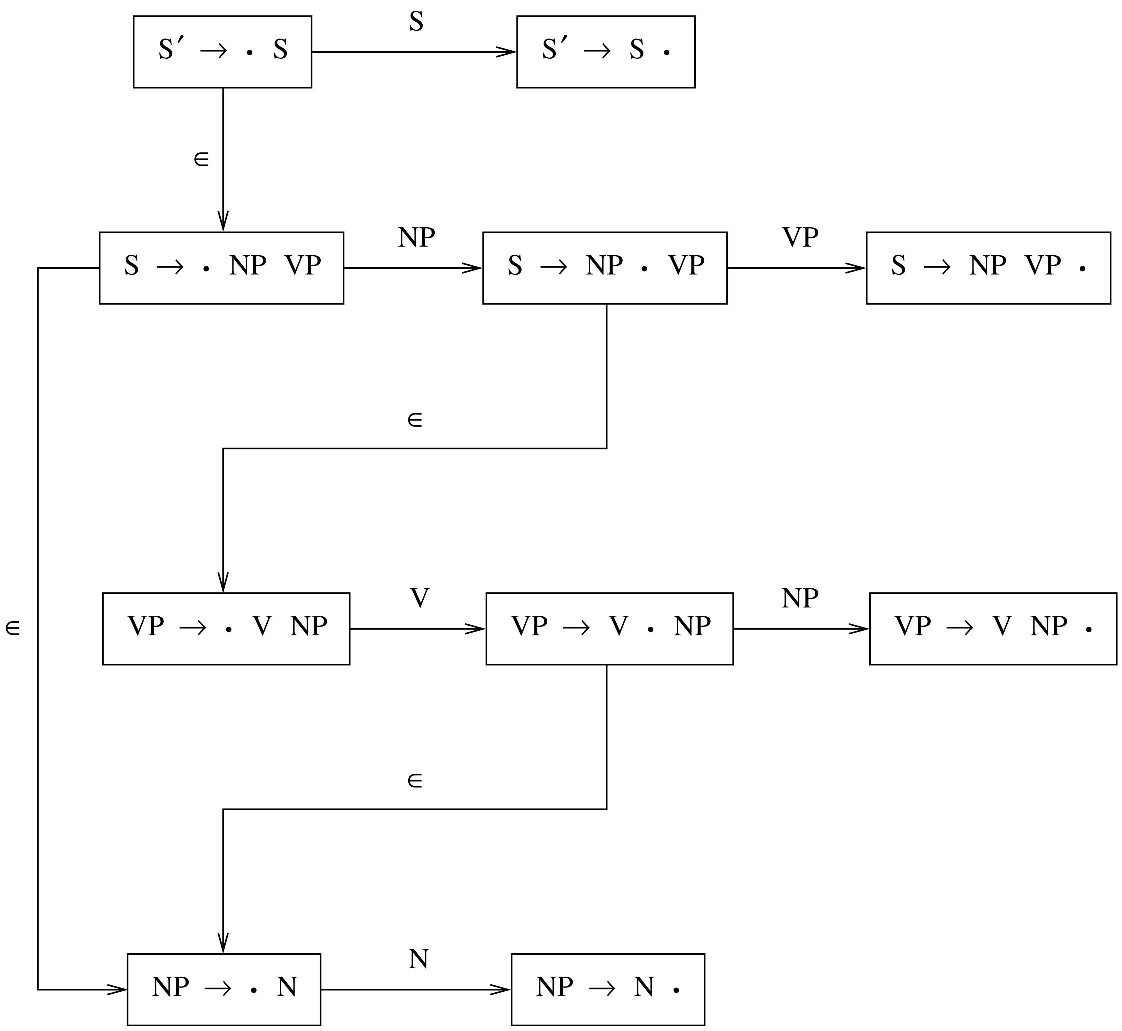


Figure 3.1: LR(0) NFA for recognising viable prefixes of grammar 3.2.

terminal symbol that may follow the item. For example, with the grammar rules

(1) S $\rightarrow$ X Y
(2) X $\rightarrow$ A B
(3) Y $\rightarrow$ C D

the LR(1) items for rule 2 would be

[X$\rightarrow$ $\cdot$ A B, C]
[X$\rightarrow$ A $\cdot$ B, C]
[X$\rightarrow$ A B $\cdot$, C]

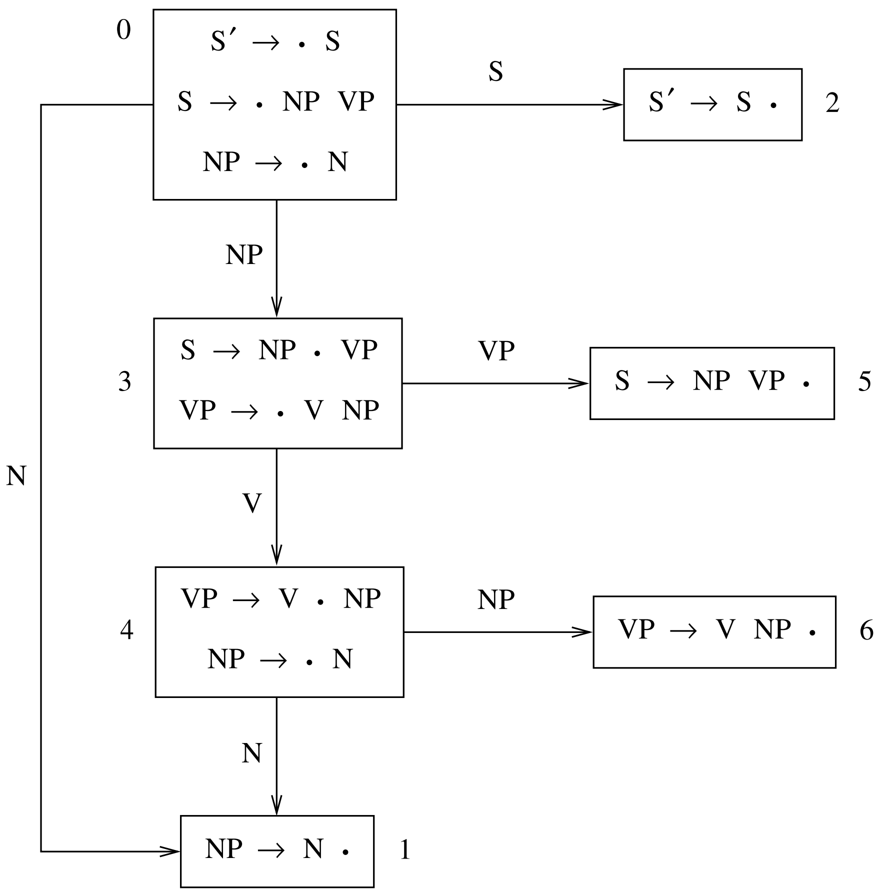


Figure 3.2: LR(0) DFA for recognising viable prefixes of grammar 3.2.

Because Y always follows X by rule 1, and C is always the first symbol of Y by rule 3, C is the only terminal symbol that follows X. Hence C is the lookahead of the LR(1) items for X → A B. The lookahead symbols can be used when the parse table is constructed to better determine which parse actions to perform for a given input.

3.3.2 Building LR parse tables from finite automata

An LR parse table can be constructed from a DFA that recognises viable prefixes. The parse table specifies the set of actions to perform in some state when a given string of lookahead symbols is next in the input. A longer lookahead provides a better determination of the actions to perform for a given input, but can increase the size of the parse table dramatically. Varying the length of lookahead defines the family of LR parse tables, where the length of lookahead is given by k in the description LR(k). Parse tables constructed from a DFA of LR(0) items are a special case. Although the items of the DFA contain no lookahead, by using the function FOLLOW(X), some lookahead can still be used in constructing the parse table. A parse table constructed from a DFA of LR(0) items is known as an SLR(k) table, where the k in this case refers to the number of symbols given by the FOLLOW function.

The auxiliary functions FIRST and FOLLOW used when building SLR tables are defined as:

- FIRST(α) is the set of terminals that begin strings derived from α.
- FOLLOW(X), for a nonterminal X, is the set of terminals that can directly follow X.

For example, consider the following grammar:

(1) S $\rightarrow$ X Y
(2) X $\rightarrow$ A B
(3) Y $\rightarrow$ C D
(4) Y $\rightarrow$ E F

Then FIRST(X) = $\{$A$\}$, because X always starts with an A by rule 2, and FOLLOW(X) = $\{$C, E$\}$, because a Y always follows an X by rule 1 and the first symbol of a Y is either a C (by rule 3) or an E (by rule 4).

The parse table encodes the states of the DFA as rows of the table. *Goto* and *shift* actions encode the transition function of the DFA. Shift actions represent transitions by terminal symbols, and goto actions represent transitions by nonterminal symbols. *Reduce* actions apply grammar rules when the symbols of the right-hand side have been derived from the input. The *accept* action specifies that the parser should indicate successful completion of the parse.

To generate the actions for a state I of the DFA:

1. If the item [S$'$ $\rightarrow$ S $\cdot$] $\in I$, add the action `accept` to ACTION[I, \$].
2. For all items of the form [X $\rightarrow \alpha\ \cdot$] $\in I$, add the action `reduce` X $\rightarrow \alpha$ to ACTION[I, x] for every $x \in$ FOLLOW(X).
3. For $\delta(I, t) = J$, where t is a terminal symbol, add the action `shift` J to ACTION[I, t].
4. For $\delta(I, X) = J$, where X is a nonterminal symbol, add the action `goto` J to ACTION[I, X].

Table 3.1 shows the SLR(1) parse table constructed from the DFA in figure 3.2 by rules 1–4 above. The table is indexed by state number *st* and grammar symbol *L*. An entry ACTION[*st*, *L*] is a set of parse actions. A blank entry represents a parse error. Actions are written as follows:

- "`s` *n*" means shift and goto state *n*.
- "`r` *n*" means reduce by the *n*-th grammar rule.
- "`g` *n*," means goto state *n*.
- "`acc`" means accept.

To exemplify the process of table construction, consider constructing the table row for state 4 of the DFA shown in figure 3.2. By rule 3, the action `s1` is added to ACTION[4, N], because $\delta(4, \mathrm{N}) = 1$, and N is a terminal. Similarly, the action `g6` is added to ACTION[4, NP] by rule 4, because $\delta(4, \mathrm{NP}) = 6$, and NP is a nonterminal.

STATE	ACTION					
	N	V	$	S	VP	NP
0	s1			g2		g3
1		r3	r3			
2			acc			
3		s4			g5	
4	s1					g6
5			r1			
6			r2			

Table 3.1: SLR(1) parse table for grammar 3.2.

3.4 Building L$*$ parse tables

The L$*$ parse table should be compiled so that eager reductions are specified whenever they will cause a syntactic attachment with no further consumption of input. Thus the table builder needs a method of evaluating whether or not performing an eager reduction in a given state would generate an attachment. This is a problem when building the parse table from an LR finite automaton. Whether an attachment results from applying a grammar rule depends on context not encoded in an LR item. Thus the table builder cannot correctly determine all the places to generate eager reductions.

A solution to this problem is to augment an LR(0) item by adding a boolean flag that indicates whether or not the item comes from a context where reduction by the rule of the item creates an attachment. An *L∗(0) item* is written [S → NP VP, a], where a is a boolean flag that has the value T if the item comes from a context where reducing by the rule of the item will create an attachment, and F otherwise. For example, the rule S → NP VP has 6 possible L∗(0) items

[S → · NP VP, F]	[S → · NP VP, T]
[S → NP · VP, F]	[S → NP · VP, T]
[S → NP VP ·, F]	[S → NP VP ·, T]

Augmenting an item to include a boolean flag encoding attachment information can be applied to the entire family of LR(k) items.

3.4.1 Creating the L∗ NFA

The L∗ NFA is defined in a similar way to the LR NFA described in section 3.3. However, the states of the L∗ NFA are the L∗(0) items of a grammar.

For an augmented grammar $G' = (N, T, R \cup \{S' \to S\}, S')$, the L∗ NFA M recognising viable prefixes for G' is defined by the 5-tuple $M = (Q, N \cup T, \delta, q_0, Q)$, where the set of states Q is the set of L∗(0) items for G', the initial state q_0 is the item [S′ → · S, F], and the transition function δ is defined by the following rules. For notational convenience, β_h means that the head of the rule is contained within the non-empty sequence of symbols β. Also, a is a variable representing the boolean attachment flag of an item.

1. $\delta([\mathrm{X} \to \alpha\ \cdot L\,\beta, a],\ L) = \{[\mathrm{X} \to \alpha\ L \cdot\ \beta, a]\}$

2. $\delta([\mathrm{X} \to \alpha\ \cdot\ \mathrm{Y}\,\beta_h, a],\ \epsilon) = \{[\mathrm{Y} \to \cdot\ \gamma, \mathrm{F}] \mid \mathrm{Y} \to \gamma \in R\}$

3. $\delta([\mathrm{X} \to \alpha\ L \cdot\ \mathrm{Y}_h\,\beta, a],\ \epsilon) = \{[\mathrm{Y} \to \cdot\ \gamma, \mathrm{T}] \mid \mathrm{Y} \to \gamma \in R\}$

4. $\delta([\mathrm{X} \to \cdot\ \mathrm{Y}_h\,\beta, a],\ \epsilon) = \{[\mathrm{Y} \to \cdot\ \gamma, a] \mid \mathrm{Y} \to \gamma \in R\}$

5. $\delta([\mathrm{X} \to \alpha_h\ \cdot\ \mathrm{Y}\,\beta, a],\ \epsilon) = \{[\mathrm{Y} \to \cdot\ \gamma, \mathrm{T}] \mid \mathrm{Y} \to \gamma \in R\}$

Rule 1 applies in the situation where the parser is in a state when α in the rule X → $\alpha\ L\ \beta$ has already been recognised, and L has just been derived. Rules 2–5 apply when the parser is expecting to see Y next in the input, and Y derives γ, so the parser must also be expecting to see γ next in the input.

The setting of the attachment flag for each of the above rules is described in detail as follows:

1. Moving the dot through a rule does not change the context of where the item comes from, therefore the setting of the attachment flag remains unchanged.

2. The head of the grammar rule comes after Y, so parsing Y does not establish any new attachments to elements of the RHS of this rule. Hence parsing γ in order to derive Y causes no attachments, so the attachment flag of the items [Y $\rightarrow \cdot\ \gamma$, F] is not set.

3. The symbols in αL left-attach to the head Y, so parsing Y generates an attachment. Hence parsing γ to derive a Y causes an attachment, so the attachment flag of the items [Y $\rightarrow \cdot\ \gamma$, T] is set.

4. Although the head of the rule has been parsed, there are no symbols before Y in the rule X $\rightarrow$ Y_h β to left-attach to the head Y, so no new attachments are created by parsing Y. However, if reducing the rule X $\rightarrow$ Y_h β indirectly causes an attachment by another grammar rule, then parsing Y allows this attachment to be established. Thus the setting of the attachment flag for the items [Y $\rightarrow \cdot\ \gamma$, a] is the same as the setting for [X $\rightarrow \cdot$ Y_h β, a].

5. Parsing Y causes a right-attachment of Y to the head of the rule in α. It follows that parsing γ to derive a Y causes an attachment, so the attachment flag of the items [Y $\rightarrow \cdot\ \gamma$, T] is set.

For example, consider constructing the L$*$ NFA for grammar 3.2, annotated with syntactic heads as follows:

(1)	S	$\rightarrow$	NP VP_h	
(2)	VP	$\rightarrow$	V_h NP	Grammar 3.3
(3)	NP	$\rightarrow$	N_h	

Figure 3.3 shows the NFA resulting from applying rules 1–5 above. For the purposes of exposition, the rule of the transition function used to create each link in the NFA is shown in italics beside the link. For example, the arc from [S $\rightarrow \cdot$ NP VP, F] to [S $\rightarrow$ NP $\cdot$ VP, F] is created by rule 1, with X = S, α = ϵ, L = NP, β = VP. Similarly, the arc from [S $\rightarrow$ NP $\cdot$ VP, F] to [VP $\rightarrow \cdot$ V NP ,T] is created by rule 3, with X = S, α = ϵ, Y = VP, β = ϵ, and γ = V NP.

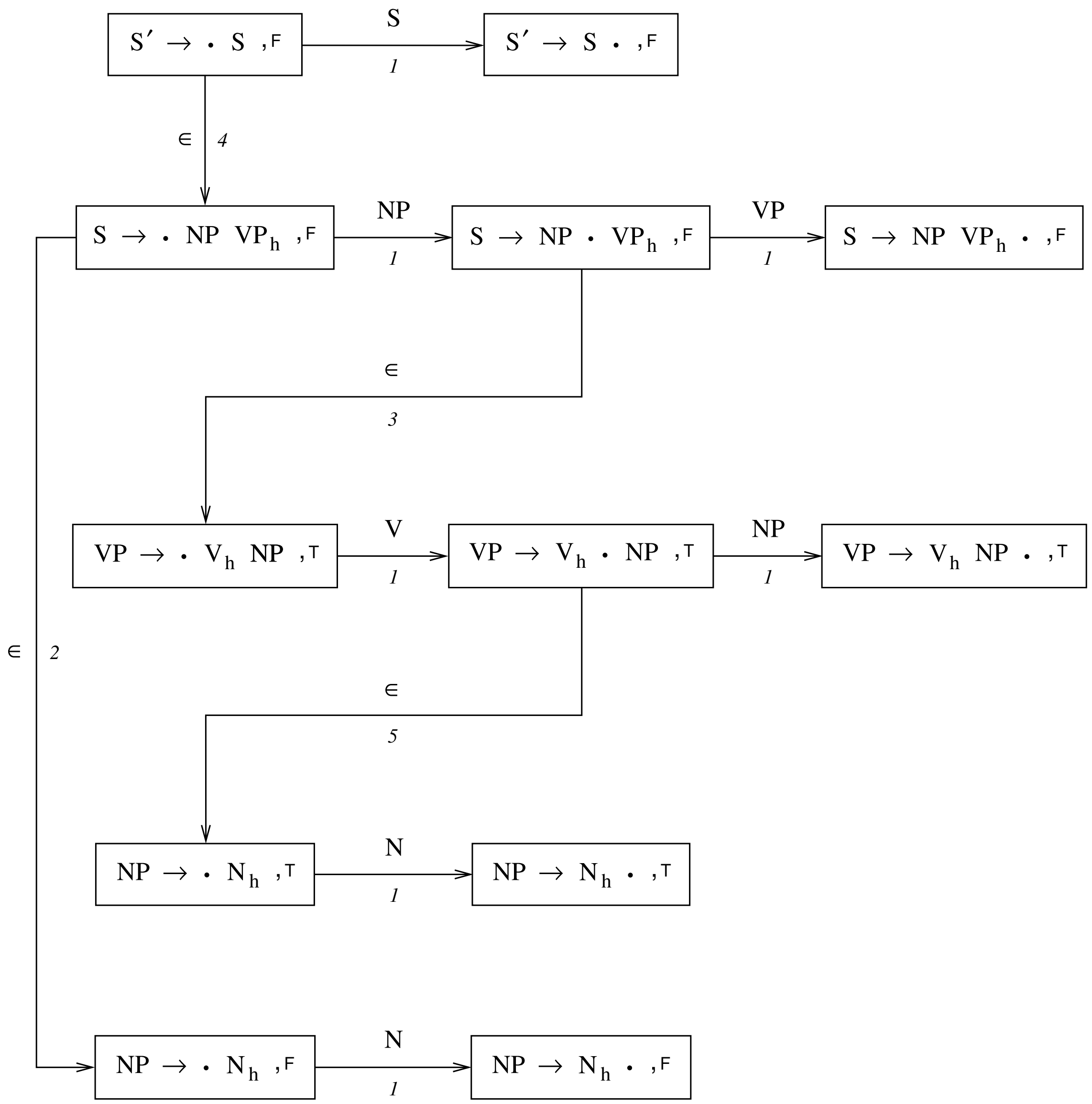


Figure 3.3: L$*$ NFA for recognising the viable prefixes of grammar 3.3.

Comparing this NFA with the LR NFA of figure 3.1, there is one major difference. In the LR NFA, the states for parsing an NP (namely [NP $\rightarrow \cdot$ N] and [NP $\rightarrow$ N $\cdot$]) were previously linked to both [S $\rightarrow \cdot$ NP VP] and [VP $\rightarrow$ V $\cdot$ NP]. In the L$*$ NFA, however, there are separate states for parsing the NP of [S $\rightarrow \cdot$ NP VP] and the NP of [VP $\rightarrow$ V $\cdot$ NP]. Parsing the NP of [S $\rightarrow \cdot$ NP VP] generates no attachment, so the attachment flag of the items [NP $\rightarrow \cdot$ N, F] and [NP $\rightarrow$ N $\cdot$, F] is not set. Parsing the NP of [VP $\rightarrow$ V $\cdot$ NP] generates a right-attachment of the

NP to the V already parsed, so the attachment flag of the items [NP $\rightarrow \cdot$ N, T] and [NP $\rightarrow$ N $\cdot$, T] is set. This exemplifies how the context-dependent nature of attachments affects the NFA.

The addition of the boolean attachment flag doubles the number of potential states in the NFA. However, not necessarily all items will be used in the NFA, because some rules may never be used both in a context where they would generate an attachment and a context where they would not.

3.4.2 Constructing an L$*$ DFA from an NFA

An L$*$ DFA can be constructed from an L$*$ NFA by subset construction, using the same method as for LR items. There is one special case in this construction. At some point in the parse, it is possible that a rule can be used both in a context where reducing it will generate an attachment, and a context where no attachment will result. For example, consider the following grammar:

(1)	S	$\rightarrow$	A X_h	
(2)	X	$\rightarrow$	Y_h	Grammar 3.4
(3)	X	$\rightarrow$	Y B_h	
(4)	Y	$\rightarrow$	Z_h C	

Parsing a Y generates an attachment in the context of rule 2, but not in the context of rule 3. This results in two different states for the rule Y $\rightarrow$ Z_h C in the NFA shown in figure 3.4. The arc from [X $\rightarrow \cdot$ Y_h, T] to [Y $\rightarrow \cdot$ Z_h C, T] is added by rule 4 for determining transitions, presented in section 3.4.1, whereas the arc from [X $\rightarrow \cdot$ Y B_h, T] to [Y $\rightarrow \cdot$ Z_h C, F] is added by rule 2. Although the two items have the same rule and position, the setting of the attachment flag is different.

This creates a complication when constructing the DFA from this NFA. Because the two items [Y $\rightarrow \cdot$ Z_h C, T] and [Y $\rightarrow \cdot$ Z_h C, F] can both be reached from the NFA state [S $\rightarrow$ A $\cdot$ X_h, F] by ϵ-transitions, both items will be included in the same state of the DFA by the subset construction algorithm. The problem is that the first item indicates that an eager reduction by rule 4 should be performed after parsing just Z because an attachment will result, whereas the second item indicates that the entire rule should be parsed before reducing by rule 4, because no attachment will result. Thus the same rule will be parsed in two different ways, when it should only be done once.

The DFA should contain a single item for any rule, so that the rule is processed only once. The attachment flag of this item should be set if there is *any* context where an eager reduction in

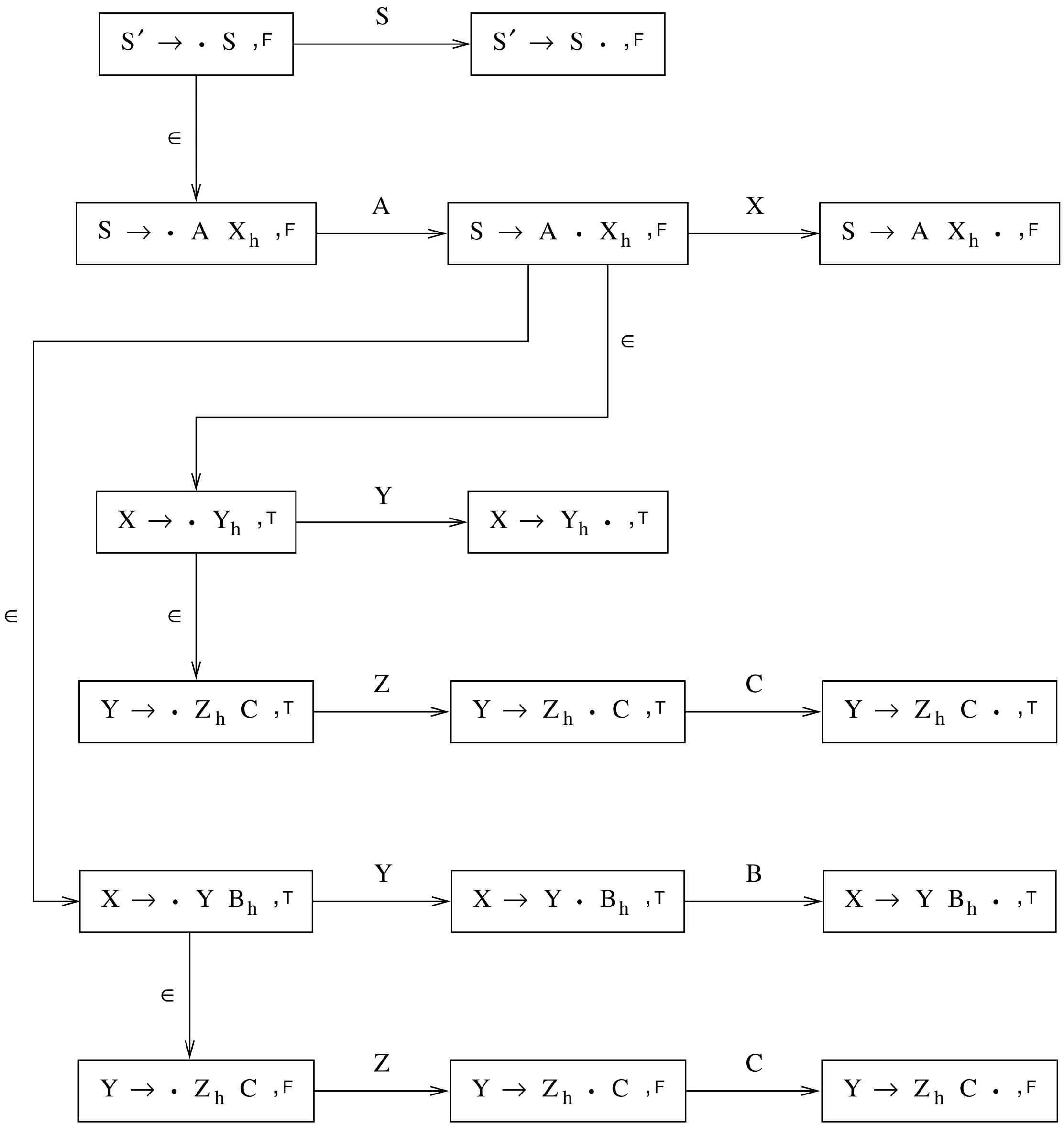


Figure 3.4: L$*$ NFA for recognising the viable prefixes of grammar 3.4.

this state would create an attachment. The item $[\mathrm{Y} \rightarrow \cdot\ \mathrm{Z}_h\ \mathrm{C}, \mathrm{T}]$ represents the fact that there is at least one context where the rule $\mathrm{Y} \rightarrow \mathrm{Z}_h\ \mathrm{C}$ will generate an attachment when reduced. Therefore, only the item $[\mathrm{Y} \rightarrow \cdot\ \mathrm{Z}_h\ \mathrm{C}, \mathrm{T}]$ is included in the state of the DFA.

Applying this principle during subset construction, the L$*$ DFA resulting from the NFA of figure 3.4 is shown in figure 3.5. In particular, note that state 2 contains only the item $[\mathrm{Y} \rightarrow \cdot\ \mathrm{Z}_h\ \mathrm{C}, \mathrm{T}]$ rather than both the items $[\mathrm{Y} \rightarrow \cdot\ \mathrm{Z}_h\ \mathrm{C}, \mathrm{T}]$ and $[\mathrm{Y} \rightarrow \cdot\ \mathrm{Z}_h\ \mathrm{C}, \mathrm{F}]$.

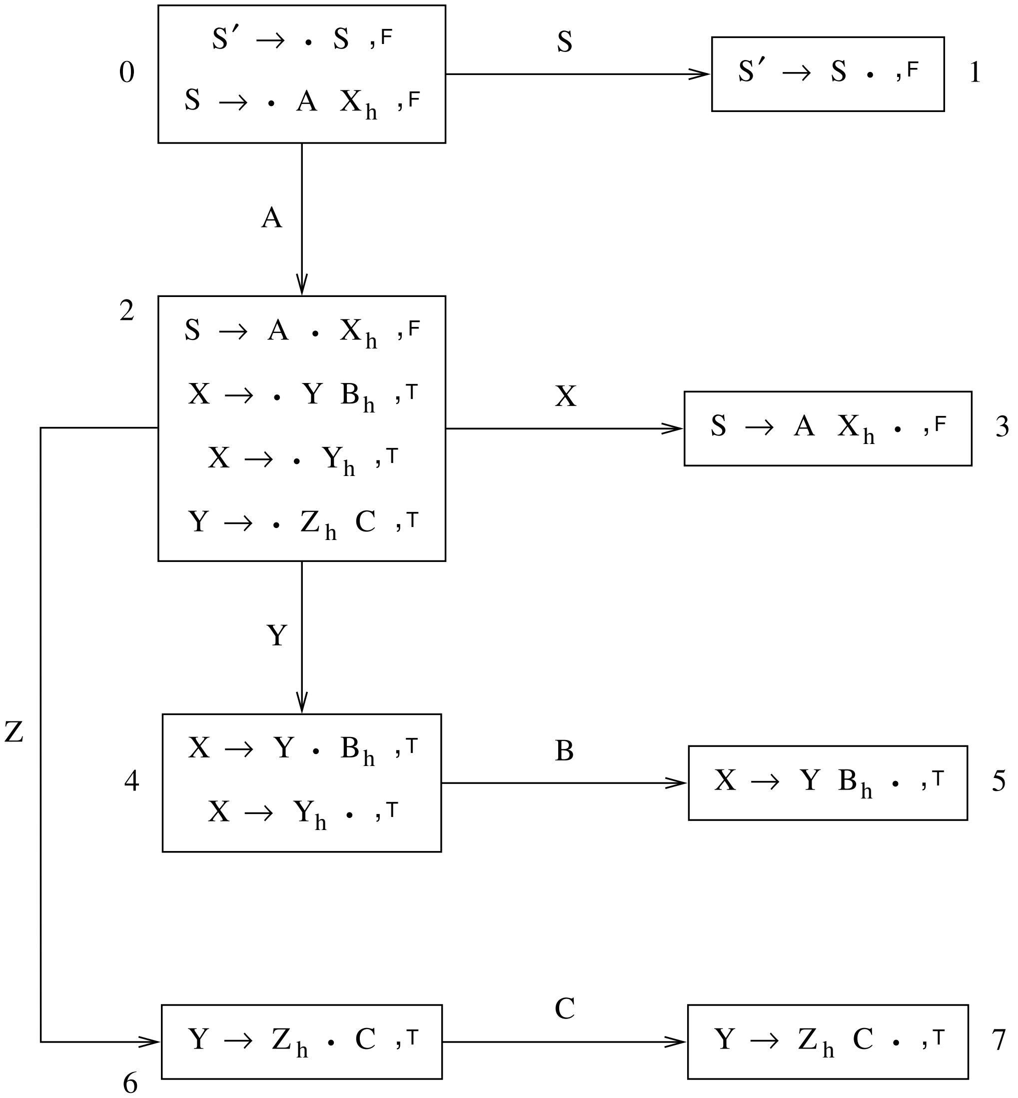


Figure 3.5: L∗ DFA for recognising the viable prefixes of grammar 3.4.

3.4.3 Constructing L∗ parse tables from finite automata

Once the DFA has been constructed, it must be converted into a parse table with actions specifying when to perform eager reductions. This is achieved by introducing a new `eager-reduce` parse action. An eager-reduce action specifies which grammar rule to reduce by and how many symbols of the right-hand side of the rule to use.

An eager reduction creates an incomplete derivation with some symbols of the RHS missing. As these missing symbols are parsed, the parser must incorporate them into the incomplete

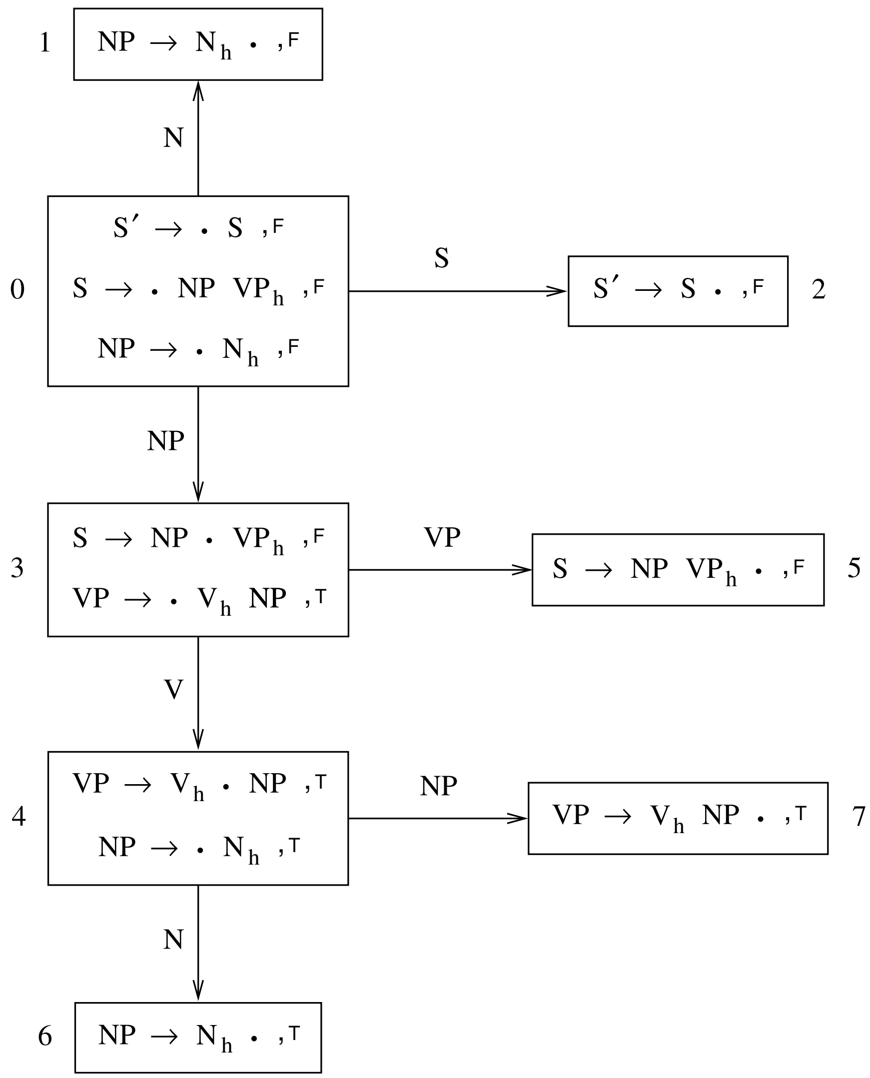


Figure 3.6: L$*$ DFA for recognising the viable prefixes of grammar 3.3.

derivation. This is specified using a new `combine` parse action. A combine action specifies that the symbol just parsed should be combined into an incomplete derivation created from a previous eager reduction by a particular grammar rule.

After performing an eager reduction, the parser carries out any further reductions triggered by the newly created nonterminal. A question immediately arises: exactly which reductions should be carried out? A GLR(k) parser would use lookahead to select only those reduce actions consistent with the k input symbols that immediately follow the substring covered by the reduction. Unfortunately, these lookahead symbols are not available when an eager reduction is performed, as the parse associated with the eagerly reduced rule is not yet complete. To replace the k symbol lookahead in this case, the dummy symbol *EAG* is introduced. The *EAG* column of the parse table contains the set of cascaded reductions to perform at a state *st* after an eager reduction. All entries in the *EAG* column are eager reductions because all the cascaded reductions have the incomplete originally eagerly reduced derivation as a descendant. Unlike other blank entries in the table (which signify a parse error), a blank in the *EAG* column means there are no cascaded reductions to be performed in this state.

To generate the actions for state I of a DFA:

1. Shifts, gotos, reduces, and accepts are generated as for a normal LR table.

2. For each item of the form $[\mathrm{X} \rightarrow \alpha_h \cdot \beta, a] \in I$

   where $\alpha_h = \gamma\ M_h \quad \wedge \quad a = \mathrm{T}$

   or $\alpha_h = \gamma\ L\ M_h \quad \wedge \quad a = \mathrm{F}$

   or $\alpha_h = L_h\ M \quad \wedge \quad a = \mathrm{F}$

   - If β is not empty, add the action `eager-reduce` *n-k* to ACTION[I, x] for every $x \in$ FIRST(β), where n = X $\rightarrow \alpha_h\ \beta$ and k = $|\alpha|$.
   - If M is a nonterminal, add the action `eager-reduce` *n-k* to ACTION[I, *EAG*], where n = X $\rightarrow \alpha_h\ \beta$ and k = $|\alpha|$.

3. For each item of the form $[\mathrm{X} \rightarrow \alpha_h \cdot K\ \beta, a]$

   where $\alpha_h = \gamma_h \quad \wedge \quad a = \mathrm{T}$

   or $\alpha_h = \gamma\ L\ \delta_h \quad \wedge \quad a = \mathrm{F}$

   or $\alpha_h = L_h\ M\ \gamma \quad \wedge \quad a = \mathrm{F}$

   - Add the action `combine` n to ACTION[I, K], where n = X $\rightarrow \alpha_h\ K\ \beta$.

Rule 1 adds the LR parsing actions to the parse table. Rule 2 adds eager-reduce actions. Rule 3 adds combine actions. Looking at rules 2 and 3 in detail:

2(a) If $\alpha_h = \gamma\ M_h \wedge a = \mathrm{T}$, then the item $[\mathrm{X} \rightarrow \alpha_h \cdot \beta, a]$ comes from a context where reducing by the rule of the item will cause an attachment (because the attachment flag is true), so once the head of the rule is parsed, the rule should be eagerly reduced. The dot is immediately after the head, indicating the head must have just been parsed. Thus an eager reduction should be generated if the end of the rule has not already been reached (β is not empty). If the head is a nonterminal, it may have been created by eager reduction, so an eager reduction for the *EAG* symbol should be generated.

2(b) If $\alpha_h = \gamma\ L\ M_h\ \wedge\ a = \text{F}$, then the item $[\text{X} \rightarrow \alpha_h\ \cdot\ \beta, a]$ does not come from a context where reducing by the rule of the item will cause an attachment (the attachment flag is false), but the symbols in γL left-attach to the head of the rule. The dot is immediately after the head, indicating the head must have just been parsed. Thus an eager reduction should be generated if the end of the rule has not already been reached (β is not empty). If the head is a nonterminal, it may have been created by eager reduction, so an eager reduction for the *EAG* symbol should be generated.

2(c) If $\alpha_h = L_h\ M\ \wedge\ a = \text{F}$, then the item $[\text{X} \rightarrow \alpha_h\ \cdot\ \beta, a]$ does not come from a context where reducing by the rule of the item will cause an attachment (the attachment flag is false), but M right-attaches to the head of the rule. M is the first symbol that right-attaches to the head and the dot is immediately after M. Thus an eager reduction should be generated if the end of the rule has not already been reached (β is not empty). If M is a nonterminal, it may have been created by an eager reduction, so an eager reduction for the *EAG* symbol should be generated.

3(a) If $\alpha_h = \gamma_h\ \wedge\ a = \text{T}$, then the item $[\text{X} \rightarrow \alpha_h\ \cdot K\ \beta, a]$ comes from a context where reducing by the rule of the item will cause an attachment, and the head of the rule has already been parsed, so the rule must have been eagerly reduced. Thus a combine action should be generated for each of the remaining symbols of the rule. The next symbol to be parsed is K, so a combine action by the rule $\text{X} \rightarrow \alpha_h\ K\ \beta$ for K is generated.

3(b) If $\alpha_h = \gamma\ L\ \delta_h\ \wedge\ a = \text{F}$, then the head of the rule of the item $[\text{X} \rightarrow \alpha_h\ \cdot K\ \beta, a]$ has been parsed, along with the symbols in γL which left-attach to the head, so the rule must have been eagerly reduced. Thus a combine action should be generated for the remaining symbols of the rule. The next symbol to be parsed is K, so a combine action by the rule $\text{X} \rightarrow \alpha_h\ K\ \beta$ for K is generated.

3(c) If $\alpha_h = L_h\ M\ \gamma\ \wedge\ a = \text{F}$, then the head of the rule of the item $[\text{X} \rightarrow \alpha_h\ \cdot K\ \beta, a]$ has been parsed, along with the symbols in $M\gamma$ which right-attach to this head, so the rule must have been eagerly reduced. Thus a combine action should be generated for the remaining symbols of the rule. The next symbol to be parsed is K, so a combine action by the rule X $\rightarrow \alpha_h\ K\ \beta$ for K is generated.

The L$*$ parse table resulting from applying these rules to the DFA in figure 3.6 is shown in table 3.2. The table is indexed by state number *st* and grammar symbol *L*. An entry ACTION[*st*, *L*] is a set of parse actions. A blank entry represents a parse error, unless it is the *EAG* entry, in which case a blank represents the fact that there are no cascaded reductions appropriate in this state.

STATE	ACTION						
	N	V	$	*EAG*	VP	NP	S
0	s1					g3	g2
1		r3	r3				
2			acc				
3		s4			g5		
4	s6, e2-1					g7, c2	
5			r1	e1-2			
6		r3	r3				
7			r2	e2-2			

Table 3.2: L$*$(1) parse table for grammar 3.3.

Actions are written as follows:

- "`s` *n*" means shift and go to state *n*.
- "`r` *n*" means reduce by the *n*-th grammar rule.
- "`g` *n*," means go to state *n*.
- "`e` *n*-*k*" means eagerly reduce by the *n*-th grammar rule using only the first *k* symbols of the RHS.
- "`c` *n*" means combine the grammar symbol *L* into an incomplete derivation created by an eager reduction by grammar rule *n*.
- "`acc`" means accept.

To illustrate the process of table construction, consider constructing the table row for state 4 of the DFA. The actions `s6` and `g7` are added by the rules for adding actions to LR parse tables. Rule 2 for generating actions can be applied to the item [VP $\rightarrow$ V$_h$ $\cdot$ NP, T], with X = VP, α = V, and β = NP. As $\beta \neq \epsilon$, the action `e2-1` is added to ACTION[4, N], because VP $\rightarrow$ V$_h$ NP is grammar rule 2, |V| = 1, and FIRST(NP) = {N}. V is not a nonterminal, so no entry is added to *EAG*. Rule 3 can also be applied to the item [VP $\rightarrow$ V$_h$ $\cdot$ NP, T], with X = VP, α = V, K = NP, and β = ϵ. The action `c2` is added to ACTION[4, NP], because K = NP, and VP $\rightarrow$ V$_h$ NP is grammar rule 2.

3.5 Left recursion

Grammars containing left-recursion can create problems with eager reduction. For example, consider the grammar rule NP $\rightarrow$ NP$_h$ PP. If the parser eagerly reduces this rule after seeing the head, it will create an NP to which this rule can again be applied, creating a third NP, and so on ad infinitum. More generally, define a relation ϕ such that $M\ \phi\ L$ if and only if $L \rightarrow M_h\ \beta$. Then a grammar is *head-left-recursive* if $L\ \phi^+\ L$. The parser will loop, infinitely proposing eager reductions, for any head-left-recursive grammar. For example, consider the following grammar:

(1)	S	$\rightarrow$	A X$_h$	
(2)	X	$\rightarrow$	Y$_h$ C	
(3)	Y	$\rightarrow$	Z	
(4)	Z	$\rightarrow$	X$_h$ W F	Grammar 3.5
(5)	Z	$\rightarrow$	B	
(6)	W	$\rightarrow$	D E$_h$	

When processing the sentence "A B C D E F", the parser will reduce B to Z by eager reduction using rule 5. It can then create a Y by a reduction using rule 3, which can then be eagerly reduced to ax X by rule 2, which can in turn be eagerly reduced to an Z by rule 4, and so on ad infinitum.

To avoid infinite recursion, an L$*$ parser must accommodate some method of dealing with head-left-recursive loops in a grammar. The standard method of dealing with the similar problem of left-recursion in top-down parsing is to rewrite the grammar (Hopcroft and Ullman, 1979). However this is an undesirable approach for L$*$ parsing because, in natural language processing, grammar structure has more significance than simply a way of recognising the sentences of a language.

A preferable approach is to delay performing eager reductions that would otherwise cause the parser to enter a loop. For example, in grammar 3.5, if rule 4 is not eagerly reduced until after W is processed, then the parser does not enter a loop. This is because, to create a W, the input symbols D and E must be processed, so the loop is only processed as many times as justified by the input, rather than infinitely. Loops can be detected during parsing, however, delaying processing eager reductions to avoid these loops introduces significant complexity and overhead.

A simpler method of avoiding loops is to construct the parse table so that eager reductions do not cause head-left-recursive loops. When constructing the parse table, a minimal set of rules which should not be eagerly reduced immediately after their head should be identified. Eager reductions by these rules must only be performed after input is processed. This avoids sending the parser into an infinite loop.

Identifying the rules in which eagerly reduction should be delayed is equivalent to finding cycles in a graph constructed from the grammar. The vertices of the graph are the terminals and nonterminals of the grammar. There is a directed edge from L to M if $M \ \phi \ L$. Cycles in this graph can be detected using a simple graph search algorithm such as depth-first search. The outline of an algorithm for identifying the rules in which eager reduction should be delayed is given in section 3.6.2.

Having identified the rules in which eager reduction should be delayed, the parse table should be constructed so that eager reductions are not generated immediately after the head of these rules. Instead, an eager reduction should be generated after the next non-empty symbol of the rule is processed. For example, when building the parse table for grammar 3.5 from the DFA shown in figure 3.7, rule 4 is identified as the only rule in which eager reduction should be delayed. Instead of generating an eager reduction by rule 4 in state 3, this eager reduction is delayed until the next non-null symbol of the rule has been parsed. As W must be non-null, rule 4 should be eagerly reduced after W is parsed. Thus, an eager reduction by rule 4 is generated in state 8 instead. The resulting parse table is shown in table 3.3.

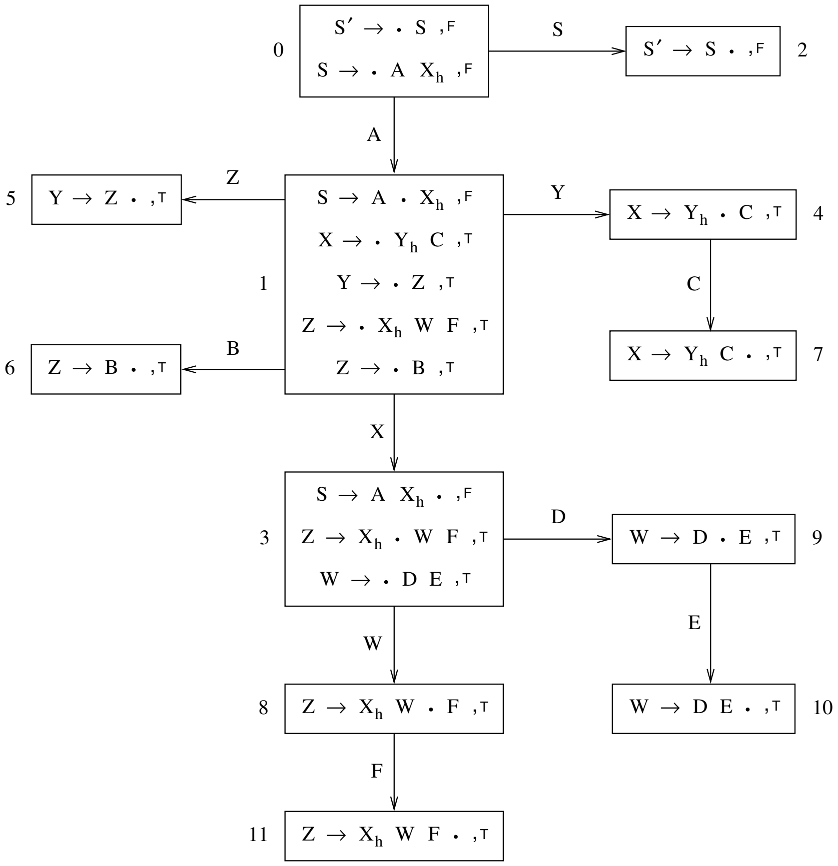


Figure 3.7: L∗ DFA for grammar 3.5.

STATE	ACTION												
	A	B	C	D	E	F	$	*EAG*	S	X	Y	Z	W
0	s1								g2				
1		s6								g3	g4	g5	
2							acc						
3				s9			r1	e1-2					g8
4			e2-1					e2-1					
			s7, c2										
5			r3					e3-1					
6			r5										
7				r2			r2						
8						e4-2		e4-2					
						s11, c4							
9					s10								
10						r6							
11			r4										

Table 3.3: L∗ parse table for grammar 3.5.

The rules for constructing the L$*$ parse table, avoiding left recursion, are as follows. To generate the actions for state I of a DFA:

1. Shifts, gotos, reduces, and accepts are generated as for a normal LR table.

2. For each item of the form [X $\rightarrow \alpha_h \cdot \beta$, a] $\in I$

   where $\alpha_h = M_h \quad \wedge \quad a = \mathtt{T}$ and not delaying eager reduction by this rule

   or $\alpha_h = \gamma\ L\ M_h$

   or $\alpha_h = L_h\ M \quad \wedge \quad a = \mathtt{T}$ and delaying eager reduction by this rule

   or $\alpha_h = L_h\ M \quad \wedge \quad a = \mathtt{F}$

   - If β is not empty, add the action `eager-reduce` *n-k* to ACTION[I, x] for every $x \in$ FIRST(β), where n = X $\rightarrow \alpha_h\ \beta$ and k = $|\alpha|$.
   - If M is a nonterminal, add the action `eager-reduce` *n-k* to ACTION[I, *EAG*], where n = X $\rightarrow \alpha_h\ \beta$ and k = $|\alpha|$.

3. For each item of the form [X $\rightarrow \alpha_h \cdot K\ \beta$, a]

   where $\alpha_h = L_h\ \gamma \quad \wedge \quad a = \mathtt{T}$ and not delaying eager reduction by this rule

   or $\alpha_h = \gamma\ L\ \delta_h$

   or $\alpha_h = L_h\ M\ \gamma \quad \wedge \quad a = \mathtt{T}$ and delaying eager reduction by this rule

   or $\alpha_h = L_h\ M\ \gamma \quad \wedge \quad a = \mathtt{F}$

   - Add the action `combine` n to ACTION[I, K], where n = X $\rightarrow \alpha_h\ K\ \beta$.

3.6 Formal table-building algorithms

3.6.1 L$*$ Table Building Algorithm

Input

An augmented context-free grammar $G' = \langle N, T, R, S\rangle$. N is a set of nonterminals, T is a set of terminals, R is a set of grammar rules of the form $X \rightarrow \alpha$, where $X \in N$ and $\alpha \in (N \cup T)^*$, and S is the start symbol.

Output

An L$*$ parse table ACTION[*state*, *symbol*] for the grammar G'.

Main Program

· Call **Table**(**ItemSets**(G'))

ItemSets($\mathcal{G}$)

Compute the set of all item sets of a grammar. The sets of item sets are the states of a DFA recognising viable prefixes of the grammar.

· REPEAT
  · $\mathcal{C} \leftarrow \{$**Closure**$([S' \rightarrow \cdot S,\ \text{F}])\}$
  · $\forall \mathcal{I} \in \mathcal{C}$
    · $\forall L$ s.t. **Goto**$(I, L) \neq \{\}$
      · $\mathcal{C} \leftarrow \mathcal{C} \cup$ **Goto**(I, L)
· UNTIL no more item-sets can be added to $\mathcal{C}$
· RETURN $\mathcal{C}$

Closure($\mathcal{I}$)

Compute all the grammar rules that may possibly apply at a point in the parse

· REPEAT
  · $\forall [X \rightarrow \alpha\ \cdot\ Y\beta,\ f] \in \mathcal{I}$
    · $\forall\,(Y \rightarrow \gamma) \in G$
      · If $\alpha \neq \epsilon$
        · f' = **KFlag**($[X \rightarrow \alpha\ \cdot\ Y\beta,\ f]$)
      Else
        · f' = **CFlag**($[X \rightarrow \alpha\ \cdot\ Y\beta,\ f]$)
      · If $\exists [Y \rightarrow \cdot\gamma,\ f''] \in \mathcal{I}$
        · If $f' = \text{T}$ then
          · $f'' \leftarrow \text{T}$
      Else
        · $\mathcal{I} \leftarrow \mathcal{I} \cup [Y \rightarrow \cdot\gamma,\ f']$
· UNTIL no more items can be added to $\mathcal{I}$
· RETURN $\mathcal{I}$

Goto$(\mathcal{I}, L)$

Calculate the item set resulting from a transition by symbol L from item set $\mathcal{I}$

- $\mathcal{J} \leftarrow \{\}$
- $\forall[X \rightarrow \alpha \cdot L\beta,\ f] \in \mathcal{I}$
  - $\mathcal{J} \leftarrow \mathcal{J} \cup [X \rightarrow \alpha L \cdot \beta,\ f]$
- RETURN **Closure**(J)

KFlag$([X \rightarrow \alpha \cdot Y\beta,\ f])$

Calculate the flags for the item resulting from the closure of a kernel item (an item where α is not empty).

- If the head of $X \rightarrow \alpha Y \beta$ is in αY or β is empty
  - RETURN T

  Else
  - RETURN F

CFlag$([X \rightarrow \cdot Y\beta,\ f])$

Calculate the flags for the item resulting from the closure of a complement item (an item where the dot is at the start of the rule).

- If Y is the head of $X \rightarrow Y\beta$ or β is empty
  - RETURN f

  Else
  - RETURN F

Table$(\mathcal{C})$

Calculate the parse table from the canonical set of items.

- $\forall \mathcal{I}_j \in \mathcal{C}$
  - If $[S' \rightarrow S \cdot,\ f] \in \mathcal{I}_j$
    - Add "`accept`" to ACTION[j, \$]
  - If $[X \rightarrow \alpha \cdot,\ f] \in \mathcal{I}_j$
    - $\forall x \in$ Follow(X)
      - Add "`reduce` $X \rightarrow \alpha$" to ACTION[j, x]
  - If $[X \rightarrow \alpha \cdot A\beta,\ f] \in \mathcal{I}_j \wedge$ **Goto**$(\mathcal{I}_j, A) = \mathcal{I}_k$
    - Add "`shift` k" to ACTION[j, A]
  - If $[X \rightarrow \alpha \cdot X\beta,\ f] \in \mathcal{I}_j \wedge$ **Goto**$(\mathcal{I}_j, X) = \mathcal{I}_k$
    - Add "`goto` k" to ACTION[j, X]
  - If $[X \rightarrow \alpha_h \cdot \beta,\ a] \in \mathcal{I}_j$ and $(\alpha_h = \gamma M_h \wedge a = \text{T}) \vee (\alpha_h = \gamma L M_h \wedge a = \text{F}) \vee (\alpha_h = L_h M \wedge a = \text{F})$
    - If β is not empty
      - $\forall x \in$ First(β)
        - Add "`eager-reduce` $X \rightarrow \alpha\beta,\ |\alpha|$" to ACTION[$j$, x]
    - If M is a nonterminal
      - Add "`eager-reduce` $X \rightarrow \alpha\beta,\ |\alpha|$" to ACTION[$j$, *EAG*]
  - If $[X \rightarrow \alpha_h \cdot K\beta,\ a] \in \mathcal{I}_j$ and $(\alpha_h = \gamma_h \wedge a = \text{T}) \vee (\alpha_h = \gamma L \delta_h \wedge a = \text{F}) \vee (\alpha_h = L_h M \gamma \wedge a = \text{F})$
    - Add "`combine` $X \rightarrow \alpha K \beta$" to ACTION[$j$, K]

□

3.6.2 Algorithm for detecting head-left-recursive loops

This section presents the outline of an algorithm for finding a minimal set of grammar rules in which to delay eager reduction to avoid left recursion.

· Build a graph where the vertices of the graph are the terminals and nonterminals of the grammar. Add a directed edge from L to M if $M \phi L$.
· Build a DFS spanning tree of the graph. Whenever a cycle is discovered, increment a cycle counter at each vertex in the cycle.
· Create a set of all vertices with cycle count > 0.
· Repeat until this set is empty (thus no cycles remain)
  · Remove a vertex with the maximum cycle count, and add it to a list of rules in which to delay eager reduction.
  · For each cycle of which thus vertex is a member, decrease the cycle count by one on all members of the cycle.
  · Remove all vertices with a cycle count of 0 from the set.
· Return the list constructed by this loop.

Chapter 4

L$*$ Parsing

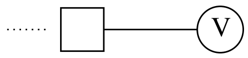
L$*$ parsing is a generalisation of bottom-up parsing that allows a grammar rule to be reduced before all members of the RHS have been derived from the input. The actions to perform during parsing are stored in a pre-compiled parse table, as described in chapter 3. The L$*$ parse table includes two new parse actions not present in a standard GLR parse table—the *eager-reduce* and *combine* actions. During parsing, the driver that executes the actions specified in the parse table must execute these new eager-reduce and combine actions.

This chapter describes the basic method used to execute the two new actions. Section 4.1 examines the eager-reduce action, and section 4.2 examines the combine action. Section 4.3 then presents a simple example to illustrate parsing with the two new actions, and section 4.4 presents a formal specification of the basic L$*$ algorithm without packing. Chapter 5 then presents improvements to the basic L$*$ algorithm that are needed to make it an efficient algorithm for practical parsing. Chapter 6 extends the algorithm to perform local ambiguity packing.

4.1 The eager-reduce action

As explained in section 3.2, an eager reduction is a reduction by a grammar rule performed before all members of the RHS of the rule have been parsed. An eager reduction is specified by an eager-reduce action `eager-reduce` *n-k*, where *n* is the grammar rule to reduce by, and *k* is the number of symbols to use in the reduction. Eager-reduce actions are stored in the parse table.

Performing an eager reduction creates an *incomplete derivation* in the parse forest. The derivation is incomplete because not all members of the RHS have been parsed when the derivation is created. For example, if the graph-structured stack includes the following stack top



and the parse table specifies an eager reduction by the rule VP $\rightarrow$ V_h NP after parsing V, then the parser creates a new branch of the stack for the result of the eager reduction and a new incomplete forest node for VP, as shown in figure 4.1.

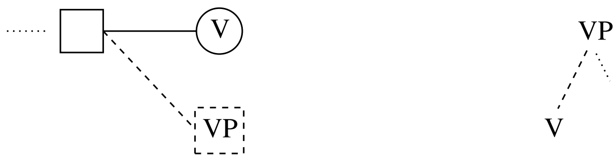


Figure 4.1: Parse state after an eager reduction.

The length of the reduction path in the stack is one, rather than two as it would be for a full reduction, because the eager reduction is performed after only seeing one symbol (V) of the RHS. Also, the forest node VP resulting from the eager reduction is incomplete: it is missing a child NP, because this NP has not been parsed at the time that the eager reduction is performed.

4.1.1 Cascaded reductions

To reap the benefits of an eager reduction, the parser should perform any further reductions triggered by the nonterminal that resulted from the eager reduction. These are called *cascaded reductions*. For example, with grammar 4.1, the parser will eagerly reduce by rule 2 after parsing A to allow the left-attachment of X to the newly created Y by rule 1. To establish this attachment, however, requires a cascaded reduction by rule 1.

(1) S $\rightarrow$ X Y_h
(2) Y $\rightarrow$ A_h B

Grammar 4.1

How should the parser access the parse table to determine which cascaded reductions to perform? Normally, the parser uses lookahead to access the table in order to perform only reductions consistent with the immediately following symbols. However, these lookahead symbols are unavailable immediately after an eager reduction is performed, because the parse associated with the eagerly reduced rule is not yet complete. To solve this problem, the dummy symbol *EAG* was introduced in chapter 3. The *EAG* column of the parse table contains the set of cascaded reductions to perform at a state after an eager reduction. To perform a cascaded reduction, the parser proceeds as usual, except it uses the *EAG* symbol rather than lookahead symbols to access the parse table.

4.2 The combine action

An eager reduction creates an incomplete derivation. The eager reduction is performed on the assumption that the missing symbols of this incomplete derivation will be subsequently derived from the input. As each missing symbol is derived, it needs to be incorporated into the incomplete derivation. In figure 4.1, the VP is eagerly reduced without the parser having seen the NP. When the NP is derived from the input, it must be added to the derivation of the VP.

A combine action of the form `combine` n specifies that the symbol just parsed should be combined into an incomplete derivation created from an eager reduction by rule n. The combine action does not encode which derivation should be combined into, because the appropriate derivation is created only during parsing.

4.2.1 Combine pointers

The L$*$ parser uses a data structure called a *combine pointer* to locate an incomplete derivation, so that missing children can be combined into the derivation as they are parsed. There are three parts to a combine pointer: an incomplete derivation, the grammar rule that was used to create the incomplete derivation, and the vertices in the stack that resulted from the eager reduction that first created the incomplete derivation. A combine pointer is written $[n \to d : \mathcal{V}]$, where n is the grammar rule, d is the incomplete derivation, and $\mathcal{V}$ is the list of vertices in the stack. Combine pointers are stored at stack vertices, and are created by eager reductions. As part of performing a combine action, the combine pointers used in the action are moved to the vertex associated with the symbol being combined into the incomplete derivation. Thus combine pointers are always stored at the vertex in the stack whose associated forest node is the rightmost child of the incomplete derivation pointed to by the combine pointer.

When performing a combine action of the form `combine` n at a given vertex, the parser locates the derivations to which the combine action should be applied by using the combine pointers of the form $[n \to d : \mathcal{V}]$ at that vertex.

If, there is a combine pointer of the form $[n \rightarrow d : \mathcal{V}]$ at a given vertex, the parser does not perform any eager reductions by rule n at that vertex, because the combine pointer indicates that such eager reductions have already been previously performed.

4.2.2 Completing reductions

After all missing elements of an incomplete derivation have been parsed and combined into the derivation, the parser marks the derivation as complete. This is accomplished by performing a *completing reduce*. Unlike other reductions, a completing reduction creates no new stack or forest structure—all appropriate structure has already been created by an eager reduction and subsequent combines. All that a completing reduction does is mark the appropriate derivation as complete.

A completing reduction appears as a reduce action in the parse table, and is identified as a completing reduction by the presence of appropriate combine pointers at the vertex where the reduction is scheduled. The combine pointers point to derivations already created from eager reduction by the grammar rule now scheduled for full reduction. The fact that a full reduction has been scheduled (as opposed to an eager reduction) indicates that the end of the rule has been reached, so the incomplete derivation previously created using the rule must be complete. The combine pointers also store a list of vertices that resulted from the original eager reduction. These are the vertices that the parser should establish as the new stack tops resulting from the completing reduce.

4.3 A simple example of parsing using the L$*$ algorithm

As an example of the L$*$ algorithm, consider the problem of parsing the simple sentence "John saw Mary" using the grammar 4.2. This grammar is the same as grammar 3.3 in chapter 3, and this example traces parsing with table 3.2 constructed in that chapter.

(1)	S	$\rightarrow$	NP VP$_h$	
(2)	VP	$\rightarrow$	V$_h$ NP	Grammar 4.2
(3)	NP	$\rightarrow$	N$_h$	

A thumbnail sketch of the parser's actions while parsing the sentence "John saw Mary" is as follows. First "John" is parsed as an NP as in ordinary GLR parsing. Next, the word "saw" is pushed onto the stack. The parser then eagerly reduces by rule 2, and pushes the resulting incomplete VP onto the stack. This immediately triggers a further eager reduction by rule 1,

STATE	ACTION						
	N	V	$	*EAG*	VP	NP	S
0	s1					g3	g2
1		r3	r3				
2			acc				
3		s4			g5		
4	s6, e2-1					g7, c2	
5			r1	e1-2			
6		r3	r3				
7			r2	e2-2			

Table 4.1: L$*$(1) parse table for grammar 4.2.

creating an incomplete S. This S is incomplete even though parse nodes for each of its immediate children exist, because its child VP node is missing a child NP. The system next parses "Mary" as an NP and combines it into the incomplete VP. A completing reduction by rule 2 then marks the VP as complete, because all its children have now been parsed. Finally, a completing reduction by rule 1 marks the S as complete, because its child VP is now complete.

To illustrate the L$*$ algorithm in detail, the state of the parser at each step is shown in a diagram with three components:

- *The graph-structured stack.*

  The graph-structured stack is drawn in the same way as the diagrams of chapter 2, with the addition that vertices are now drawn with their combine pointers below them. Vertices created by eager reduction are drawn with dashed lines.

- *The parse forest.*

  The parse forest is drawn in the same was as the diagrams of chapter 2, with the addition that dashed lines joining forest nodes represent links created by eager rather than normal reduction, and a dotted line indicates the forward edge of an incomplete derivation created by eager reduction. As parsing proceeds, combine actions incrementally extend the derivation. When all RHS elements are in place, a completing reduction marks the forest node as complete, depicted by changing the dashed lines to solid lines.

- *The current word.*

  As in chapter 2, the word the parser is currently processing is shown in a box at the right-hand edge of the diagram.

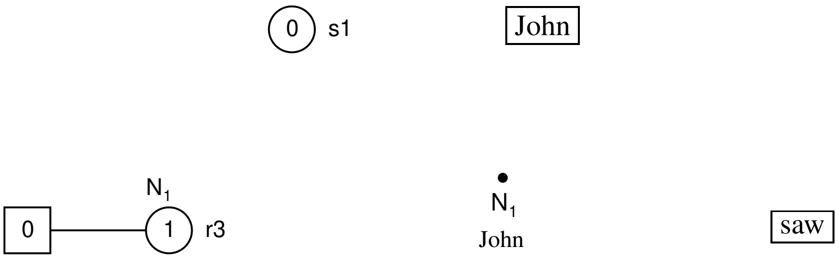


Figure 4.2: Trace of the L∗ algorithm parsing “John saw Mary”.

Table 4.1 shows the L∗ parse table (from table 3.2) for grammar 4.2. The table is indexed by state number *st* and grammar symbol *L*. A blank entry represents a parse error, unless it is the *EAG* entry, in which case a blank entry represents the fact that there are no cascaded reductions appropriate in this state. An entry ACTION[*st*, *L*] is a set of parse actions, as described in section 3.4.3.

The remainder of this section presents a detailed trace of the L∗ algorithm parsing the sentence “John saw Mary”. The parser is initialised with a single vertex with state 0, shown in the first diagram of figure 4.2. Initially the parse proceeds as with GLR parsing, with “John” being pushed onto the stack (second diagram of figure 4.2), and reduced to NP_1 by rule 3 (first diagram of figure 4.3). Next “saw” is shifted onto the stack, leaving the parser in the state shown in the second diagram of figure 4.3.

The next word to process is “Mary,” with lexical category N. The parse actions to perform are given by ACTION[4, N] = {`e2-1`, `s6`}. Reductions are processed before shifts as in GLR parsing. Thus the parser next eagerly reduces by rule 2 using only one symbol in the reduction. Performing this reduction creates a new forest node VP_1 and new stack vertex whose state is determined using the parse table entry ACTION[3, VP] = {`g5`}. The parse tree for VP_1 is drawn using dashed and dotted lines to indicate that the parse of the VP is incomplete, and the stack vertex is drawn with dashed lines indicating it was created by eager reduction. The eager reduction also establishes a new combine pointer at the vertex with state 4. This combine pointer indicates that the incomplete VP_1 was created by an eager reduction by rule 2, and indicates that the reduction created the new vertex with state 5 in the stack (third diagram of figure 4.3).

Having just performed an eager reduction, the parser now performs any *cascaded* reductions

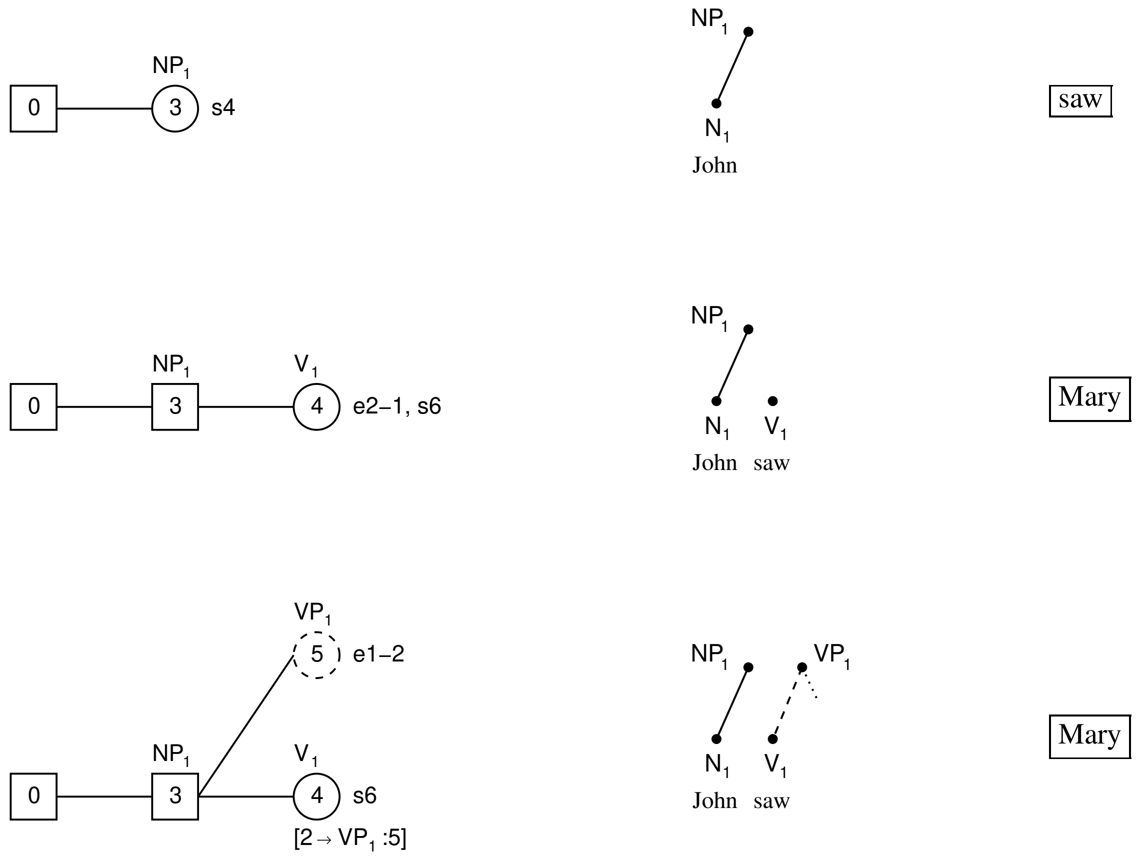


Figure 4.3: Trace of the L$*$ algorithm parsing "John saw Mary" (cont.).

stored in the *EAG* column of the parse table. Because ACTION[5, *EAG*] = {`e1-2`}, the parser carries out a cascaded eager reduction by rule 1 and creates the new forest node S_1 from NP_1 and VP_1, and a new stack vertex with state determined from ACTION[0, S] = {`g2`}. The entry ACTION[2, *EAG*] is blank, so the parser does nothing further at this vertex (first diagram of figure 4.4).

No unprocessed reductions remain for the current word ("Mary"), so the parser finally performs the outstanding shift action `s6` at the vertex with state 4. The word "Mary" is shifted onto the stack, making the sentinel $ the current word (second diagram of figure 4.4). The parser then reduces N_2 at the top of the stack to NP_2 by rule 3. The table entry ACTION[4, NP] = {`g7`, `c2`} is used to determine the state of the resulting vertex. The goto action specifies that the state of the new vertex is state 7. A combine action by rule 2 must also be performed. The combine pointer at the vertex with state 4 identifies VP_1 as the node created by eager reduction by rule 2. The parser

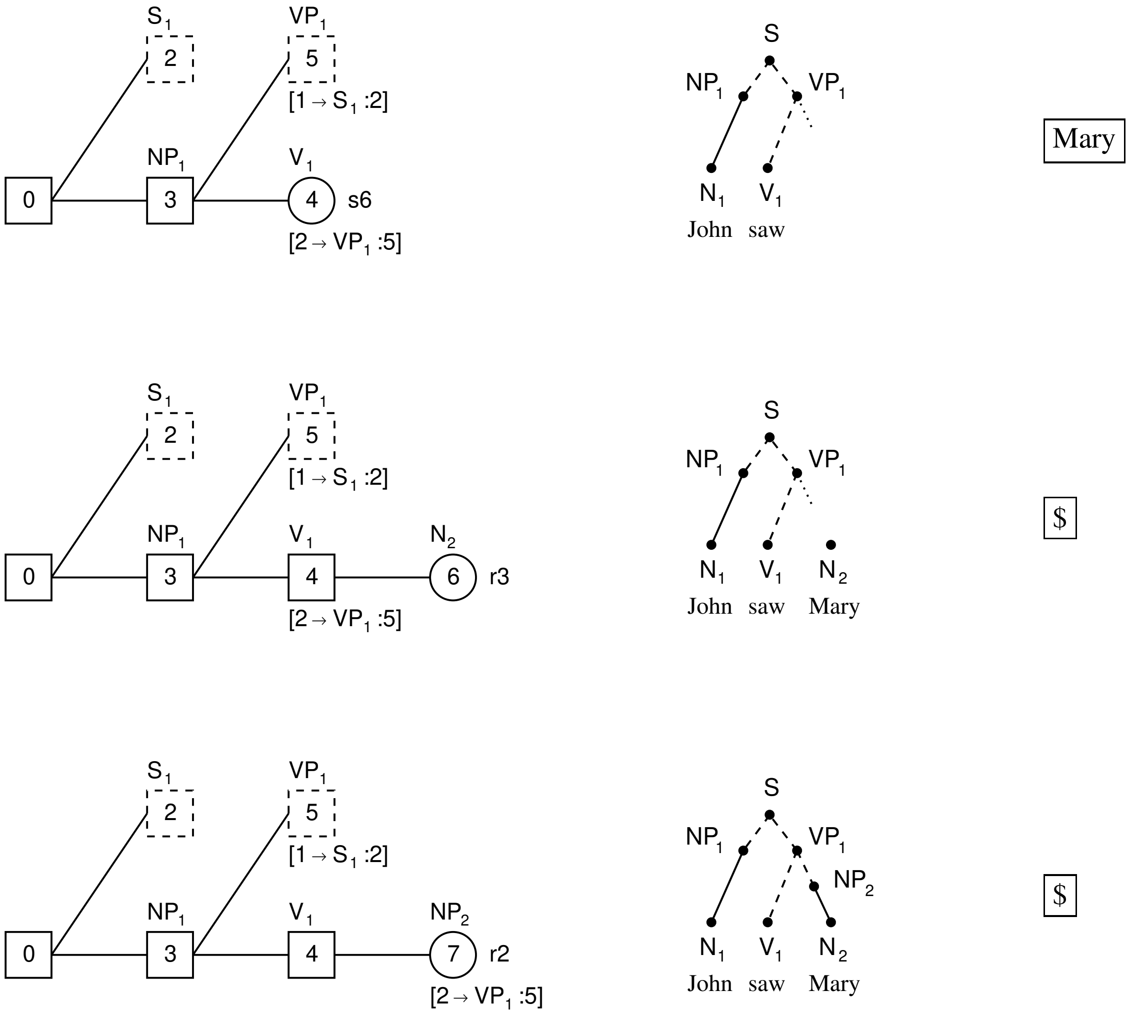


Figure 4.4: Trace of the L∗ algorithm parsing "John saw Mary" (cont.).

therefore installs NP_2 as the rightmost child of VP_1 (third diagram of figure 4.4). The combine pointer is also moved to the vertex with state 7, so that the combine pointer is stored at the vertex associated with the rightmost child of VP_1.

At the new vertex with state 7, ACTION[7, \$] = $\{\mathtt{r2}\}$. There is a combine pointer created by rule 2 at this vertex, so this reduction is a completing reduction, covering the same ground as the eager reduction by rule 2 carried out earlier. The parser pops the RHS elements V_1 and NP_2 off the stack, establishing the vertex with state 5 (identified by the combine pointer) as a new stack top, and marks VP_1 as complete. This action is depicted by changing the dashed lines to solid lines (first diagram of figure 4.5). Unlike other reductions, no new parse structure is

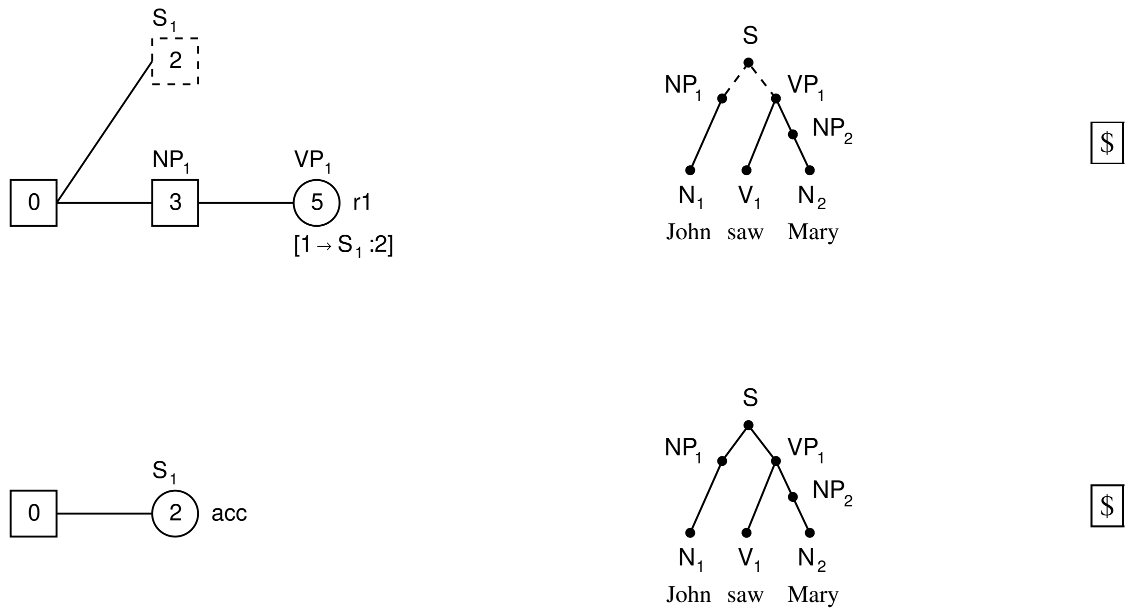


Figure 4.5: Trace of the L$*$ algorithm parsing "John saw Mary" (cont.).

created for the completing reduction, because all relevant structure was constructed earlier by the eager-reduce and combine actions. Having completed VP_1, the parser next performs a second completing reduction, this time by rule 1, which indicates that S_1 is also complete. A single stack top remains, with state 2 (second diagram of figure 4.5). As ACTION[2, $] = $\{\texttt{acc}\}$, the parse succeeds, with S_1 as the root of the resulting parse tree.

4.4 Formal L$*$ algorithm without packing

Input

A parse table ACTION[*state*, *symbol*] for the context-free grammar $G = \langle N, T, R, S\rangle$ and an input string $z \in T^*$. Entries in the parse table are sets of parsing actions. Each action has the form "`shift` s", "`reduce` r", "`eager-reduce` r–k", "`goto` s", "`combine` r", or "`accept`". N is a set of nonterminals, T is a set of terminals, R is a set of grammar rules of the form $X \rightarrow \alpha$, where $X \in N$ and $\alpha \in (N \cup T)^+$, and S is the start symbol. The state s_0 is designated as the start state.

Output

A list of root nodes of a shared parse forest for z if $z \in L(G)$, otherwise an error indication.

Data Structures

A *vertex* in the graph-structured stack is a tuple $\langle s, n, \mathcal{C}, \mathcal{S}\rangle$, where s is a state, n is a forest node, $\mathcal{C}$ is a set of combine pointers, and $\mathcal{S}$ is the set of successor vertices in the stack. For notational convenience, the elements of a vertex v can be referenced using the functions $\mathsf{State}(v)$, $\mathsf{Node}(v)$, $\mathsf{Combine\text{-}Ptrs}(v)$, and $\mathsf{Successors}(v)$.

A *node* in the shared forest is a tuple $\langle X, d\rangle$, where $X \in N$, and d is a derivation, written as a list of child forest nodes. The elements of a node n can be referenced using the functions $\mathsf{Nonterm}(n)$ and $\mathsf{Children}(n)$.

A *combine pointer* is a tuple $\langle r, d, \mathcal{V}\rangle$, where $r \in R$, d is a derivation, and $\mathcal{V}$ is a set of vertices. The elements of a combine pointer c can be referenced using the functions $\mathsf{Rule}(c)$, $\mathsf{Deriv}(c)$, and $\mathsf{Vertices}(c)$.

A *path* is a contiguous sequence of vertices $v_1, \ldots, v_k$ in the stack. That is, for $i = 2, \ldots, k$, $v_i \in \mathsf{Successors}(v_{i-1})$.

FRONTIER stores a set of pairs $\langle v, a\rangle$, where v is a vertex and a is a parse action yet to be performed at v. The vertices within the pairs of this list form the active stack tops.

ω denotes the current input symbol.

Main Loop

- Add a terminator symbol \$ to the end of the input string z
- $\omega \leftarrow$ The first symbol of z
- $v_0 \leftarrow \langle s_0, \text{NIL}, \{\}, \{\}\rangle$ · Initialise the stack
- Call **Schedule**(v_0, ω)
- Loop · Perform reductions followed by shifts until acceptance or rejection.
  - Call **Reduce**()
  - Call **Shift**()
  - If FRONTIER contains only pairs of the form $\langle v$, "`accept`"$\rangle$ then halt and return $\{\mathsf{Node}(v) \mid \langle v$, "`accept`"$\rangle \in$ FRONTIER$\}$
  - If FRONTIER = $\{\}$ then halt and signal an error.

Reduce()

Perform all outstanding reductions by calling the subroutine appropriate to each reduce action. Non-eager-reduce actions are processed first, followed by eager-reduce actions.

- While $\exists x \in$ FRONTIER of the form $\langle v, \text{“}\texttt{reduce}\ X \rightarrow \alpha\text{”}\rangle$:
  - Remove x from FRONTIER
  - $\mathcal{C} \leftarrow \{c \in \mathsf{Combine\text{-}Ptrs}(v) \mid \mathsf{Rule}(c) = X \rightarrow \alpha\}$
  - If $\mathcal{C} \neq \{\}$ then
    - Call **Completing-Reduce**$(v, \mathcal{C})$

    Else
    - $\mathcal{P} \leftarrow \{p \mid p$ is a path of length $|\alpha|$ starting at v in the stack$\}$
    - $\forall p \in \mathcal{P}$, call **Full-Reduce**$(p, X \rightarrow \alpha)$
- While $\exists x \in$ FRONTIER of the form $\langle v, \text{“}\texttt{eager-reduce}\ r\text{–}k\text{”}\rangle$:
  - Remove x from FRONTIER
  - $\mathcal{P} \leftarrow \{p \mid p$ is a path of length k starting at v in the stack$\}$
  - $\forall p \in \mathcal{P}$, call **Eager-Reduce**$(p, r)$

- Process all outstanding non-eager reductions at stack tops.
- Each element of $\mathcal{C}$ corresponds to a previous eager reduction that is now complete.
- Process all outstanding eager reductions at stack tops.

Completing-Reduce$(v, \mathcal{C})$

Perform a completing reduction at vertex v. As this reduction covers the same ground as a previous eager reduction, there is no new parse structure to create, and all that needs to be done is schedule any actions at the vertices associated with the result of the completing reduction.

- $\forall c \in \mathcal{C}$
  - Remove c from $\mathsf{Combine\text{-}Ptrs}(v)$
  - $\forall v' \in \mathsf{Vertices}(c)$
    - Call **Schedule**(v', ω)

- $\mathcal{C}$ is a set of combine pointers that point to the vertices created earlier by an eager reduction that is now being completed.

Full-Reduce($v_1, \ldots, v_k, X \to \alpha$)

Non-eagerly reduce by the rule $X \to \alpha$, creating a new forest node X whose children are the nodes of vertices $v_1, \ldots, v_k$. Combine this new node into other partial derivations created previously by eager reduction, and schedule any further actions triggered by this reduction.

· $n' \leftarrow \langle X, (\mathsf{Node}(v_k), \ldots, \mathsf{Node}(v_1))\rangle$	· Create a new forest node.
· $\Pi \leftarrow$ A partition of Successors(v_k) by goto value on symbol X	· v belongs to the set $\pi_s \in \Pi$ if and only if "`goto` s" $\in$ ACTION[State(v),X]. See figure 4.6(a).
· $\forall \pi_s \in \Pi$ · $v' \leftarrow \langle s, n', \{\}, \pi_s\rangle$	· Create new vertices containing the new forest node, one for each element of Π.

· $\forall v \in \pi_s$
    · $\forall a \in$ ACTION[State(v), ω] s.t. $a =$ "`combine` r"
      · Call **Combine**(v, v', r)
  · Call **Schedule**(v', ω)
	· Combine the newly created n' into previous eager reductions by rule r whose corresponding partial derivations have Node(v) as their rightmost element.

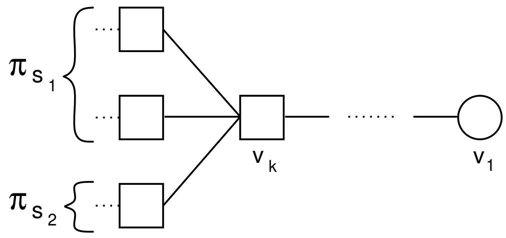


Figure 4.6(a): Stack before full reduction.

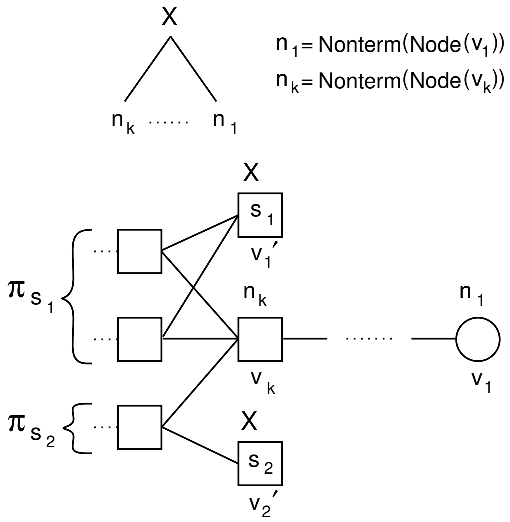


Figure 4.6(b): Stack and new forest node after full reduction.

Eager-Reduce$(v_1, \ldots, v_k, X \rightarrow \alpha)$

Eagerly reduce by the rule $X \rightarrow \alpha$, creating a new incomplete forest node X whose children are the nodes of vertices $v_1, \ldots, v_k$. Combine this new node into other partial derivations created previously by eager reduction, and schedule any further actions triggered by reduction.

- $d \leftarrow$ (Node$(v_k), \ldots,$ Node(v_1))
- $n' \leftarrow \langle X, d \rangle$ · Create a new forest node.
- $c \leftarrow \langle X \rightarrow \alpha, d, \{\} \rangle$ · Create a new combine pointer.
- $\Pi \leftarrow$ A partition of Successors(v_k) by goto value on symbol X · v belongs to the set $\pi_s \in \Pi$ if and only if "`goto` s" $\in$ ACTION[State(v),X]. See figure 4.7(a).
- $\forall \pi_s \in \Pi$ · Create new vertices containing this new node, one for each element of Π.
  - $v' \leftarrow \langle s, n', \{\}, \pi_s \rangle$
  - Add v' to Vertices(c)
  - $\forall v \in \pi_s$ · Combine the newly created n' into previous eager reductions by rule r whose corresponding partial derivations have Node(v) as their rightmost element.
    - $\forall a \in$ ACTION[State$(v), \omega$] s.t. $a =$ "`combine` r"
      - Call **Combine**(v, v', r)
  - Call **Schedule**$(v'$, *EAG*)
- Add c to Combine-Ptrs(v_1)

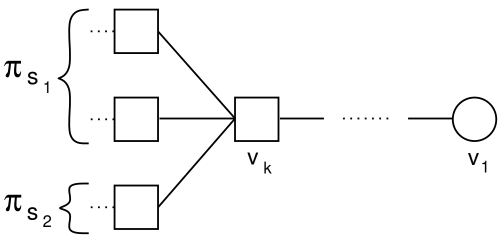


Figure 4.7(a): Stack before eager reduction.

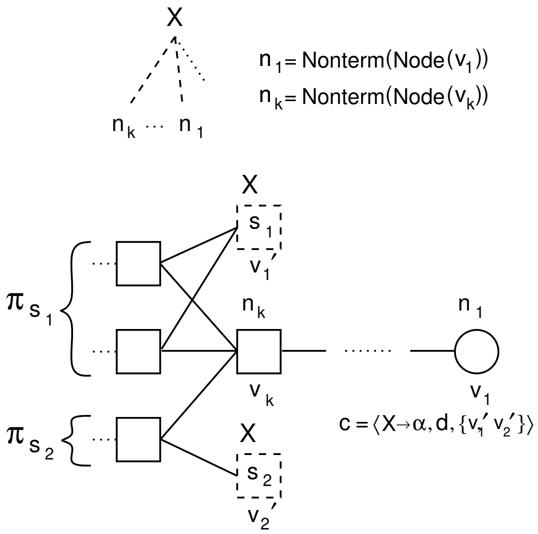


Figure 4.7(b): Stack and new forest node after eager reduction.

Combine(v_f, v_t, r)

Combine Node(v_t) into partial derivations created by eager reduction whose rightmost element is currently Node(v_f). These derivations are identified by the combine pointers associated with v_f.

- $\forall c \in$ Combine-Ptrs(v_f) s.t. Rule(c) = r
  - Add Node(v_t) to the end of Deriv(c)
  - Remove c from Combine-Ptrs(v_f)
  - Add c to Combine-Ptrs(v_t)

- Move the combine pointers forward to v_t so that further combines (or completing reductions) can be performed there.

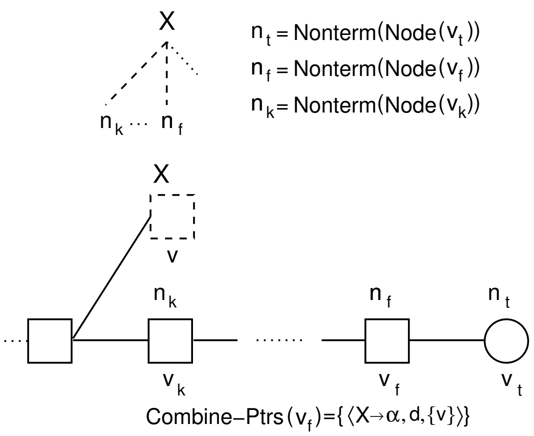


Figure 4.8(a): Stack and forest node before combine.

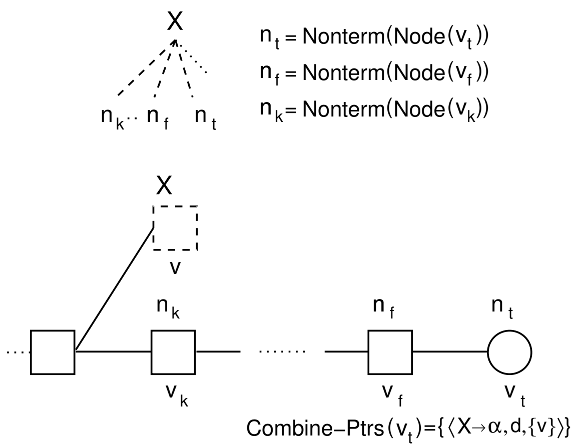


Figure 4.8(b): Stack and forest node after combine.

Shift()

Shift the next terminal symbol onto all the stack tops and create a new node for it in the parse forest. Combine this new node into other partial derivations created previously by eager reduction, and schedule any actions triggered by the shift.

· $n \leftarrow \langle \omega, \{\} \rangle$ · $\omega \leftarrow$ The next symbol of the input string	· Create a new forest node for the shifted input symbol.
· $\mathcal{S} \leftarrow \{\langle v, a \rangle \in$ FRONTIER $\mid a =$ "`shift` s"$\}$ · FRONTIER $\leftarrow$ FRONTIER $- \mathcal{S}$	· $\mathcal{S}$ is the set of all shift actions to perform.
· $\Pi \leftarrow$ A partition of $\mathcal{S}$ according to goto state of the `shift` actions	· Each $\pi_s \in \Pi$ consists of elements of the form $\langle v,$ "`shift` s"$\rangle$.

· $\forall \pi_s \in \Pi$
  · $\mathcal{V} \leftarrow \{v \mid \langle v, a \rangle \in \pi_s\}$
  · $v_s \leftarrow \langle s, n, \{\}, \mathcal{V} \rangle$
	· Create a new vertex for each member of Π.

· $\forall x \in$ FRONTIER of the form $\langle v,$ "`combine` r"$\rangle$ s.t. $v \in \mathcal{V}$
  · Remove x from FRONTIER
  · Call **Combine**(v, v_s, r)
· Call **Schedule**(v_s, ω)
	· Combine the newly created n into previous eager reductions by rule r whose corresponding partial derivations have Node(v) as their rightmost element.

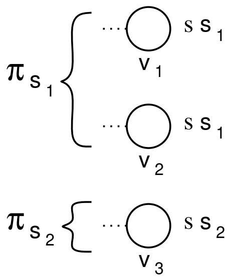


Figure 4.9(a): Stack tops with outstanding shift actions.

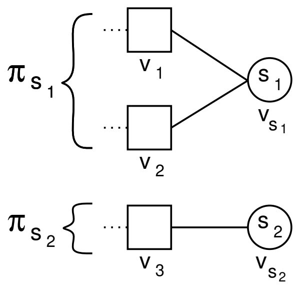


Figure 4.9(b): Stack after shifting.

Schedule(v, L)

Add to FRONTIER all possible actions to be performed at vertex v.

· $\mathcal{A} \leftarrow \{a \in$ ACTION[State$(v), L] \mid$
  $(a =$ "`eager-reduce` r–k" $\Rightarrow$
  $\neg\exists c \in$ Combine-Ptrs(v) s.t. Rule$(c) = r)$
· $\forall a \in \mathcal{A}$, add $\langle v, a \rangle$ to FRONTIER
	· Schedule all actions except eager reductions that repeat an eager reduction carried out earlier.

□

Chapter 5

L∗ Parsing Extensions

The L∗ algorithm described in chapter 4 can be extended in two ways. Section 5.1 presents an extension that eliminates inefficiencies inherited from GLR parsing. Section 5.2 presents an extension that allows the L∗ parser to be interfaced with a modular NLP system. Section 5.3 presents an example of parsing with these extensions, and section 5.4 presents a formal specification of the L∗ algorithm including the two extensions.

5.1 Subtree Sharing

A GLR parser uses a *packed shared forest* to represent all possible parses of a sentence. Sharing and packing are important because they provide a method of compactly representing the parse, and they eliminate the repetition of work when building parse forests. Sharing results in subtrees being parsed and represented only once if they are headed by the same nonterminal, cover the same input substring, and have the same structure. Packing results in subtrees being put together in a single forest node if they are headed by the same nonterminal, cover the same input substring, but have different structure. Further parses are then performed only once for the entire node, rather than once for each member of the packed node.

The parser must recognise opportunities to share or pack parses. Where the parser fails to do so, it will repeat work unnecessarily. For example, consider the possible parses of the sentence "A B C D E" using grammar 5.1.

(1) $S \rightarrow X_h\ Z\ E$
(2) $S \rightarrow Y\ Z\ E_h$
(3) $X \rightarrow A$
(4) $Y \rightarrow A$
(5) $Z \rightarrow W_h\ D$
(6) $W \rightarrow B_h\ C$

Grammar 5.1

The optimally packed shared parse forest for this sentence is shown in figure 5.1. The L$*$ parser, however, does not produce this parse forest. Instead it produces the forest shown in figure 5.2. There are two sources of non-optimality in this parse forest. The L$*$ parser creates two separate nodes for S, instead of packing the two parses of S together into a single forest node. This is because the L$*$ algorithm as stated does not perform local ambiguity packing. This problem is addressed in chapter 6.

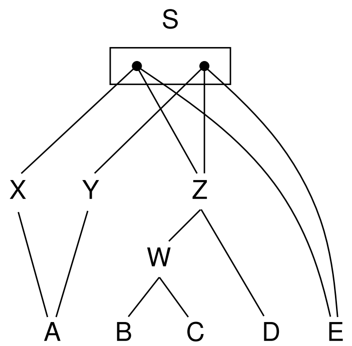


Figure 5.1: Optimal parse forest for the sentence "A B C D E" with grammar 5.1.

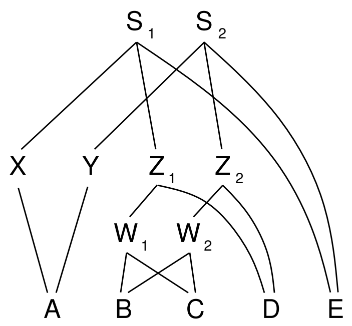


Figure 5.2: L$*$ parse forest for the sentence "A B C D E" with grammar 5.1.

State	Action											
	A	B	C	D	E	$	*EAG*	S	X	Y	Z	W
0	s3							g2	g4	g1		
1		s11									g13	g12
2						acc						
3		r4, r3										
4		s5									g7	g6
5			e6-1									
			s10, c6									
6				e5-1			e5-1					
				s9, c5								
7					e1-2		e1-2					
					s8, c1							
8						r1						
9					r5							
10				r6								
11			s16									
12				s15								
13					s14							
14						r2						
15					r5							
16				r6								

Table 5.1: L$*$ parse table for grammar 5.1.

The other source of non-optimality is the missed opportunities to share nodes in the parse forest. The nodes W_1 and W_2 are identical: they have the same nonterminal and the same children. Only one node for W should be created, and this node should be shared wherever needed. The L$*$ parser, however, creates the two separate forest nodes W_1 and W_2. This in turn leads to two separate nodes Z_1 and Z_2 (one for each W), when only one node for Z should be created.

To see how these problems arise, it is useful to trace the actions of the parser in processing this sentence. The parser employs the parse table shown in table 5.1.

1. The parser shifts A onto the stack, then reduces it to X_1 by rule 3, and Y_1 by rule 4.

2. The parser shifts B onto the stack, leaving the stack as shown in figure 5.3. Next, on the upper branch of the stack, the parser eagerly reduces B_1 to W_1 by rule 6. This is followed by a cascaded eager reduction by rule 5, creating the incomplete node Z_1. A further cascaded reduction by rule 1 then establishes the right-attachment of Z_1 to the previously parsed X_1, creating S_1. On the lower branch of the stack with Y followed by B, no eager reduction is performed because no attachments would result from the eager reduction.

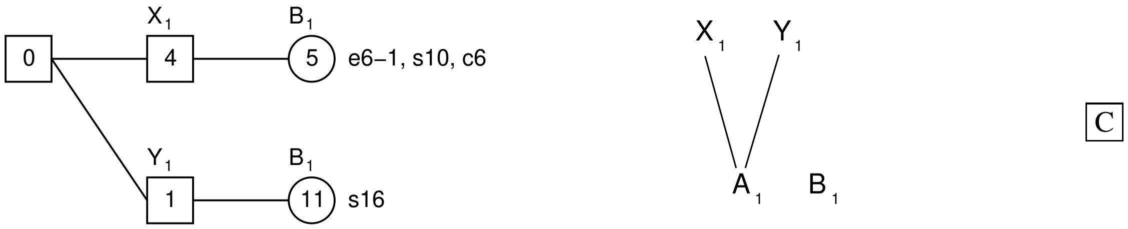


Figure 5.3: State of the L∗ parser after processing “A B” with table 5.1.

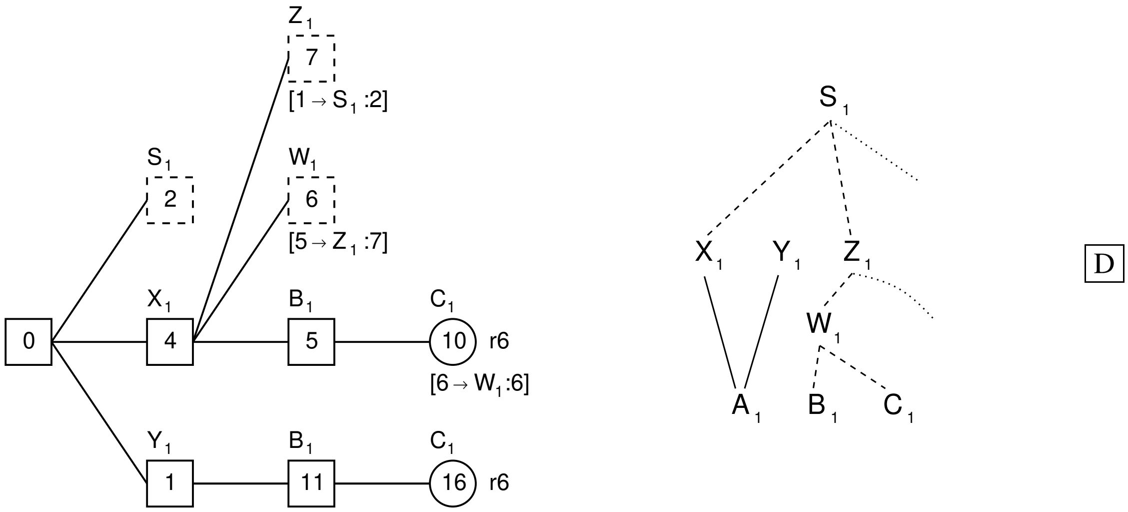


Figure 5.4: State of the L∗ parser after processing “A B C” with table 5.1.

3. The parser shifts C onto both branches of the stack separately, and also combines C into W_1, which was created from eager reduction by rule 6. The stack at this point is shown in figure 5.4. Next, the parser performs a completing reduction by rule 6 on the upper branch of the stack, marking W_1 as complete. On the lower branch of the stack, the parser then performs a separate reduction by rule 6, creating a second derivation of W.

4. The parser shifts D onto both branches of the stack, and also combines D into Z_1, on the upper branch of the stack. This leaves the stack as shown in figure 5.5. Following this, the parser performs a completing reduction by rule 5 on the upper branch of the stack. The parser also performs a separate reduction by rule 5 on the lower branch of the stack, creating a second derivation of Z.

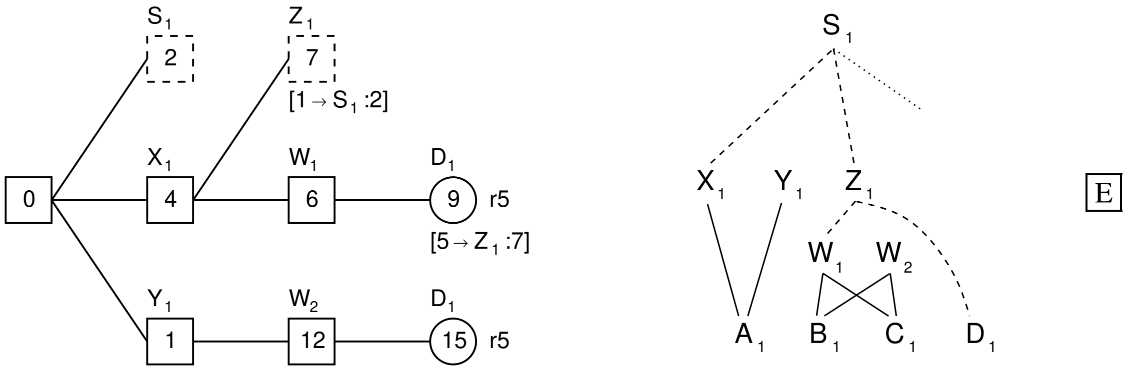


Figure 5.5: State of the L∗ parser after processing “A B C D” with table 5.1.

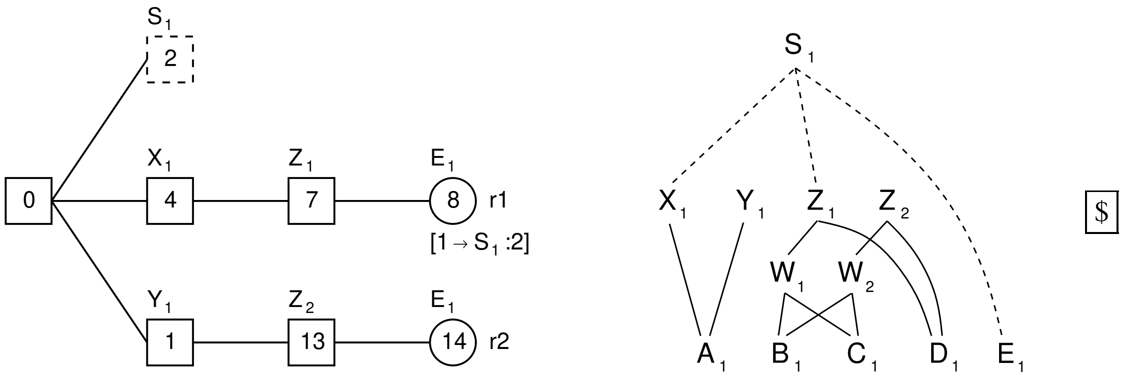


Figure 5.6: State of the L∗ parser after processing “A B C D E” with table 5.1.

5. The parser shifts E onto the stack, and combines E into the incomplete node S_1 (as shown in figure 5.6). A completing reduction by rule 1 on the upper branch of the stack, and a reduction by rule 2 on the lower branch of the stack create the complete parses S_1 and S_2 of the sentence “A B C D E”, as shown in figure 5.2.

The non-optimal sharing in the parse forest results from contextual distinctions in the states of the parse table, which restrict the opportunities to share. Sharing only occurs when branches of the stack are rejoined. Results from the common branch are shared by all the rejoined branches. For example, in figure 5.7, if “A B” at the top of the stack is reduced to an X, this X can be shared by both branches Y and Z. Branches of the stack are only rejoined, however, when the states pushed onto those branches are equal, as shown in figures 5.8 and 5.9. Two states of the table may not be equal as a result of contextual differences encoded in the state, even though they both

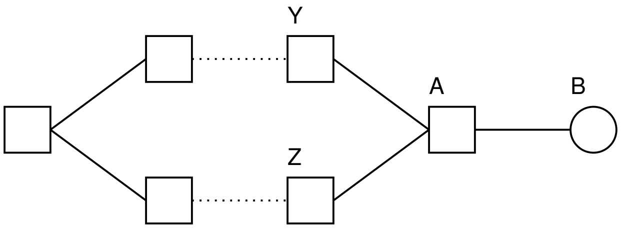


Figure 5.7: Rejoining of the stack allows sharing.

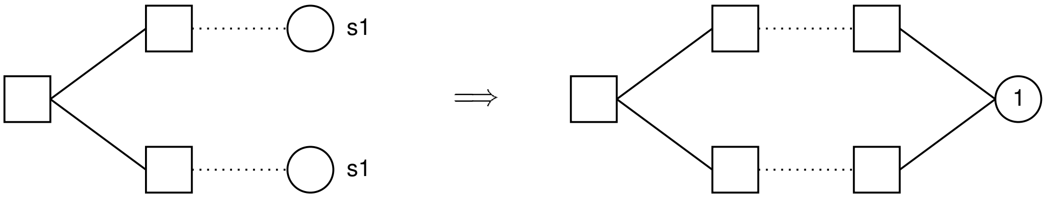


Figure 5.8: The stack is rejoined when states are equal.

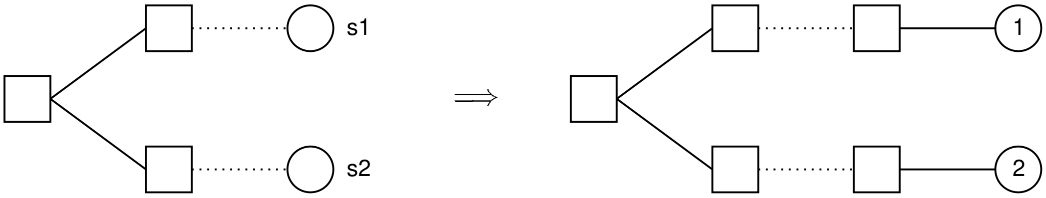


Figure 5.9: The stack is not rejoined when states are not equal.

represent the parser being at the same point in processing the same set of grammar rules. When two or more such states both occur in the stack at the same time, they will be on different branches that cannot be rejoined, and will therefore cause the same grammar rules to be applied separately on each branch.

To see how such contextual distinctions arise, consider states 5 and 11 from the previous example. These states both represent the same position in processing the same set of rules. This can be seen by examining the DFA shown in figure 5.10, from which the parse table of table 5.1 was constructed. Both states consist of a single item containing the grammar rule W $\rightarrow$ B$_h$ C, with the dot after the B. The two items (and therefore states) are not equal, however, because of a difference in the setting of their attachment flag. This difference results from a difference in the context from which the items were created. The attachment flag of the item [W $\rightarrow$ B$_h$ $\cdot$ C, T] in

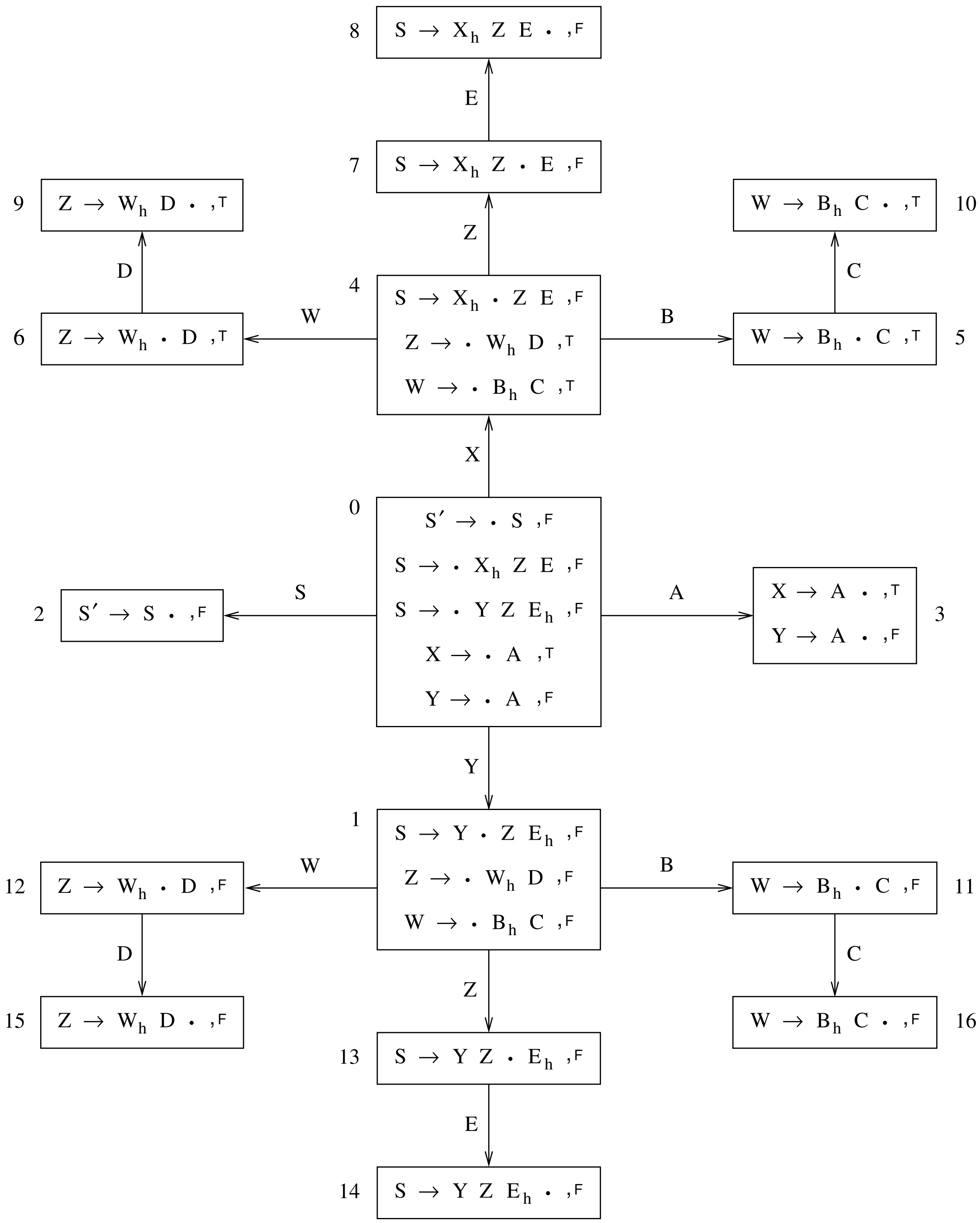


Figure 5.10: L∗ DFA for grammar 5.1.

state 5 is set because the item comes from a context of having first parsed an X, meaning that an attachment can be established by rule 1 after the head of W $\rightarrow$ B$_h$ C is parsed. The attachment flag of the item [W $\rightarrow$ B$_h$ $\cdot$ C, F] in state 11 is not set, because the item comes from a context of having first parsed a Y, meaning that no attachment results from parsing W $\rightarrow$ B$_h$ C. Because the states are differentiated by the setting of the attachment flag, the parser incorrectly creates separate derivations of W depending on whether an X or a Y was first parsed. This can be seen in figure 5.4, where a reduction by rule 6 is specified separately on both the upper and lower branches of the stack, creating the two separate forest nodes W_1 and W_2.

Increasing the contextual information encoded in a state results in an increase of missed opportunities for sharing. For example, adding lookahead to items can reduce sharing in the parse forest (Billot and Lang, 1989; Lankhorst, 1991).

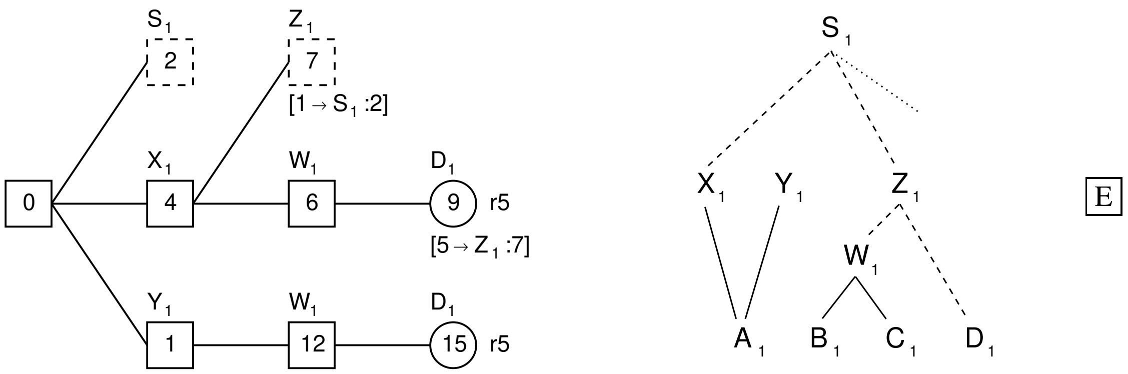


Figure 5.11: State of the L$*$ parser (reusing derivations) after processing "A B C D".

Rekers (1992) has also studied this problem. He proposes a solution that bases the decision to share on the equality of derivations in the parse forest, rather than equality of states in the parse table. When the parser performs a reduction, it searches for an existing derivation with exactly the same children as the symbols being used in the current reduction. If such a derivation exists, the parser reuses it instead of creating a new one. This results in an optimally shared parse forest. However, it does not eliminate unnecessary actions performed by the parser. The stack is still constructed as in GLR parsing, with branches only being rejoined when states are equal. Thus reductions are duplicated on separate branches of the stack, although the same derivation is used as the result of both reductions.

For example, applying Rekers's algorithm in the previous example of parsing the sentence

“A B C D E” with grammar 5.1, the parser processes “A B C” in exactly the same way as explained above, resulting in a stack identical to that shown in figure 5.4. However, when the reduction by rule 6 is performed on the lower branch of the stack, the parser searches for an existing derivation of W with children B and C, and finds the node W_1. Therefore it reuses W_1 as the result of the reduction on the lower branch of the stack. Having performed this reduction, the parser shifts D onto the stack, leaving the stack as shown in figure 5.11 (cf. figure 5.5). There are still two distinct branches in the stack, and the reduction by rule 6 is still performed twice, even though the same result was used both times.

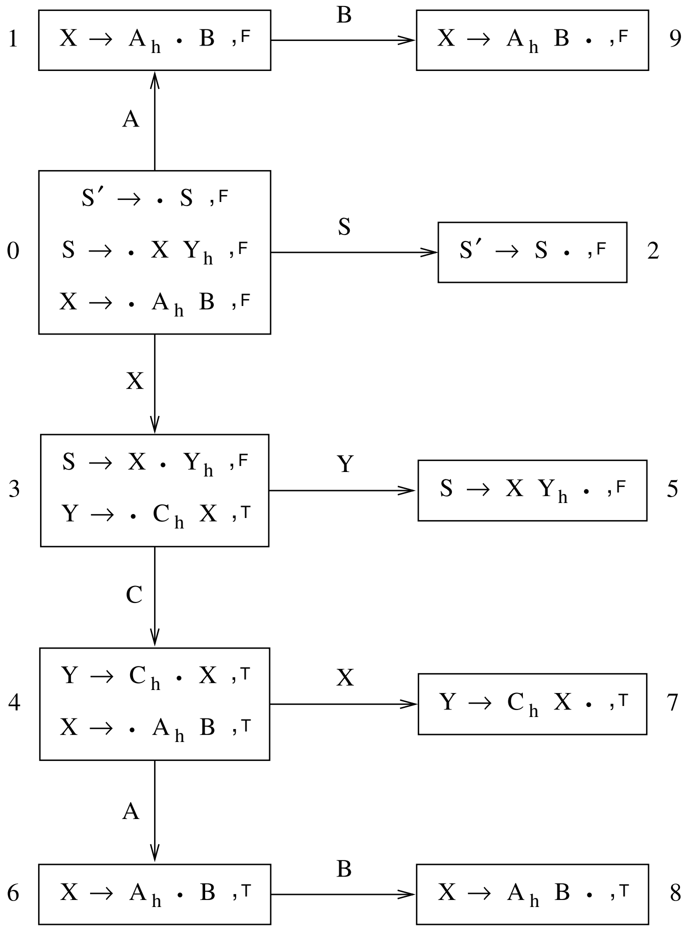


Figure 5.12: L$*$ DFA for grammar 5.2.

An alternative approach worth considering is to merge states that have a common *core*. The core of a state is the set of cores of the items in that state, where the core of an item is the rule and position of its dot. For example, in figure 5.10, states 5 and 11 both have the core $\{[\mathrm{W} \rightarrow \mathrm{B}_h \ \cdot \ \mathrm{C}]\}$.

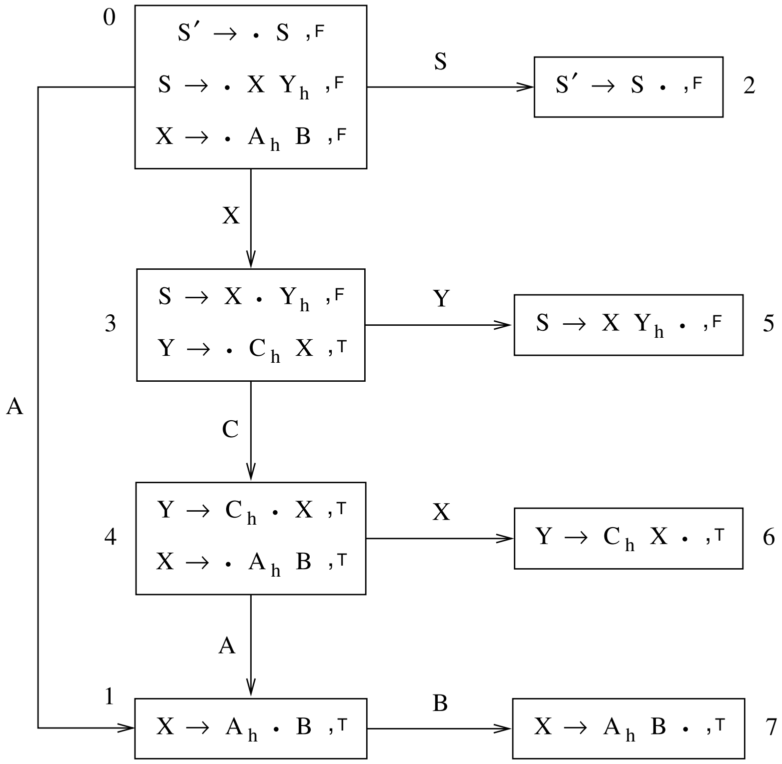


Figure 5.13: L∗ DFA, after merging common core, for grammar 5.2.

STATE	ACTION							
	A	B	C	$	*EAG*	S	X	Y
0	s1					g2	g3	
1		e3-1						
		s7, c3						
2				acc				
3			s4					g5
4	e2-1, s1						g6, c2	
5				r1	e1-2			
6				r2	e2-2			
7			r3	r3				

Table 5.2: L∗ parse table for grammar 5.2.

By merging states with a common core, the effect of extra contextual information such as the attachment flag is removed. The idea of merging states is used in LALR table building. States in an LR DFA with the same core are merged together, with the lookahead of the resulting state being the union of the lookahead of the merged states.

Unfortunately, using this method to construct an L$*$ parse table means that the parser may perform eager reductions where no attachments result. For example, consider the following grammar:

$$
\begin{array}{lll}
(1) & \mathrm{S} \rightarrow \mathrm{X}\,\mathrm{Y}_h & \\
(2) & \mathrm{Y} \rightarrow \mathrm{C}_h\,\mathrm{X} & \qquad\qquad \text{Grammar 5.2} \\
(3) & \mathrm{X} \rightarrow \mathrm{A}_h\,\mathrm{B} &
\end{array}
$$

Figure 5.12 shows the L$*$ DFA of this grammar. States 1 and 6 have common cores, so are merged into a single state, as are states 8 and 9. The resulting DFA is shown in figure 5.13. Table 5.2 shows the parse table constructed from this DFA. Now consider parsing the sentence "A B C A B" with this parse table. Firstly, the parser shifts A onto the stack, leaving the parser in state 1, and making B the next word to process. The table entry ACTION[1, B] = {`e3-1`, `s7`, `c3`}, so the parser eagerly reduces A to an X by rule 3. However, ACTION[3, *EAG*] = {}, so no further eager reductions are performed. No attachments have been established, so the parser has performed the extra work of the eager reduction for no gain.

STATE	EQUIV.	ACTION											
	CLASS	A	B	C	D	E	$	*EAG*	S	X	Y	Z	W
0	0	s3							g2	g4	g1		
1	1		s11									g13	g12
2	2						acc						
3	3		r4, r3										
4	4		s5									g7	g6
5	5			e6-1 s10, c6									
6	6				e5-1 s9, c5			e5-1					
7	7					e1-2 s8, c1		e1-2					
8	8						r1						
9	9					r5							
10	10				r6								
11	5			s16									
12	6				s15								
13	11					s14							
14	12						r2						
15	9					r5							
16	10				r6								

Table 5.3: New L$*$ parse table for grammar 5.1, constructed from the DFA in figure 5.10.

To avoid unjustified eager reductions, states should only be merged when warranted by the parse. If, during parsing, two states with the same core occur together at the top of the stack, they should be merged. To identify such states, the parser uses equivalence classes. An *equivalence class* contains all items with the same core. The equivalence class is encoded as an integer stored with each state in the parse table. For example, the L$*$ parse table with equivalence classes for grammar 5.1 is shown in table 5.3. This parse table is calculated from the DFA shown in figure 5.10. States 5 and 11 are both members of equivalence class 5, because they both have the same items with dots in the same position. Similarly, states 10 and 16 both belong to equivalence class 10.

When the parser is about to create two or more vertices with states from the same equivalence class, it merges these states into a single vertex. The actions to perform at this vertex are the union of the actions to perform in the individual states.

For example, consider parsing the sentence "A B C D E" using the new parse table shown in table 5.3. There is one difference to the stack diagrams drawn previously. The equivalence class of a vertex is drawn above a list of states stored in the vertex. Both these numbers are drawn inside the vertex.

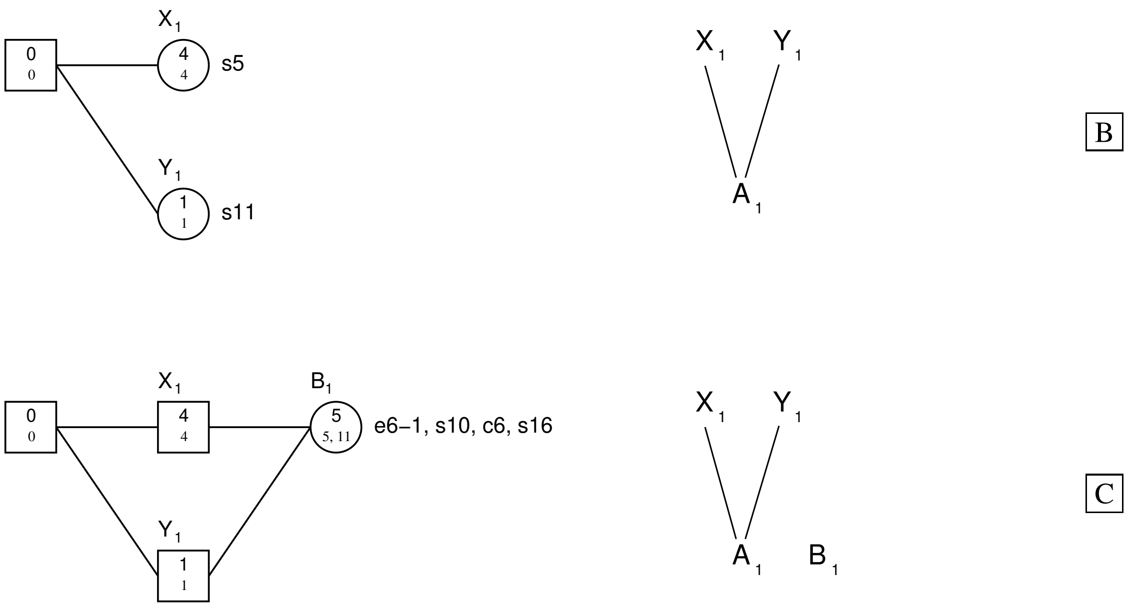


Figure 5.14: State of the L$*$ parser after processing "A B" with table 5.3.

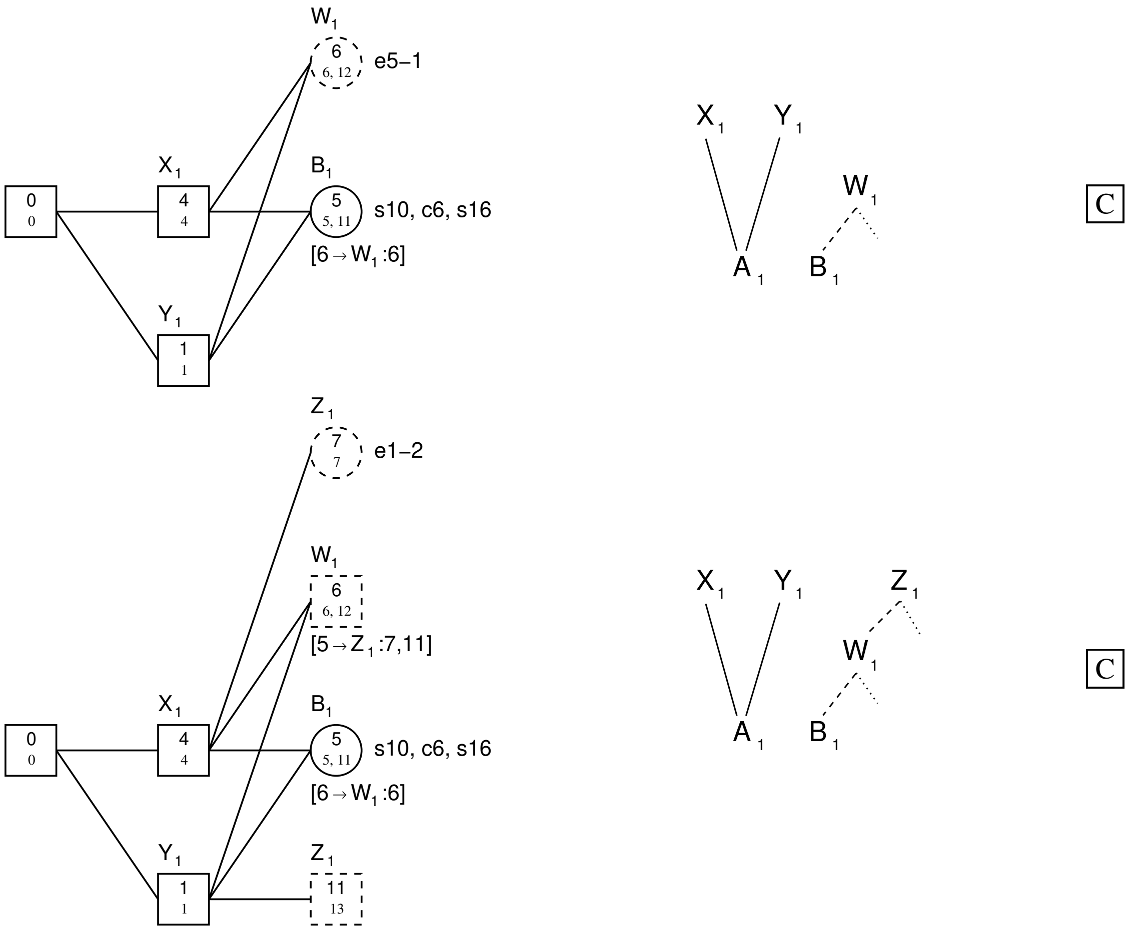


Figure 5.15: State of the L∗ parser after processing "A B C" with table 5.3.

1. The parser shifts A onto the stack, then reduces it to an X by rule 3, and a Y by rule 4, leaving the stack as shown in the first diagram of figure 5.14.

2. There are outstanding shift actions to states 5 and 11. However, rather than creating two separate vertices from these shifts, the parser creates only a single vertex which represents the merger of states 5 and 11, because both states belong to equivalence class 5. The actions to perform at the new vertex are the union of the actions to perform in states 5 and 11. The result is shown in the second diagram of figure 5.14. Next, the parser eagerly reduces B to a W by rule 6. The entries ACTION[4, W] = {g6} and ACTION[1, W] = {12} determine the states of the resulting vertices. Both states 6 and 12 belong to equivalence class 6, so again, the parser creates a single vertex representing the merger of these two states

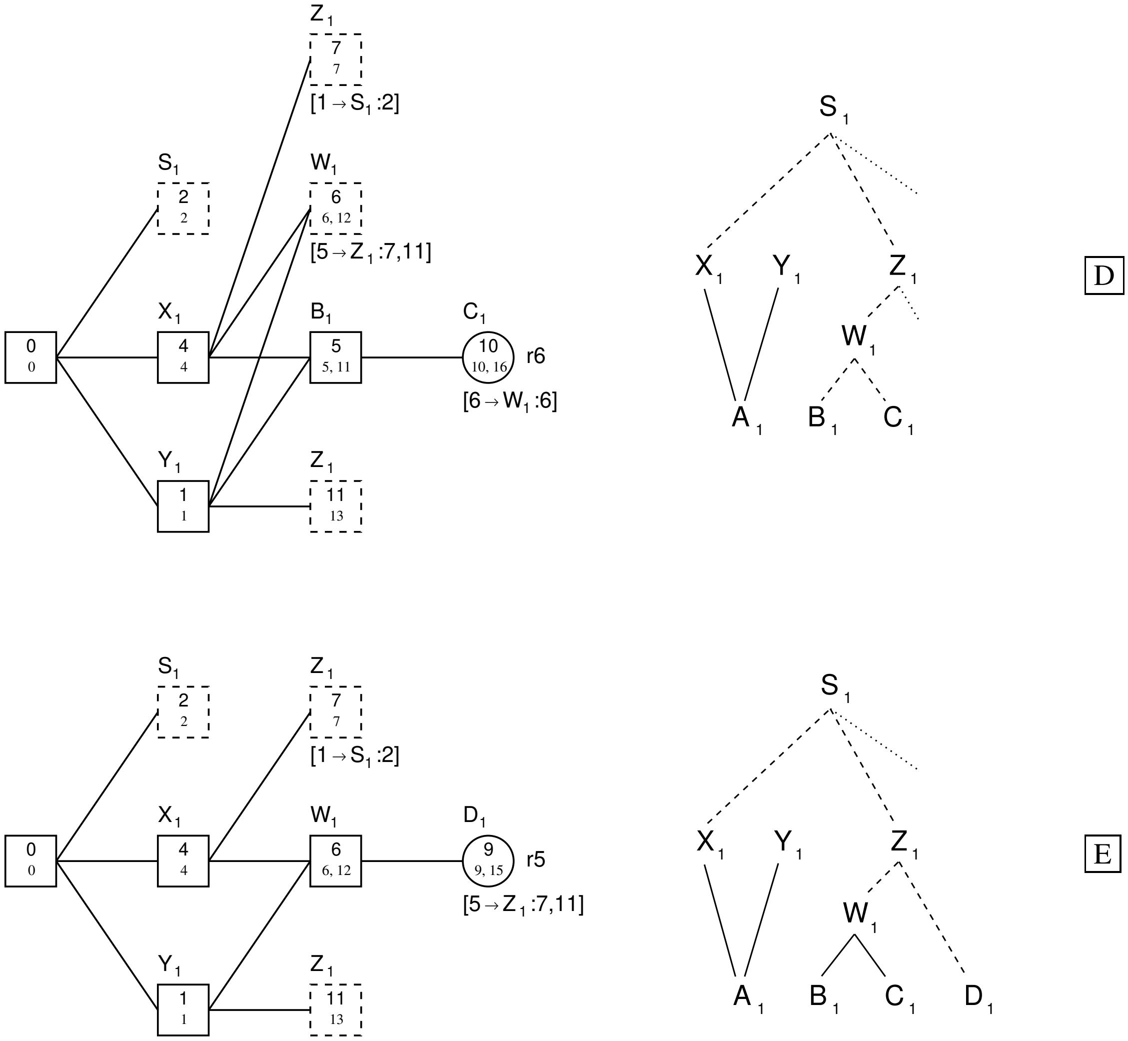


Figure 5.16: State of the L∗ parser after processing “A B C D” with table 5.3.

(first diagram of figure 5.15). Next, the parser performs a cascaded reduction by rule 5 at the new vertex, creating the incomplete node Z_1. Note that the parser also creates two separate vertices in the stack as a result of this reduction because ACTION[4, Z] = {`g7`} and ACTION[1, Z] = {`g13`}, and states 7 and 13 belong to different equivalence classes. This leaves the stack as shown in the second diagram of figure 5.15. The parser then performs a further cascaded eager reduction by rule 1 that establishes the right-attachment of Z_1 to the previously parsed X_1, creating the incomplete node S_1.

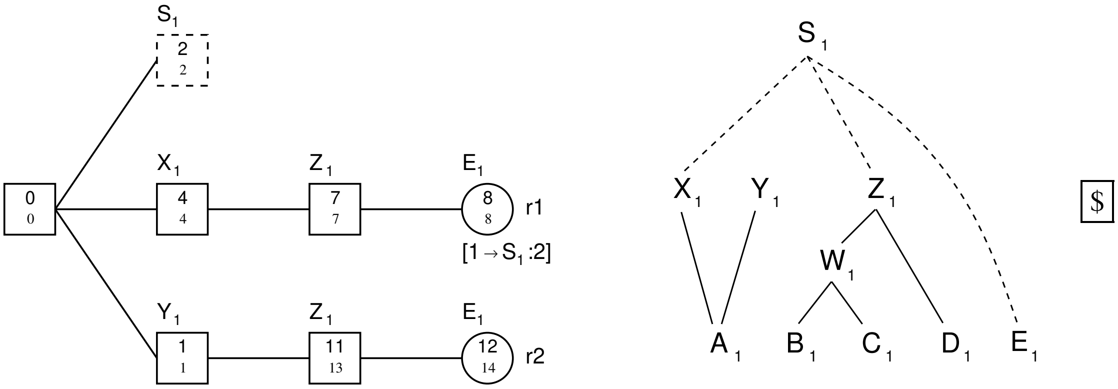


Figure 5.17: State of the L∗ parser after processing “A B C D E” with table 5.3.

3. There are two outstanding shift actions to perform at the stack top with equivalence class 5. However, both state 10 and state 16 belong to equivalence class 10, so the parser only creates a single vertex as a result of shifting C onto the stack. The parser also combines C into the incomplete W_1 because $c6 \in$ ACTION[5, C] (first diagram of figure 5.16). Next, the parser performs a completing reduction by rule 6, marking W_1 as complete.

4. The parser shifts D onto the stack, and D is combined into Z_1. This leaves the parser as shown in the second diagram of figure 5.16. Next, the parser performs a completing reduction by rule 5, marking Z_1 as complete.

5. The parser shifts E onto the stack, and E is combined into the incomplete node S_1 (as shown in figure 5.17). A completing reduction by rule 1 on the upper branch of the stack and a reduction by rule 2 on the lower branch of the stack create the complete parses S_1 and S_2 of the sentence “A B C D E”, as shown in figure 5.1.

The parser performed only one reduction by rule 6, and one reduction by rule 5, and therefore only created one forest node for W and one node for Z.

As discussed earlier, increasing the length of lookahead k in LR(k) parse tables results in a loss of sharing and therefore a loss of efficiency that makes these parse tables impractical. Equivalence classes can also be employed to eliminate the contextual distinctions introduced by lookahead in LR(k) tables, making parsing with such tables practical.

5.2 Integrating the L$*$ parser with an external oracle

The L$*$ parser is intended to form part of a modular natural language processing system. The NLP system can provide feedback to the L$*$ parser on the validity or plausibility of the parses it is pursuing, which hopefully leads to greater efficiency because the system only pursues those parses which are syntactically, semantically, and pragmatically consistent.

The larger NLP system can be modelled as an *oracle* that evaluates parses. The oracle may reject a parse on the basis of information such as mismatched grammatical features, selectional restrictions, or pragmatic information. Whenever a reduce or a combine action is performed, the L$*$ parser accepts input from the oracle, giving an evaluation of the new structure.

When the oracle rejects a parse, the parser should cease all further work on that parse. When it is a full reduction that created the rejected parse, this can be achieved by deleting any vertices created by the rejected reduction. For example, consider parsing the sentence "A B C D" with the following grammar, a parse table for which is shown in figure 5.4:

$$
\begin{array}{lrcl}
(1) & S & \rightarrow & V\ X_h \\
(2) & S & \rightarrow & W\ Y_h \\
(3) & V & \rightarrow & A \\
(4) & W & \rightarrow & A \\
(5) & X & \rightarrow & B_h\ C\ D \\
(6) & Y & \rightarrow & Z_h\ D \\
(7) & Z & \rightarrow & B\ C_h
\end{array}
\qquad \text{Grammar 5.3}
$$

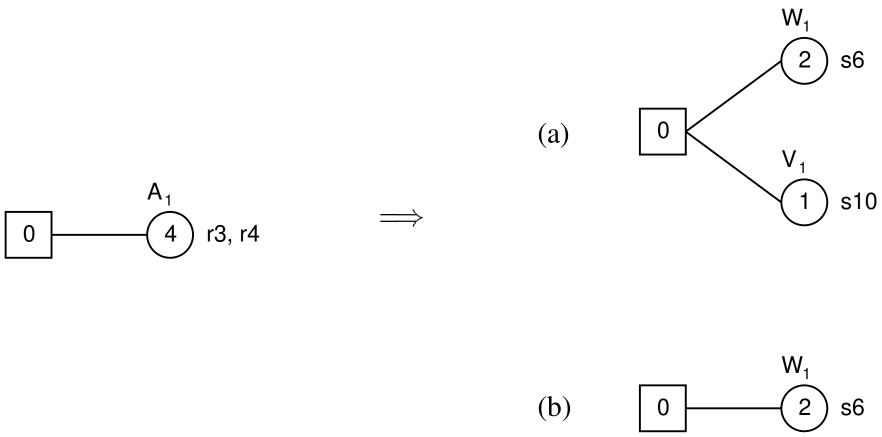


Figure 5.18: Two possible outcomes of performing reductions by rules 3 and 4.

STATE	ACTION											
	A	B	C	D	$	*EAG*	S	W	V	X	Y	Z
0	s4						g3	g2	g1			
1		s10								g11		
2		s6									g7	g5
3					acc							
4		r3, r4										
5				e6-1		e6-1						
				s9, c6								
6			s8									
7					r2	e2-2						
8				r7								
9					r6							
10			e5-1									
			s12, c5									
11					r1	e1-2						
12				s13, c5								
13					r5							

Table 5.4: L$*$ parse table for grammar 5.3.

Firstly, the parser shifts A onto the stack, leaving the stack as shown on the left-hand side of figure 5.18. The actions to perform next are given by ACTION[4, B] = $\{\texttt{r3}, \texttt{r4}\}$. The parser performs each of these reductions and passes the result to the oracle for evaluation. If the oracle does not reject either reduction, then the parser operates as usual, and the stack after the two reductions is as shown in figure 5.18(a). If, however, the oracle rejects the reduction by rule 3, then the parser deletes the vertex with state 1, leaving the stack as shown in figure 5.18(b).

If the rejected parse was created by an eager reduction or combine action, then there are additional complications. It is insufficient just to delete the vertices created by the rejected action. No missing constituents of a rejected incomplete derivation should be parsed and combined into the derivation. Nor should any completing reduction of a rejected derivation be attempted.

A rejected parse may also contain incomplete children, in which case no further work should be done to complete these children. Therefore, the complete definition of a *rejected parse* is a parse that has been rejected by the oracle, or a parse for which all possible parents have been rejected.

For example, consider parsing the sentence “A B C D” again with grammar 5.3. First, the parser shifts A onto the stack, and reduces it to both a V and a W. Next, the parser shifts B onto the stack, leaving the state of the parser as shown in the first diagram of figure 5.19. The parser then eagerly reduces by rule 5, creating an incomplete derivation of X (second diagram of figure 5.19).

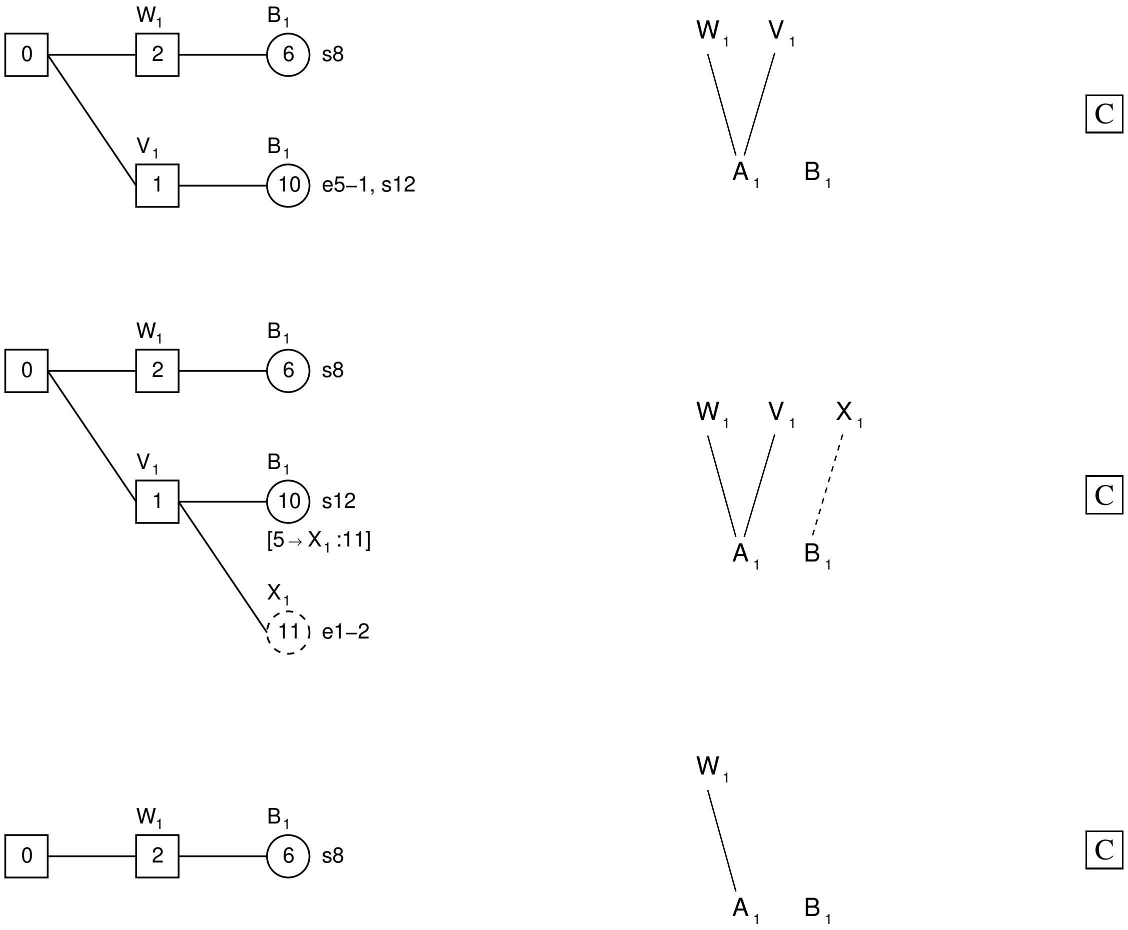


Figure 5.19: Trace of the L∗ algorithm parsing "A B C D" with grammar 5.3.

This triggers a cascaded reduction by rule 1, producing an incomplete parse of an S. Suppose that the oracle rejects this incomplete parse. Not only should all further work on the S be abandoned, but any further work on parsing X_1 should also be stopped. The only way X_1 can be used in a complete parse of the sentence has been rejected, so it is pointless to complete X_1. Thus the parser deletes both the vertex with state 11 and the vertex with state 10, leaving the stack as shown in the third diagram of figure 5.19.

Care must be taken when deleting vertices in the stack. A state of the parse can represent the parser being part way through parsing a number of different rules. In this case, the vertex cannot be deleted until all possible parses have been rejected. For example, consider parsing the sentence "A B C D" with grammar 5.4, which is a slight modification of grammar 5.3. A parse table for this new grammar is shown in table 5.5.

(1) S $\rightarrow$ W X$_h$
(2) S $\rightarrow$ W Y$_h$
(3) W $\rightarrow$ A
(4) X $\rightarrow$ B$_h$ C D
(5) Y $\rightarrow$ Z$_h$ D
(6) Z $\rightarrow$ B C$_h$

Grammar 5.4

STATE	ACTION										
	A	B	C	D	$	*EAG*	S	W	X	Y	Z
0	s3						g2	g1			
1		s4							g7	g6	g5
2					acc						
3		r3									
4			e4-1								
			s9, c4								
5				e5-1		e5-1					
				s8, c5							
6					r2	e2-2					
7					r1	e1-2					
8					r5						
9				r6							
				s10, c4							
10					r4						

Table 5.5: L$*$ parse table for grammar 5.4.

The parser shifts A onto the stack and reduces it to a W. Next, it pushes B onto the stack and eagerly reduces it to an X by rule 4 (first diagram of figure 5.20). This triggers a cascaded reduction by rule 1; suppose that the oracle rejects the resulting incomplete derivation of S. The only possible parent of the eagerly created X has been rejected, so the parse of X is rejected, and the vertex with state 7 is deleted (second diagram of figure 5.20). The vertex with state 4 cannot be deleted, however, because C may still be used as part of parsing a Z, and needs to be shifted onto the stack to continue this parse. State 4 represents being part way through two different partial parses of the input seen so far, namely X $\rightarrow$ B$_h$ C D and Z $\rightarrow$ B C$_h$, as can be seen by examining the items in the DFA for state 4 (figure 5.21).

More generally, state 4 contains two *kernel* items—items where the dot is not at the start of the RHS. The number of kernel items valid for a given lookahead determines the number of partial parses represented by the vertex. This can be calculated during the parse table building process and stored in a separate table. An entry KCOUNTS[cl, L] in this table is the number of

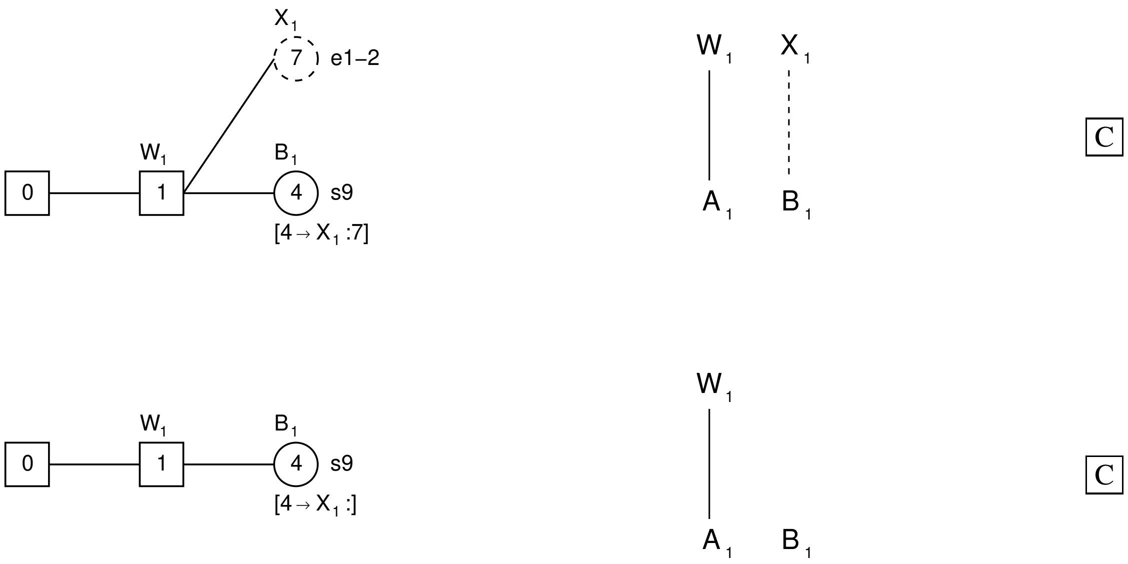


Figure 5.20: Trace of the L∗ algorithm parsing “A B C D” with grammar 5.4.

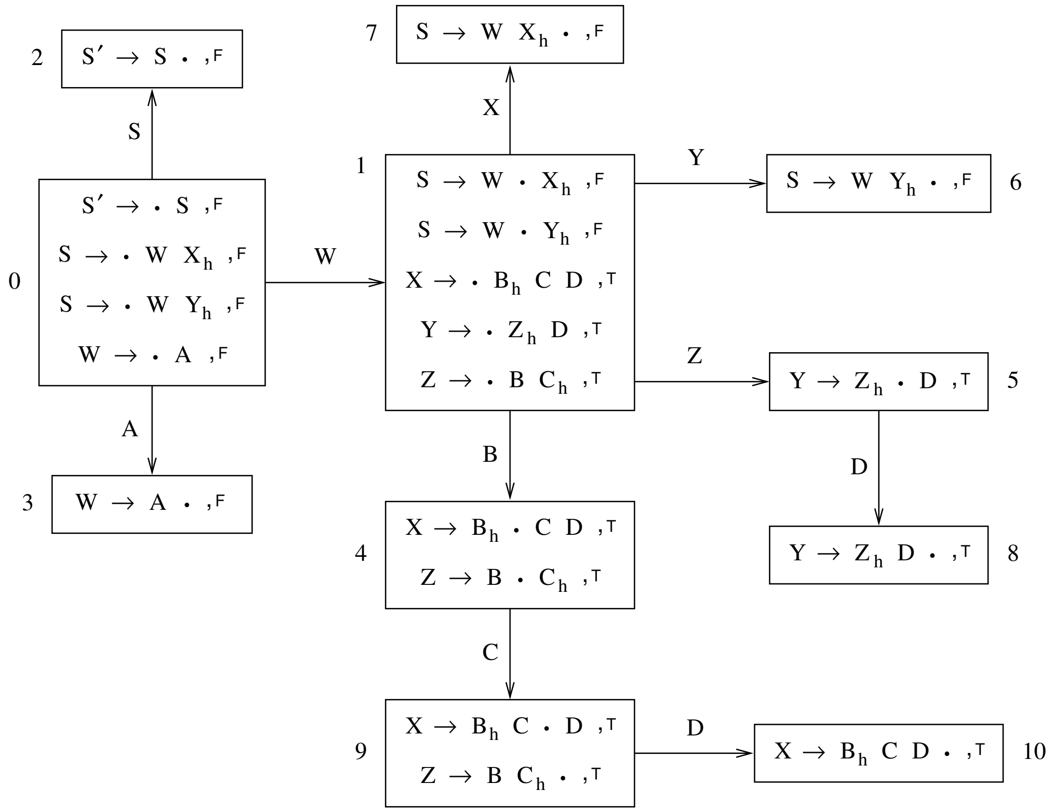


Figure 5.21: DFA for the grammar 5.4.

kernel items that are valid in equivalence class cl with L as the lookahead. It is straightforward to construct this table: for each kernel item $[\mathrm{X} \rightarrow \alpha \cdot \beta, a]$ of a state I, increment the count KCOUNTS[I, x] for every $x \in$ FIRST(β), or if β is empty, then for every $x \in$ FOLLOW(X). The entry KCOUNTS[I, *EAG*] is the total number of kernel items in state I. For example, table 5.6 shows the KCOUNTS table constructed by this method from the DFA of figure 5.21.

CLASS	KCOUNTS					
	A	B	C	D	$	*EAG*
0	0	0	0	0	0	0
1	0	2	0	0	0	2
2	0	0	0	0	1	1
3	0	1	0	0	0	1
4	0	0	2	0	0	2
5	0	0	0	1	0	1
6	0	0	0	0	1	1
7	0	0	0	0	1	1
8	0	0	0	0	1	1
9	0	0	0	2	0	2
10	0	0	0	0	1	1

Table 5.6: L$*$ kernel counts table for grammar 5.4.

During parsing, the KCOUNTS table is used to determine the number of possible parses being pursued at a vertex and therefore the number of parses that must be rejected before the vertex can be deleted. Associated with each vertex is a counter of the number of parses being tracked at that vertex. When a parse is rejected, the counter is decremented by one until it reaches zero, at which time the vertex is deleted.

There is also a problem that, because vertices may not be deleted when a parse is rejected, combine actions may be performed when they should not be because there are still pointers to the rejected vertices elsewhere in the stack. The parse associated with this vertex has been rejected, so references to it stored in combine pointer elsewhere in the stack should be deleted. To be able to locate these combine pointers, a reverse *kill pointer* is maintained at each vertex. A kill pointer points to the combine pointer created by the eager reduction that created the vertex now being deleted. Thus when a vertex is rejected, the parser follows the kill pointer to identify the combine pointer from which reference to the deleted vertex is then removed.

If, by this process, the combine pointer is left not pointing to any vertices, then the parse associated with that combine pointer has been rejected, so the counter at the vertex where the empty combine pointer is stored is decremented by one, and the whole procedure of testing the counter and chasing kill pointers is recursively applied at this vertex. If the vertex is not deleted, the empty combine pointer is retained, thus stopping completing reductions from being performed.

The count at a vertex is initialised from the KCOUNTS table, by indexing the table using the current lookahead symbol, or the *EAG* symbol if the vertex is created from an eager reduction.

A full reduction always decrements the count at the vertex, because regardless of whether it is rejected or not, the parser has finished tracking that parse.

Continuing the example of parsing the sentence “A B C D” with grammar 5.4 from figure 5.20, the parser next shifts C onto the stack, creating a new vertex with state 9, and a counter initialised from the entry KCOUNTS[9, D] = 2. The empty combine pointer is moved from the vertex with state 4 to the new vertex, but C is not combined into X_1, which was created by eager reduction, because the combine pointer is empty. Instead, the counter at the new vertex is decremented by one because of the empty combine pointer. The counter is then tested for being zero, in which case the vertex would be deleted. However, the counter is not zero, so the vertex is not deleted, and the stack is left as shown in the first diagram of figure 5.22. Next, the parser performs a full reduction by rule 6 at the vertex with state 9, which creates the node Z_1, and the new stack vertex with state 5. Also, the counter at the vertex with state 9 is decremented, making it 0. Thus the vertex with state 9 is deleted, and the outstanding action `s10` is never processed. This is because the only possible use for this action is to extend the parse by rule 4, which has already been rejected. The resulting stack is shown in the second diagram of figure 5.22.

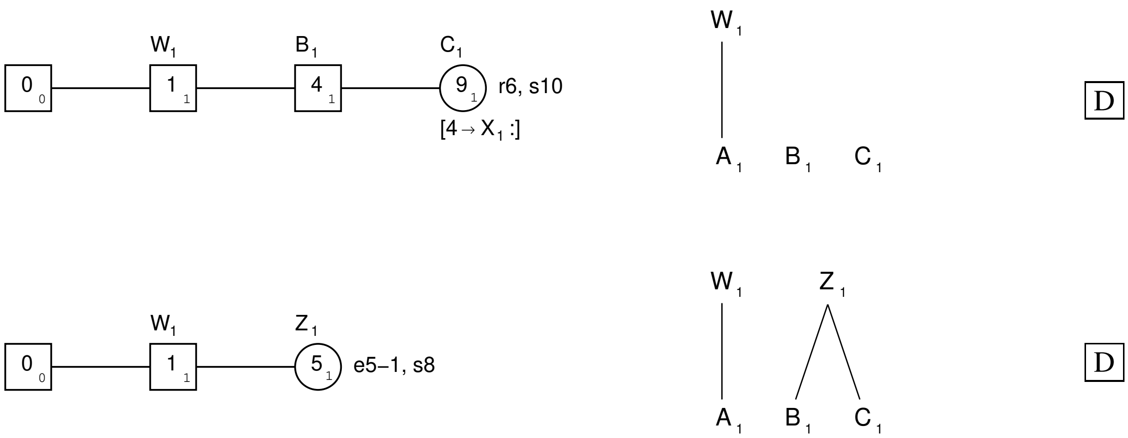


Figure 5.22: Trace of the L∗ algorithm parsing “A B C D” with grammar 5.4 (cont.).

5.3 An example of using the extended L$*$ parser

Consider parsing the sentence "The courses taught at the academy were very demanding" with grammar 5.5 using the extended L$*$ algorithm, which includes equivalence classes and an interface to an external oracle.

$$
\begin{array}{lllll}
(1) & \text{S} & \rightarrow & \text{NP VP}_h \\
(2) & \text{RCl} & \rightarrow & \text{VP} \\
(3) & \text{NG} & \rightarrow & \text{Det N}_h \\
(4) & \text{NP} & \rightarrow & \text{NG}_h \\
(5) & \text{NP} & \rightarrow & \text{NG}_h\ \text{RCl} \\
(6) & \text{VP} & \rightarrow & \text{V}_h\ \text{PP} \\
(7) & \text{VP} & \rightarrow & \text{V}_h\ \text{Adv Adj} \\
(8) & \text{PP} & \rightarrow & \text{Prep}_h\ \text{NG}
\end{array}
\qquad \text{Grammar 5.5}
$$

To illustrate the L$*$ algorithm in detail, the state of the parser at different stages of the example is shown in a diagram with three components:

- *The graph-structured stack.*

  The graph-structured stack is drawn as for the diagrams of chapter 4, except that the number inside a vertex is the equivalence class of the vertex instead of the state. There is also a second number in the lower right corner of each vertex which is the counter of the number of live parses the vertex represents.

- *The parse forest.*

  The parse forest is drawn as for the diagrams of chapter 4. Parts of the parse forest that are rejected are displayed in grey.

- *The current word.*

  As in chapter 4, the word the parser is currently processing is shown in a box at the right-hand edge of the diagram.

Table 5.7 shows the L$*$ parse table for grammar 5.5. In addition to the the information stored in the parse tables constructed in chapter 3, this parse table contains an extra column giving the equivalence class of each state of the parse table, referred to as EQCLASS[st]. Table 5.8 shows the new KCOUNTS table which store the number kernel items included in each state. This is used determine the number of possible parses that a vertex represents.

STATE	EQUIV. CLASS	ACTION													
		Det	N	V	Adv	Adj	Prep	$	*EAG*	S	RCl	VP	NP	NG	PP
0	0	s1								g2			g4	g3	
1	1		s16												
2	2							acc							
3	3			s5, r4					e4-1		g15	g14			
4	4			s5								g6			
5	5				e7-1		e6-1								g9
					s8, c7		s7								c6
6	6							r1	e1-2						
7	7	e8-1												g12	
		s11												c8	
8	8					s10, c7									
9	9			r6				r6	e6-2						
10	10			r7				r7							
11	1		s13												
12	11			r8				r8	e8-2						
13	12			r3				r3							
14	13			r2					e2-1						
15	14			r5					e5-2						
16	12			r3				r3							

Table 5.7: L$*$ parse table for grammar 5.5.

CLASS	KCOUNTS							
	Det	N	V	Adv	Adj	Prep	$	*EAG*
0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	1
2	0	0	0	0	0	0	1	1
3	0	0	2	0	0	0	0	2
4	0	0	1	0	0	0	0	1
5	0	0	0	1	0	1	0	2
6	0	0	0	0	0	0	1	1
7	1	0	0	0	0	0	0	1
8	0	0	0	0	1	0	0	1
9	0	0	1	0	0	0	1	1
10	0	0	1	0	0	0	1	1
11	0	0	1	0	0	0	1	1
12	0	0	1	0	0	0	1	1
13	0	0	1	0	0	0	0	1
14	0	0	1	0	0	0	0	1

Table 5.8: L$*$ kernel counts table for grammar 5.5.

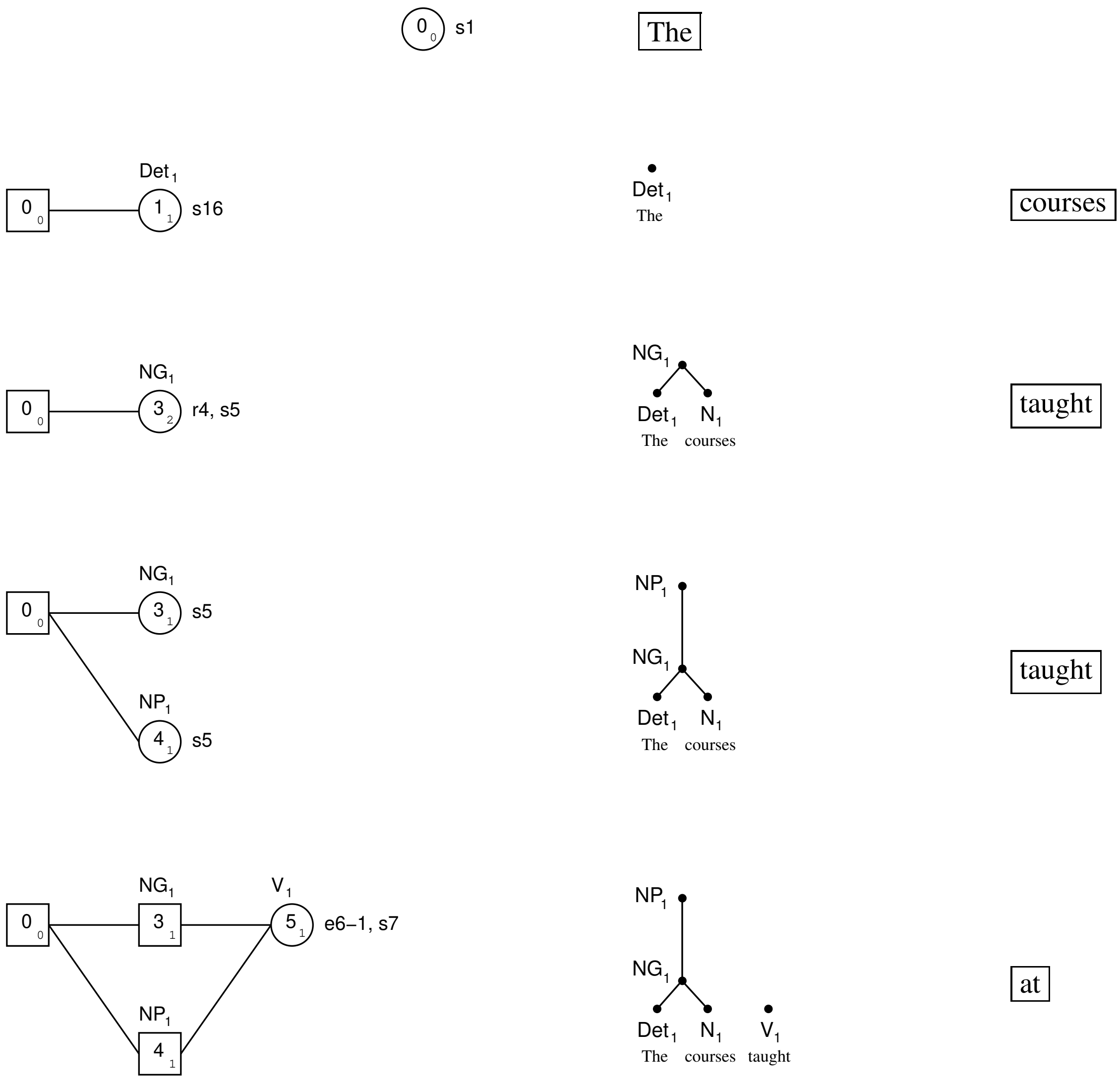


Figure 5.23: Trace of the L∗ algorithm parsing "The courses taught. . . ".

The remainder of this section presents a detailed trace of the L∗ algorithm parsing the sentence "The courses taught at the academy were very demanding". The parser is initialised with a single vertex with state 0 (first diagram of figure 5.23). To begin, the parser shifts the word "The" onto the stack, creating a new vertex that has equivalence class 1 because EQCLASS[1] = 1. The kernel counter of the vertex is initialised from the entry KCOUNTS[1, Det] = 1. The resulting stack is as shown in the second diagram of figure 5.23. Following this, the parser shifts the word "courses" onto the stack and reduces Det_1 and N_1 to NG_1 by rule 3 (third diagram of figure 5.23). This NG is in turn reduced to NP_1 by rule 4. The counter of the vertex with equivalence class 3

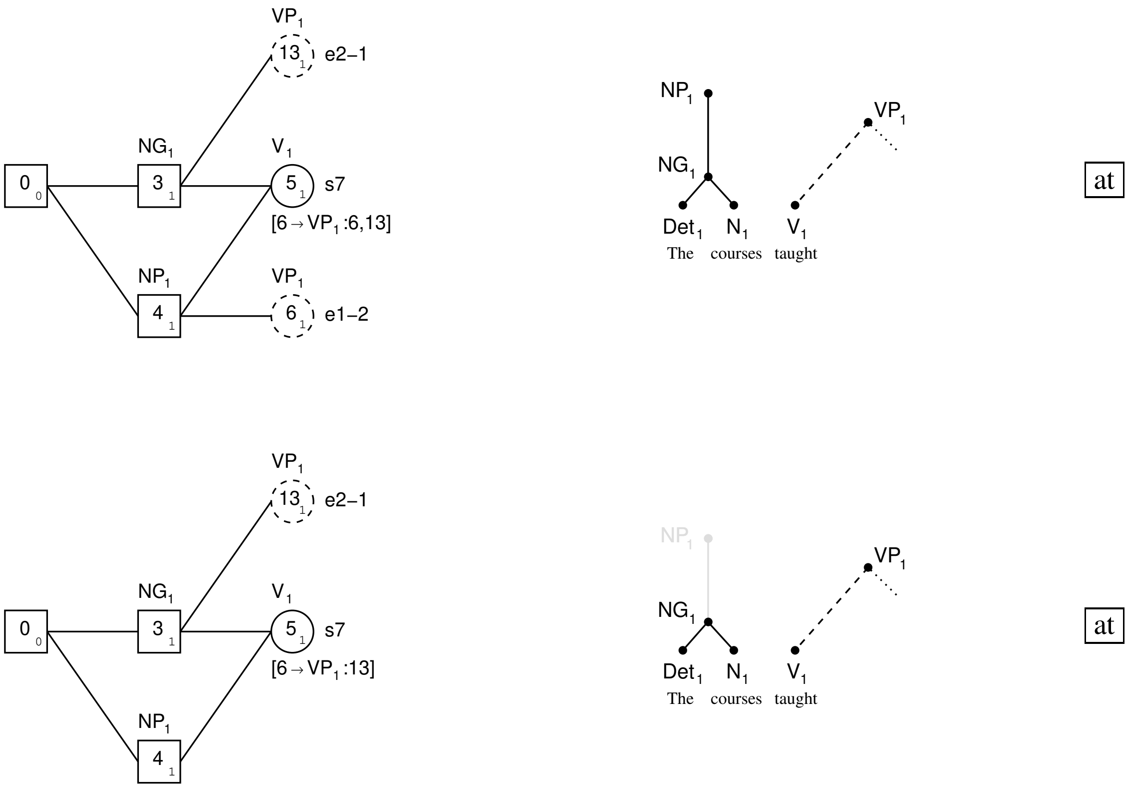


Figure 5.24: Trace of the L∗ algorithm parsing "The courses taught. . . " (cont.).

is decremented because one parse has been processed completely. The stack now has with two branches, both of which have an outstanding `s5` action at the end (fourth diagram of figure 5.23), so the parser rejoins the stack by shifting the word "taught" onto the stack, as shown in the fifth diagram of figure 5.23.

The next word to process is "at". An eager reduction by rule 6 creates the incomplete node VP_1. The eager reduction also establishes a combine pointer at the vertex with equivalence class 5, and a kill pointer that points to this combine pointer at the vertex with equivalence class 6 (first diagram of figure 5.24). At this point, the parser performs an eager reduction by rule 1 at the vertex with equivalence class 6. This reduction attempts to construct an S from NP_1 "the courses" and the incomplete VP_1 "taught . . . ". When this parse of S is passed to the oracle for evaluation, the oracle uses the knowledge that courses cannot teach to reject this parse. Thus the parser stops all further work pursuing this parse. Accordingly, the counter at the vertex with equivalence class 6 is decremented. This makes the counter 0, so the vertex is deleted. The kill pointer at this vertex is used to identify the combine pointer at the vertex with equivalence class 6, and the vertex

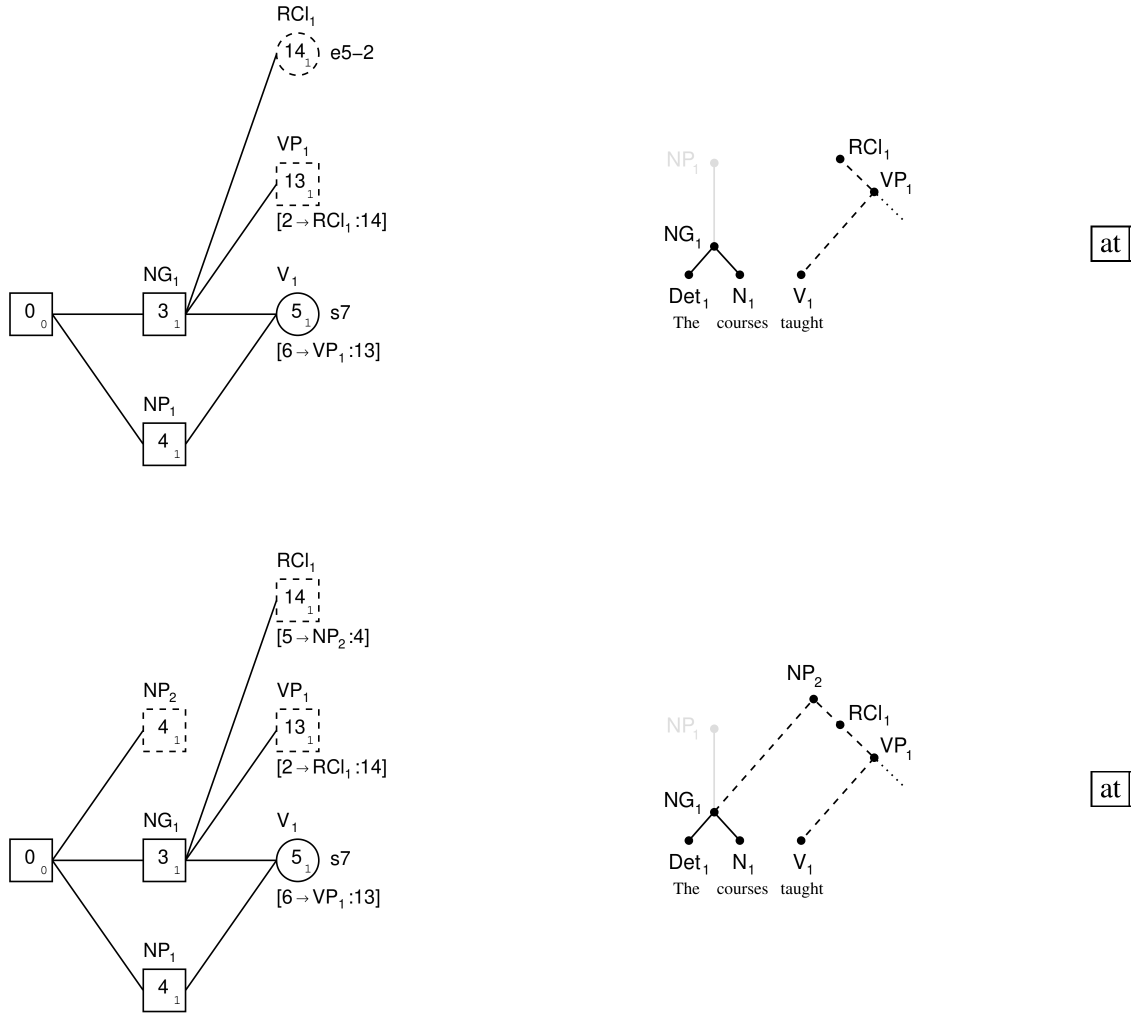


Figure 5.25: Trace of the L∗ algorithm parsing “The courses taught. . . ” (cont.).

with equivalence class 6 is removed from the combine pointer (second diagram of figure 5.24). The eager reduction of V_1 to VP_1 is not yet rejected, however, because there is one other place it may be used, represented by the vertex with equivalence class 13 stored in the combine pointer, at which there is an unprocessed eager reduction by rule 2. This reduction is processed next by the parser, creating the incomplete node RCl_1 from VP_1 (first diagram of figure 5.25). A cascaded eager reduction by rule 5 then creates NP_2 from NG_1 and RCl_1, leaving the stack as shown in the second diagram of figure 5.25.

The parser shifts the word “at” onto the stack and eagerly reduces it to PP_1 by rule 8. This incomplete PP is combined into the previously created node VP_1 (first diagram of figure 5.26). Next, the words “the academy” are shifted onto the stack, and reduced to an NG by rule 3. This

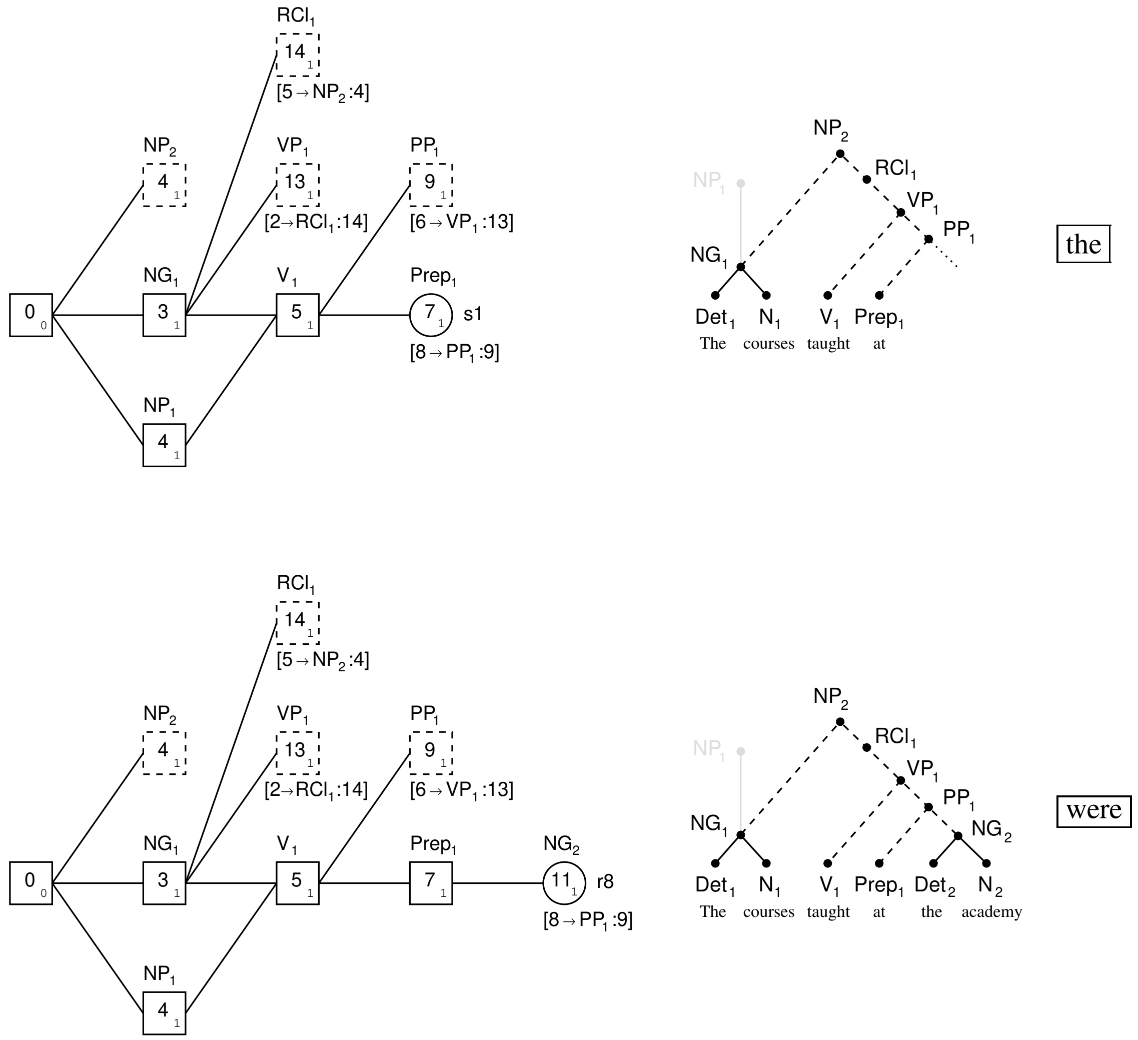


Figure 5.26: Trace of the L∗ algorithm parsing "The courses taught. . ." (cont.).

NG is combined into the incomplete PP_1, leaving the stack as shown in the second diagram of figure 5.26. Next, the parser performs a completing reduction by rule 8, marking PP_1 as complete (first diagram in figure 5.27). This is followed by a completing reduction by rule 6, marking VP_1 as complete. Note how, although there are two paths connected to the vertex with equivalence class 5, the completing reduction only gets propagated back along one of them. The other path was rejected in the earlier rejected eager reduction, and the resulting vertex was deleted from the combine pointer. Thus the branch of the stack with NP_1 in it disappears after this reduction is completed, because it cannot participate in any complete parse. This leaves the much simpler looking stack shown in the second diagram of figure 5.27. Next, the parser performs

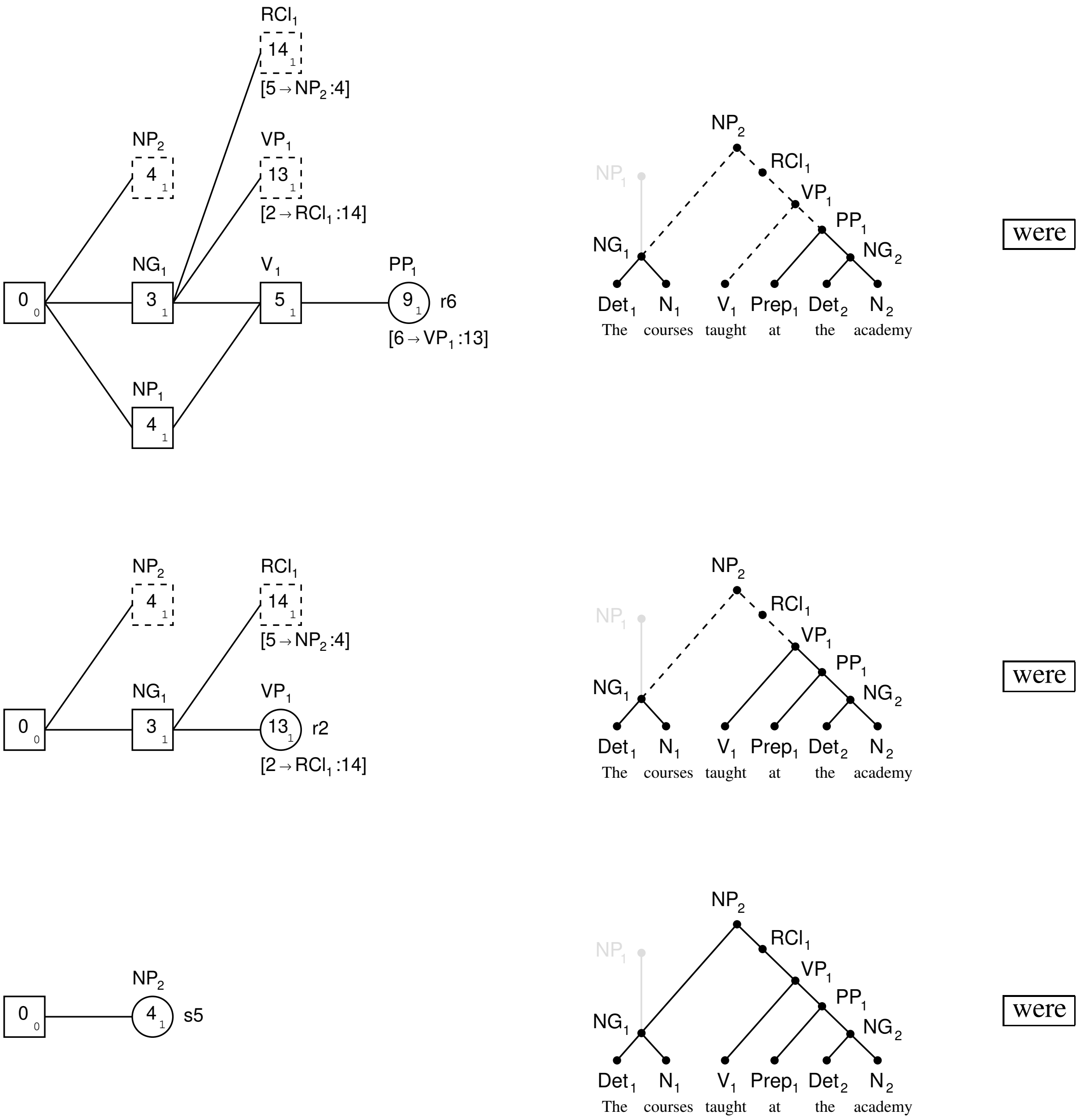


Figure 5.27: Trace of the L∗ algorithm parsing "The courses taught. . ." (cont.).

a completing reduction by rule 2, marking RCl_1 as complete, which is followed by a further completing reduction marking NP_2 as complete. At this point, the stack is as shown in the third diagram of figure 5.27.

The parser completes the parse of the sentence by shifting the word "were" onto the stack (first diagram in figure 5.28), eagerly reducing it VP_2, which is used in conjunction with NP_2 to form S_1 (second diagram in figure 5.28). The parser then shifts the words "very demanding" onto the stack and combines them into the incomplete VP_1. Completing reductions by rule 7 (third

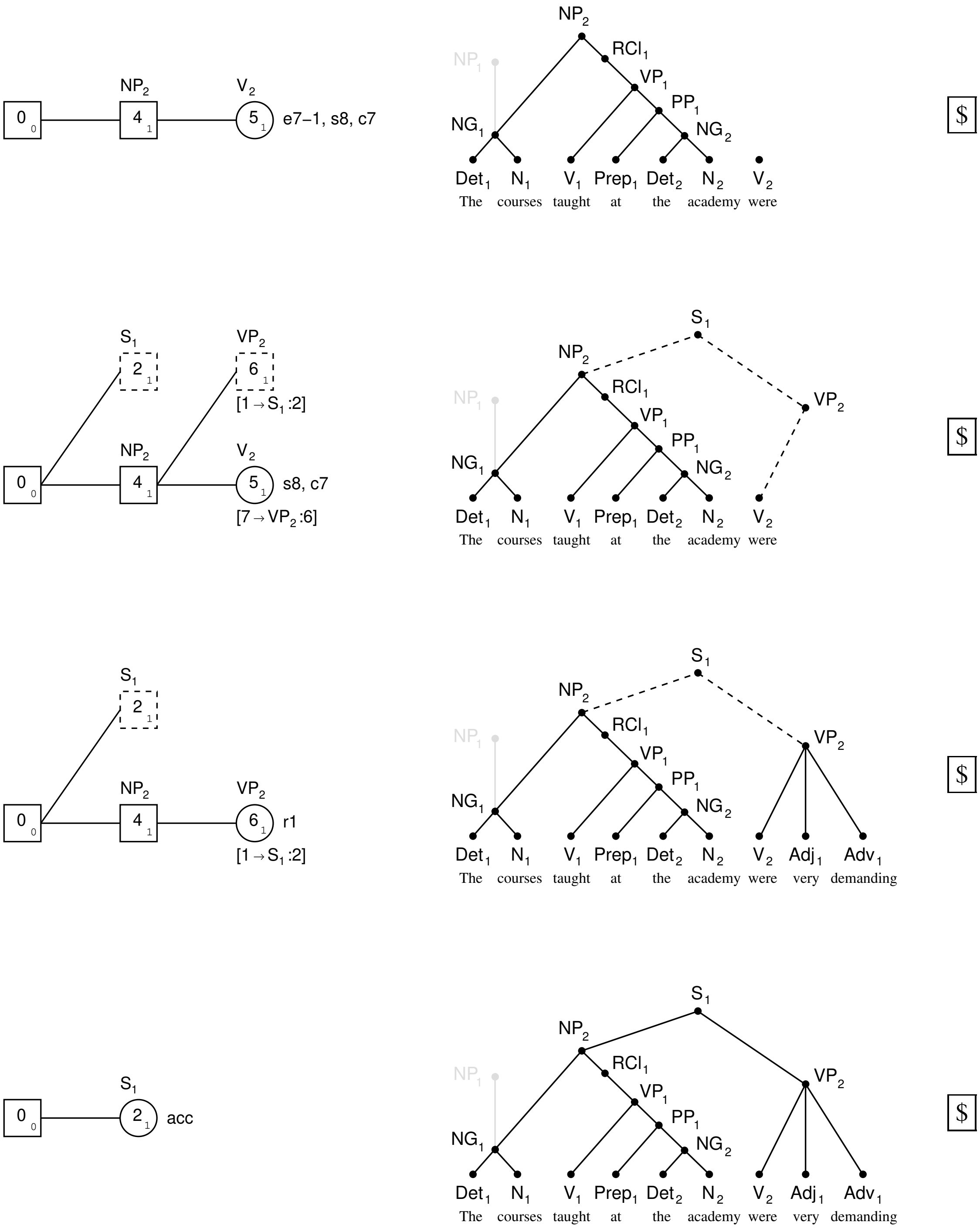


Figure 5.28: Trace of the L∗ algorithm parsing "The courses taught. . . " (cont.).

diagram in figure 5.28) and rule 1 make S_1 complete , and the parser finishes the parse with S_1 as the root of the parse forest (fourth diagram in figure 5.28).

5.4 Formal Algorithm

Input

A parse table ACTION[*state*, *symbol*] for the context-free grammar $G = \langle N, T, R, S\rangle$ and an input string $z \in T^*$. Entries in the parse table are sets of parsing actions. Each action has the form "`shift` s", "`reduce` r", "`eager-reduce` r–k", "`goto` s", "`combine` r", or "`accept`". N is a set of nonterminals, T is a set of terminals, R is a set of grammar rules of the form $X \rightarrow \alpha$, where $X \in N$ and $\alpha \in (N \cup T)^+$, and S is the start symbol. The state s_0 is designated as the start state.

A table EQCLASS[*state*] that gives the equivalence class of each state.

A table KCOUNTS[*class*, *symbol*] that gives the number of kernel items of an equivalence class that are valid for a given input symbol.

Output

A list of root nodes of a shared parse forest for z if $z \in L(G)$, otherwise an error indication.

Data Structures

A *vertex* in the graph-structured stack is a tuple $\langle e, \mathcal{S}, n, o, k, \mathcal{C}, \mathcal{P}\rangle$, where e is an equivalence class, $\mathcal{S}$ is a set of states, n is a forest node, o is a count of unprocessed paths, k is a kill pointer, $\mathcal{C}$ is a set of combine pointers, and $\mathcal{P}$ is the set of successor vertices in the stack. For notational convenience, the elements of a vertex v can be referenced using the functions Class(v), States(v), Node(v), KCount(v), Kill-Ptr(v), Combine-Ptrs(v), and Successors(v).

A *node* in the shared forest is a tuple $\langle X, d\rangle$, where $X \in N$, and d is a derivation, written as a list of child forest nodes. The elements of a node n can be referenced using the functions Nonterm(n) and Children(n).

A *combine pointer* is a tuple $\langle r, d, \mathcal{V}\rangle$, where $r \in R$, d is a derivation, and $\mathcal{V}$ is a set of vertices. The elements of a combine pointer c can be referenced using the functions Rule(c), Deriv(c), and Vertices(c).

A *kill pointer* is a pair $\langle v, c\rangle$, where v is a vertex, and c is a combine pointer. The elements of a kill pointer k can be referenced using the functions Kill-Vert(k), and Kill-Cptr(k).

A *path* is a contiguous sequence of vertices $v_1, \ldots, v_k$ in the stack. That is, for $i = 2, \ldots, k$, $v_i \in$ Successors(v_{i-1}).

FRONTIER stores a set of pairs $\langle v, a\rangle$, where v is a vertex and a is a parse action yet to be performed at v. The vertices within the pairs of this list form the active stack tops.

KILL-CHECK is a set of vertices for which the number of unprocessed paths must be checked.

ω denotes the current input symbol.

Main Loop

· Add a terminator symbol \$ to the end of the input string z	
· $\omega \leftarrow$ The first symbol of z	
· $v_0 \leftarrow \langle \text{EQCLASS}[s_0], \{s_0\}, \text{NIL}, 0, \text{NIL}, \{\}, \{\}\rangle$	· Initialise the stack
· Call **Schedule**(v_0, ω)	
· Loop	· Perform reductions followed by shifts until acceptance or rejection.
· Call **Reduce**()	
· Call **Check-Kills**()	
· Call **Shift**()	
· If FRONTIER contains only pairs of the form $\langle v, \text{“}\texttt{accept}\text{”}\rangle$ then halt and return $\{\mathsf{Node}(v) \mid \langle v, \text{“}\texttt{accept}\text{”}\rangle \in \text{FRONTIER}\}$	
· If FRONTIER = $\{\}$ then halt and signal an error.	

Reduce()

Perform all outstanding reductions by calling the subroutine appropriate to each reduce action. Non-eager reduce actions are processed first, followed by eager reduce actions.

- While $\exists x \in$ FRONTIER of the form $\langle v, \text{“}\texttt{reduce}\ X \rightarrow \alpha\text{”}\rangle$: — Process all outstanding non-eager reductions at stack tops.
  - Remove x from FRONTIER
  - $\mathcal{C} \leftarrow \{c \in \textsf{Combine-Ptrs}(v) \mid \textsf{Rule}(c) = X \rightarrow \alpha\}$ — Each element of $\mathcal{C}$ corresponds to a previous eager reduction that is now complete.
  - If $\mathcal{C} \neq \{\}$ then
    - Call **Completing-Reduce**$(v, \mathcal{C})$

    Else
    - $\mathcal{P} \leftarrow \{p \mid p$ is a path of length $|\alpha|$ starting at v in the stack$\}$
    - $\textsf{KCount}(v) \leftarrow \textsf{KCount}(v) + |\mathcal{P}| - 1$
    - $\forall p \in \mathcal{P}$, call **Full-Reduce**$(p, X \rightarrow \alpha)$
- While $\exists x \in$ FRONTIER of the form $\langle v, \text{“}\texttt{eager-reduce}\ r\text{–}k\text{”}\rangle$: — Process all outstanding eager reductions at stack tops.
  - Remove x from FRONTIER
  - $\mathcal{P} \leftarrow \{p \mid p$ is a path of length k starting at v in the stack$\}$
  - $\forall p \in \mathcal{P}$, call **Eager-Reduce**$(p, r)$

Completing-Reduce$(v, \mathcal{C})$

Perform a completing reduction at vertex v. As this reduction covers the same ground as a previous eager reduction, there is no new parse structure to create, and all that needs to be done is schedule any actions at the vertices associated with the result of the completing reduction.

- $\textsf{KCount}(v) \leftarrow \textsf{KCount}(v) - 1$
- Add v to KILL-CHECK
- $\forall c \in \mathcal{C}$ — $\mathcal{C}$ is a set of combine pointers that point to the vertices created earlier by an eager reduction that is now being completed.
  - Remove c from $\textsf{Combine-Ptrs}(v)$
  - $\forall v' \in \textsf{Vertices}(c)$
    - Call **Schedule**(v', ω)

Full-Reduce($v_1, \ldots, v_k, X \rightarrow \alpha$)

Non-eagerly reduce by the rule $X \rightarrow \alpha$, creating a new forest node X whose children are the nodes of vertices $v_1, \ldots, v_k$. Combine this new node into other partial derivations created previously by eager reduction, and schedule any further actions triggered by this reduction.

- KCount(v_1) $\leftarrow$ KCount(v_1) $- 1$
- Add v_1 to KILL-CHECK
- Propose reduction of Node(v_k), ..., Node(v_1) by $X \rightarrow \alpha$ to the Oracle
- If the Oracle does not reject the reduction then
  - $n' \leftarrow \langle X, (\mathsf{Node}(v_k), \ldots, \mathsf{Node}(v_1))\rangle$ — · Create a new forest node.
  - $\mathcal{A} \leftarrow \{\langle v, a\rangle \mid v \in \mathsf{Successors}(v_k) \wedge s \in \mathsf{States}(v) \wedge a \in \text{ACTION}[s, X] \wedge a = \text{“}\texttt{goto}\ s'\text{”}\}$
  - $\Pi \leftarrow$ A partition of $\mathcal{A}$ s.t. $\langle v, a\rangle \in \pi_e$ if and only if $a = \text{“}\texttt{goto}\ s\text{”} \wedge \text{EQCLASS}[s] = e$
  - $\forall \pi_e \in \Pi$ — · Create new vertices containing the new forest node, one for each element of Π.
    - $\mathcal{S} \leftarrow \{s \mid \langle v, a\rangle \in \pi_e \wedge a = \text{“}\texttt{goto}\ s\text{”}\}$
    - $\mathcal{V} \leftarrow \{v \mid \langle v, a\rangle \in \pi_e\}$
    - $o \leftarrow \text{KCOUNTS}[e, \omega]$
    - $v' \leftarrow \langle e, \mathcal{S}, n', o, \text{NIL}, \{\}, \pi_e\rangle$
    - $\forall v \in \mathcal{V}$ — · Combine the newly created n' into previous eager reductions by rule r whose corresponding partial derivations have Node(v) as their rightmost element.
      - $\forall s \in \mathsf{States}(v)$
        - $\forall a \in \text{ACTION}[s, \omega]$ s.t. $a = \text{“}\texttt{combine}\ r\text{”}$
          - Call **Combine**(v, v', r)
    - Call **Schedule**(v', ω)

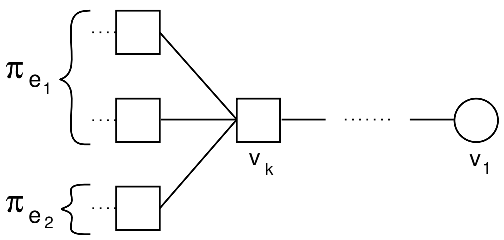


Figure 5.29(a): Stack before full reduction.

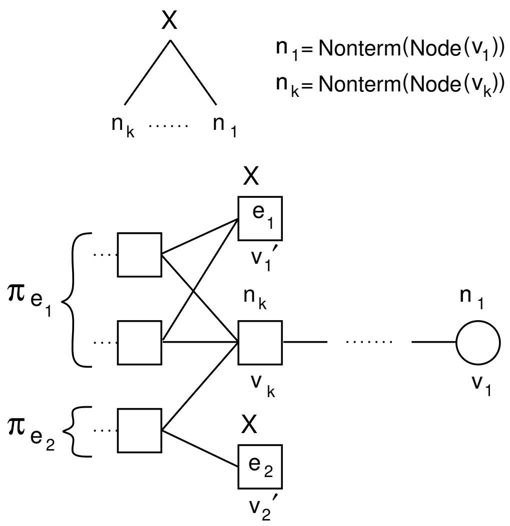


Figure 5.29(b): Stack and new forest node after full reduction.

Eager-Reduce$(v_1, \ldots, v_k, X \rightarrow \alpha)$

Eagerly reduce by the rule $X \rightarrow \alpha$, creating a new incomplete forest node X whose children are the nodes of vertices $v_1, \ldots, v_k$. Combine this new node into other partial derivations created previously by eager reduction, and schedule any further actions triggered by reduction.

· Propose reduction of Node$(v_k), \ldots,$ Node(v_1) by $X \rightarrow \alpha$
  to the Oracle
· If the Oracle rejects the reduction then
  · KCount$(v_1) \leftarrow$ KCount$(v_1) - 1$
  · Add v_1 to KILL-CHECK
  Else
  · $d \leftarrow$ (Node$(v_k), \ldots,$ Node(v_1))
  · $n' \leftarrow \langle X, d \rangle$ — · Create a new forest node.
  · $c \leftarrow \langle X \rightarrow \alpha, d, \{\} \rangle$ — · Create a new combine pointer.
  · $\mathcal{A} \leftarrow \{\langle v, a \rangle \mid v \in$ Successors$(v_k) \wedge s \in$ States$(v) \wedge$
    $a \in$ ACTION$[s, X] \wedge a =$ "goto s'"$\}$
  · $\Pi \leftarrow$ A partition of $\mathcal{A}$ s.t. $\langle v, a \rangle \in \pi_e$ if and only if
    $a =$ "goto s" $\wedge$ EQCLASS$[s] = e$
  · $\forall \pi_e \in \Pi$ — · Create new vertices containing this new node, one for each element of Π.
    · $\mathcal{S} \leftarrow \{s \mid \langle v, a \rangle \in \pi_e \wedge a =$ "goto s"$\}$
    · $\mathcal{V} \leftarrow \{v \mid \langle v, a \rangle \in \pi_e\}$
    · $o \leftarrow$ KCOUNTS$[e, \omega]$
    · $k \leftarrow \langle v, c \rangle$
    · $v' \leftarrow \langle e, \mathcal{S}, n', o, k, \{\}, \mathcal{V} \rangle$
    · Add v' to Vertices(c)
    · $\forall v \in \mathcal{V}$ — · Combine the newly created n' into previous eager reductions by rule r whose corresponding partial derivations have Node(v) as their rightmost element.
      · $\forall s \in$ States(v)
        · $\forall a \in$ ACTION$[s, \omega]$ s.t. $a =$ "combine r"
          · Call **Combine**(v, v', r)
    · Call **Schedule**(v', EAG)
  · Add c to Combine-Ptrs(v_1)

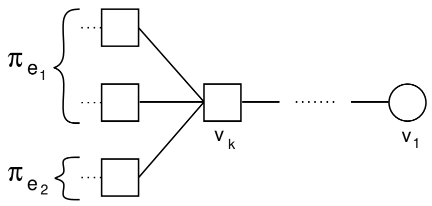


Figure 5.30(a): Stack before eager reduction.

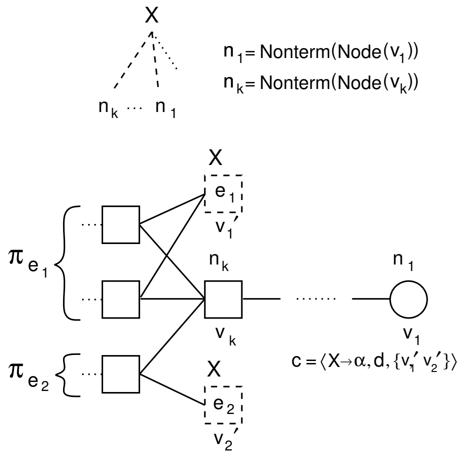


Figure 5.30(b): Stack and new forest node after eager reduction.

Combine(v_f, v_t, r)

Combine Node(v_t) into partial derivations created by eager reduction whose rightmost element is currently Node(v_f). These derivations are identified by the combine pointers associated with v_f.

- $\forall c \in$ Combine-Ptrs(v_f) s.t. Rule(c) = r
  - Remove c from Combine-Ptrs(v_f)
  - If Vertices$(c) = \{\}$ then
    - KCount$(v_t) \leftarrow$ KCount$(v_t) - 1$

    Else
    - Propose combining Node(v_t) into Deriv(c) to the Oracle
    - If the Oracle rejects combine then
      - KCount$(v_t) \leftarrow$ KCount$(v_t) - 1$
      - Vertices$(c) \leftarrow \{\}$
      - Add c to Combine-Ptrs(v_t)

      Else
      - Add Node(v_t) to the end of Deriv(c)
        - All elements of Vertices(c) have the same forest node.
      - Add c to Combine-Ptrs(v_t)
        - Move the combine pointers forward to v_t so that further combines (or completing reductions) can be performed there.

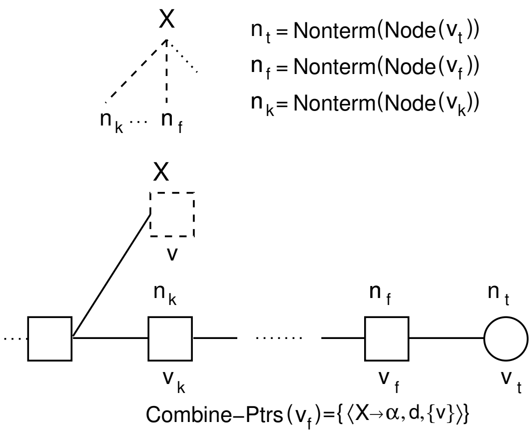


Figure 5.31(a): Stack and forest node before combine.

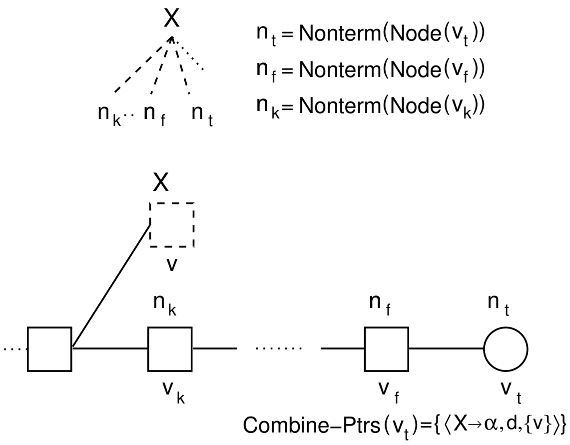


Figure 5.31(b): Stack and forest node after combine.

Check-Kills()

Kills off any actions at dead vertices.

- While $\exists v \in$ KILL-CHECK
  - Remove v from KILL-CHECK
  - If KCount(v) = 0 then
    - Remove any actions of the form $\langle v, a\rangle$ from FRONTIER
    - $k \leftarrow$ Kill-Ptr(v)
    - Delete v from Kill-Cptr(k)
    - If Vertices(Kill-Cptr(k)) = $\{\}$
      - KCount(Kill-Vert(k)) $\leftarrow$ KCount(Kill-Vert(k)) -1
      - Add Kill-Vert(k) to KILL-CHECK

Shift()

Shift the next terminal symbol onto all the stack tops and create a new node for it in the parse forest. Combine this new node into other partial derivations created previously by eager reduction, and schedule any actions triggered by the shift.

- $n \leftarrow \langle \omega, \{\}\rangle$
- $\omega \leftarrow$ The next symbol of the input string
- $\mathcal{A} \leftarrow \{\langle v, a\rangle \in$ FRONTIER $\mid a$ = "`shift` s"$\}$
- FRONTIER $\leftarrow$ FRONTIER $- \mathcal{A}$
- $\Pi \leftarrow$ A partition of $\mathcal{A}$ s.t. $\langle v, a\rangle \in \pi_e$ if and only if
  a = "`shift` s" $\wedge$ EQCLASS[s] = e
- $\forall \pi_e \in \Pi$
  - $\mathcal{S} \leftarrow \{s \mid \langle v, a\rangle \in \pi_e \wedge a$ = "`shift` s"$\}$
  - $\mathcal{V} \leftarrow \{v \mid \langle v, a\rangle \in \pi_e\}$
  - $o \leftarrow$ KCOUNTS[e, ω]
  - $v_e \leftarrow \langle e, \mathcal{S}, n, o, \text{NIL}, \{\}, \mathcal{V}\rangle$
  - $\forall x \in$ FRONTIER of the form $\langle v,$ "`combine` r"$\rangle$ s.t. $v \in \mathcal{V}$
    - Remove x from FRONTIER
    - Call **Combine**(v, v_e, r)
  - Call **Schedule**(v_e, ω)

- Create a new forest node for the shifted input symbol.
- $\mathcal{A}$ is the set of all shift actions to perform.
- Create a new vertex for each member of Π.
- Combine the newly created n into previous eager reductions by rule r whose corresponding partial derivations have Node(v) as their rightmost element.

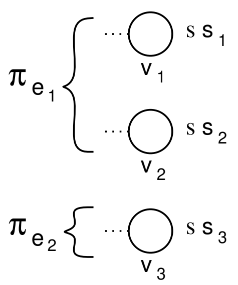


Figure 5.32(a): Stack tops with outstanding shift actions.

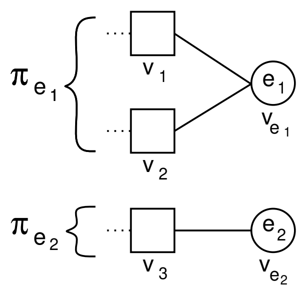


Figure 5.32(b): Stack after shifting.

Schedule(v, L)

Add to FRONTIER all possible actions to be performed at vertex v.

- $\forall s \in \mathsf{States}(v)$
  - $\mathcal{A} \leftarrow \{a \in \text{ACTION}[s, L] \mid$
    $(a =$ "`eager-reduce` r–k" $\Rightarrow$
    $\neg\exists c \in \mathsf{Combine\text{-}Ptrs}(v)$ s.t. $\mathsf{Rule}(c) = r)$
  - $\forall a \in \mathcal{A}$, add $\langle v, a \rangle$ to FRONTIER if not already there

- Schedule all actions except eager reductions that repeat an eager reduction carried out earlier.

□

Chapter 6

Local Ambiguity Packing

The ambiguity of natural language is such that the number of valid parses of a sentence can be exponential in the length of the sentence (Maruyama, 1990), and there are often hundreds of parses for an average sentence in a newspaper (Jacobs et al., 1991). To avoid the cost of processing each parse individually, the GLR algorithm uses *local ambiguity packing*. Local ambiguity packing can reduce the amount of stack and parse forest structure, created during parsing, by an exponential factor, because it allows reuse of computation between parses.

Section 6.1 examines how a GLR parser performs local ambiguity packing. Section 6.2 then describes the modifications of the L$*$ algorithm of chapter 4 that are needed to perform packing. Section 6.3 presents a complete example of parsing using the new L$*$ algorithm. Section 6.4 presents a formal specification of the L$*$ algorithm with packing.

6.1 Determining packing in a GLR parser

It is the job of the parser to determine opportunities to pack. Derivations should be packed when they are headed by the same nonterminal and cover the same input substring. The GLR parser verifies that derivations cover the same input substring by determining that the derivations have the same start and end points in the input. It does this using the configuration of the graph-structured stack. Derivations start at the same point if the vertices resulting from the reductions have the same state and the same successors in the stack. Derivations finish at the same point if they are created by reductions at the same word.

The GLR parser therefore packs any derivations that are created by reductions which meet the following criteria:

1. The reductions all result in the same nonterminal.

2. The reductions are specified at the same input word.

3. The reductions result in vertices in the stack with the same state and the same successors.

For example grammar 2.3 from chapter 2 (reproduced here as grammar 6.1), allows two different parses of S—either as an NP followed by a VP (rule 1) or an S followed by a PP (rule 2).

(1)	S	$\rightarrow$	NP VP	
(2)	S	$\rightarrow$	S PP	
(3)	NP	$\rightarrow$	N	
(4)	NP	$\rightarrow$	Det N	Grammar 6.1
(5)	NP	$\rightarrow$	NP PP	
(6)	PP	$\rightarrow$	Prep NP	
(7)	VP	$\rightarrow$	V NP	

Both forms of the S can be derived from the sentence "John saw a man in the park":

(1) $[[\texttt{John}]_{\text{NP}}\ [\texttt{saw a man in the park}]_{\text{VP}}]_{\text{S}}$

(2) $[[\texttt{John saw a man}]_{\text{S}}\ [\texttt{in the park}]_{\text{PP}}]_{\text{S}}$

These two different derivations of S should be packed together in the same forest node because they cover the same input substring. When parsing this sentence, the parser reaches a state as shown in the first diagram of figure 6.1. The next action the parser executes from this state is the reduction by rule 1, creating a derivation of S from NP_1 and VP_2. The table entry ACTION[0, S] = $\{\texttt{g2}\}$ determines the state of the vertex resulting from the reduction. Thus the parser should create a new vertex with state 2 whose successor is the vertex with state 0. However, such a vertex already exists, so the parser performs local ambiguity packing instead, adding the new derivation of S to S_2, the node associated with the already existing vertex (second diagram in figure 6.1).

6.2 Determining packing in an L$*$ parser

The L$*$ parser uses the method of the GLR parser to decide when full reductions should be packed together. Eager reduction, however, introduces a complication: it becomes difficult to determine

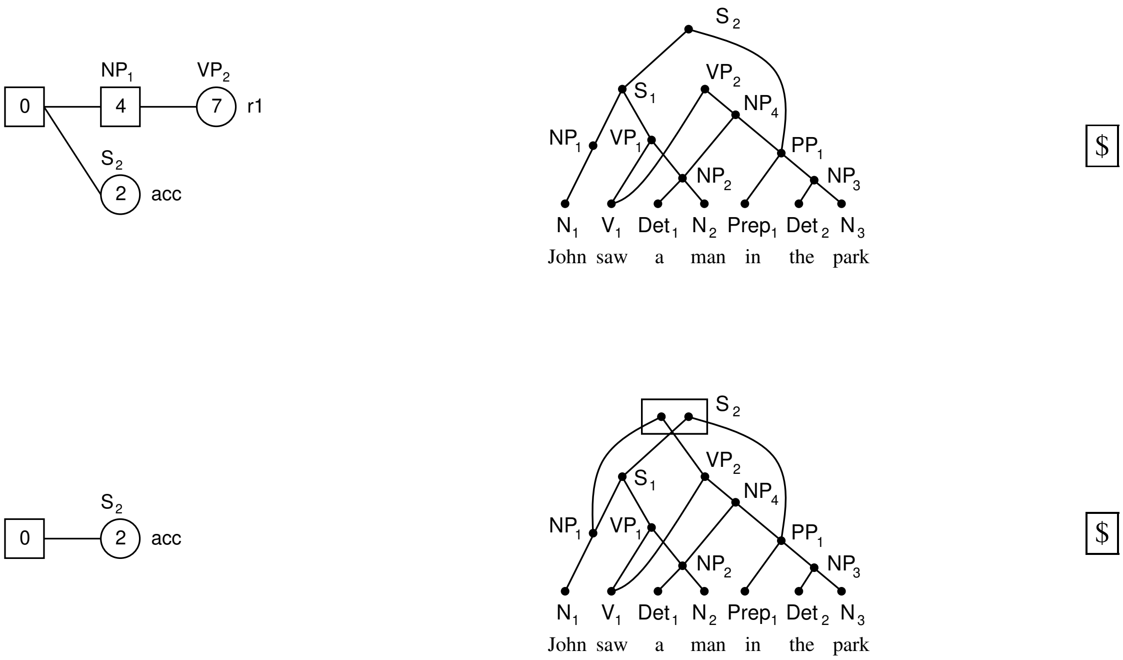


Figure 6.1: State of the GLR parser after processing “John saw a man in the park”.

the input substring that a derivation covers. The starting point of an incomplete derivation created by eager reduction can be determined using the graph-structured stack, as in GLR parsing. However, the word where the eager reduction is performed is not the word where the derivation will be complete. Indeed, the word at which the derivation will be complete is unknown when the eager reduction is performed. It follows that the method used to determine packing opportunities in a GLR parser cannot be applied to derivations created by eager reduction.

The solution of the L$*$ parser is to *provisionally pack* incomplete derivations together if they start at the same input word and derive the same nonterminal. The parser provisionally packs derivations, even though it cannot determine whether the packing will be correct, to avoid unnecessarily repeating work. Whether or not to provisionally pack is also determined using the graph-structured stack. The L$*$ parser packs any incomplete derivations created by eager reductions which meet the following criteria:

1. The reductions all result in the same nonterminal.

2. The reductions result in vertices in the stack with the same state and the same successors.

For example, consider parsing the sentence “A B C” using grammar 6.2. An L$*$ parse table for this grammar is shown in table 6.1. The following notation is employed in diagrams of the parse forest. A provisionally packed node is depicted in diagrams of the parse forest as a box with a dotted line. Individual derivations in a packed node are identified by appending a letter to the name of the forest node containing them. For example, if there are two derivations of a nonterminal X_1, they are labelled $\mathsf{X}_1(\mathsf{a})$ and $\mathsf{X}_1(\mathsf{b})$.

(1)	$\mathrm{S} \rightarrow \mathrm{A}\ \mathrm{X}_h$	
(2)	$\mathrm{X} \rightarrow \mathrm{B}_h\ \mathrm{Y}$	
(3)	$\mathrm{X} \rightarrow \mathrm{B}_h\ \mathrm{Z}$	Grammar 6.2
(4)	$\mathrm{Y} \rightarrow \mathrm{C}$	
(5)	$\mathrm{Z} \rightarrow \mathrm{C}$	
(6)	$\mathrm{Z} \rightarrow \mathrm{C}\ \mathrm{D}$	

STATE	ACTION									
	A	B	C	D	$	*EAG*	S	X	Y	Z
0	s1						g2			
1		s3						g4		
2					acc					
3			e2-1, e3-1						g7, c2	g6, c3
			s5							
4					r1	e1-2				
5				s8	r5, r4					
6					r3	e3-2				
7					r2	e2-2				
8					r6					

Table 6.1: L$*$ parse table for grammar 6.2.

To begin the parse of “A B C”, the parser shifts “A B” onto the stack, leaving the stack as shown in the first diagram of figure 6.2. At the stack top with state 3, there are outstanding eager reductions by rules 2 and 3. The parser first executes the eager reduction by rule 2, creating the incomplete node X_1, as shown in the second diagram of figure 6.2. Next, the parser eagerly reduces by rule 3. This creates another incomplete derivation of X. The table entry ACTION[1, X] = $\{\mathtt{g4}\}$ determines the state of the vertex resulting from the reduction. Thus the parser should create a new vertex with state 4, whose successor is the vertex with state 1. However, there is already an eagerly created vertex in the stack that matches this description, so the parser provisionally packs the new derivation of X into the forest node X_1 (third diagram of figure 6.2). This X is then eagerly reduced to S_1 by rule 1 (fourth diagram of figure 6.2).

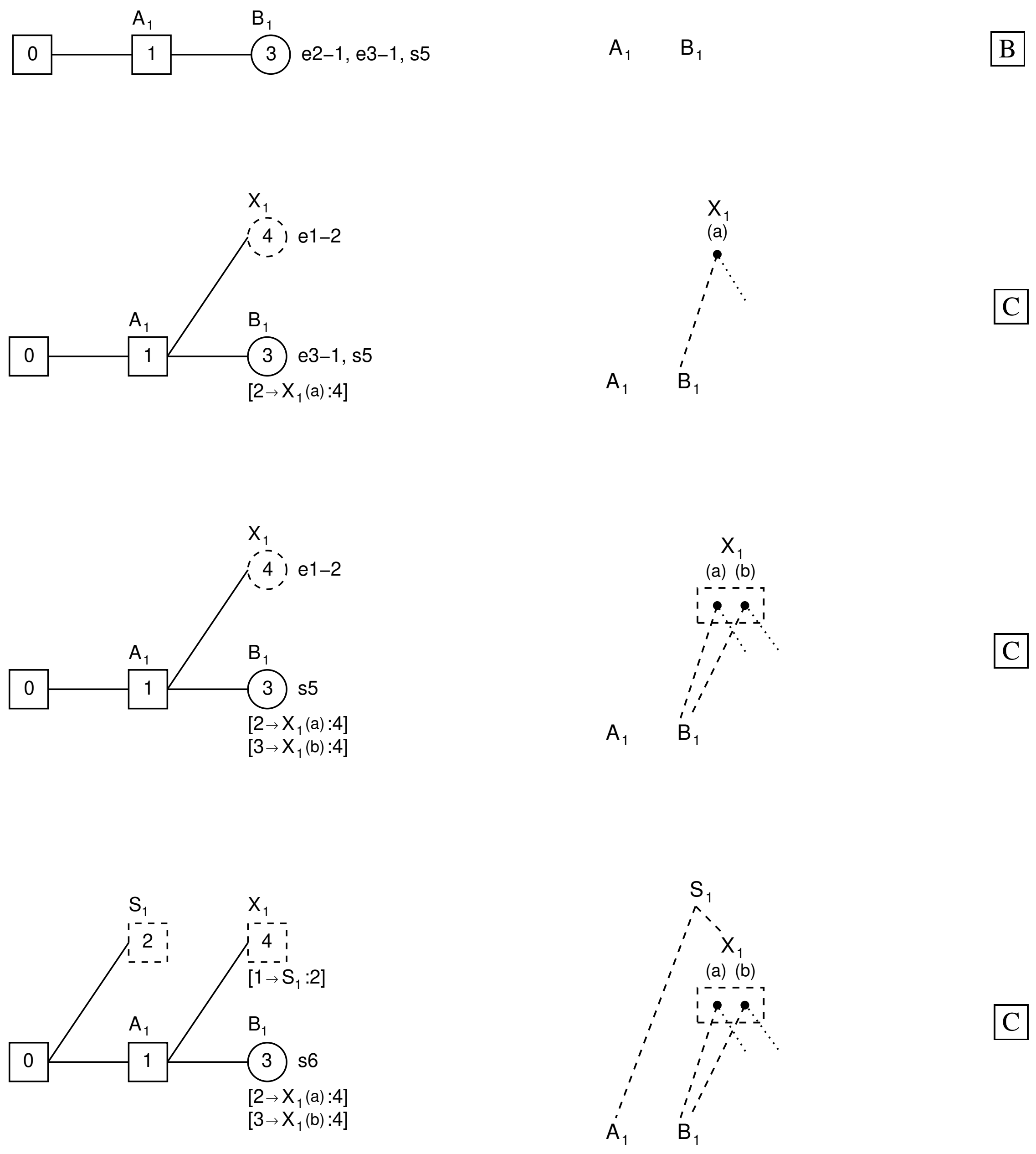


Figure 6.2: Trace of the L∗ algorithm parsing "A B C. . . " with grammar 6.2.

Complete and incomplete derivations are never packed together because they cover different substrings. At a given point in the input, any incomplete derivations created at that point must extend to cover at least the next word, while any complete derivation can extend no further.

6.2.1 Checking provisional packing

The final decision about whether a provisional packing is correct or not can only be made once the derivations of the provisionally packed node are completed. Derivations are marked as complete by completing reductions. When a node contains only one derivation, this is the same as marking the forest node storing the derivation as complete. However, packed nodes are only marked as complete when the provisional packing of the node has been *checked*. The packing of a node should be checked after derivations in the node are completed, and before the next word is shifted onto the stack. Complete derivations should remain packed together, because they must cover the same input substring—they finish at the same point, and they must start at the same point because they were provisionally packed. Incomplete derivations should be separated out of the node because they must extend over a longer substring of the input. These incomplete derivations cannot be deleted, however, because although the provisional packing was incorrect, they may still be completed at a later point in the parse. Instead, the incomplete derivations are put in a newly created forest node. A new vertex is also created in the stack, representing the parses of these incomplete derivations. It has the same state and same successors as the vertex at which the packing check was executed, but it is an eager vertex, because the derivations stored in the node associated with the vertex are incomplete.

The parser specifies that the packing of a node should be checked with the *packing check* action. This is a new action, not stored in the parse table, that is scheduled by the parser at vertices where a completing reduction has completed a derivation of the packed node associated with the vertex. It is written as "`pc`" at stack tops in diagrams of the stack.

For example, again consider grammar 6.2. This grammar accepts the two sentences "A B C" and "A B C D", depending on whether Z is parsed using rule 5 or rule 6. For both these sentences, the parser eagerly reduces by rules 2 and 3 after processing "A B" and provisionally packs the result, as discussed in the previous section, and shown in figure 6.2. This provisional packing is correct only if the sentence is "A B C".

If the input sentence is "A B C", the parser continues processing from the fourth diagram of figure 6.2 by shifting C onto the stack, and reducing it to both a Y by rule 4 and a Z by rule 5 (first diagram of figure 6.3). Next, the parser performs a completing reduction by rule 3, making derivation $X_1(a)$ complete. Because one of the derivations of X_1 has been completed, a packing check action is scheduled at the vertex with state 4 (second diagram of figure 6.3). Before this

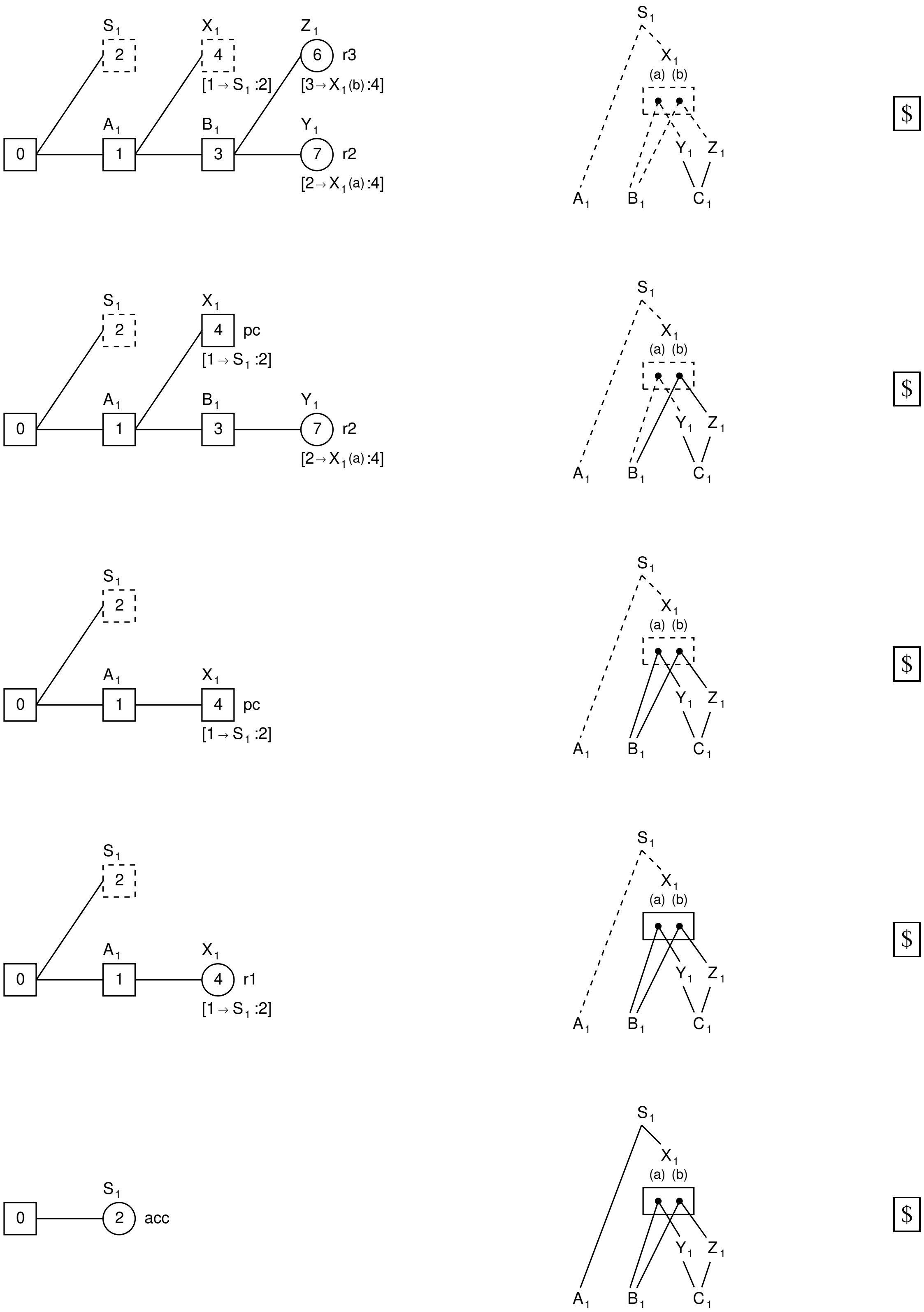


Figure 6.3: Trace of the L∗ algorithm parsing “A B C” with grammar 6.2.

action is executed, however, the parser performs a completing reduction by rule 2, marking the derivation $X_1(b)$ as also complete (third diagram of figure 6.3). The parser now checks the packing of X_1, and because both derivations of the packed node are complete, the parser marks the node X_1 as complete (fourth diagram of figure 6.3). Having verified that the packing of X_1 is correct, the parser performs a completing reduction by rule 1, marking S_1 as complete, thus completing the parse (fifth diagram of figure 6.3).

If the input sentence is "A B C D", the parser continues processing from the fourth diagram of figure 6.2 by shifting C and D onto the stack and reducing them to a Z by rule 6 (as shown in the first diagram of figure 6.4). The parser next performs a completing reduction by rule 3, marking the derivation $X_1(b)$ as complete. Again, a derivation of X_1 has been completed, so the parser schedules a packing check action at the vertex with state 4 (second diagram in figure 6.4). There are no other actions to perform, so the parser executes the packing check of X_1. Derivation $X_1(b)$ is complete, but $X_1(a)$ is not, so the two derivations must be separated. The parser puts the incomplete derivation $X_1(a)$ into a newly created forest node. It also creates a new vertex in the stack whose associated forest node is $X_1(a)$, and whose state and successors are the same as those of the vertex at which the packing check action was performed. The new vertex is an eager vertex because its associated forest node contains only incomplete derivations. This leaves the stack as shown in the third diagram of figure 6.4. Having separated the incomplete derivation into a new forest node, the node $X_1(b)$ can be used as part of a completing reduction by rule 1, marking S_1 as complete (fourth diagram of figure 6.4).

6.2.2 Processing order of parse actions

It is important that the parser performs all completing reductions that might possibly affect a forest node before the packing of the forest node is checked. If the parser does not do this, then it may incorrectly unpack a forest node, because a derivation in the node has not been marked as complete when the packing is checked. For example, in the second diagram of figure 6.3, the parser schedules a packing check action at the vertex with state 4, having performed a completing reduction for $X_1(b)$. There is then a choice between performing the packing check action, or performing the completing reduction by rule 2. If the packing check is performed next, the parser will incorrectly unpack the node X_1, because $X_1(a)$ has not yet been marked as complete by the outstanding completing reduction by rule 2.

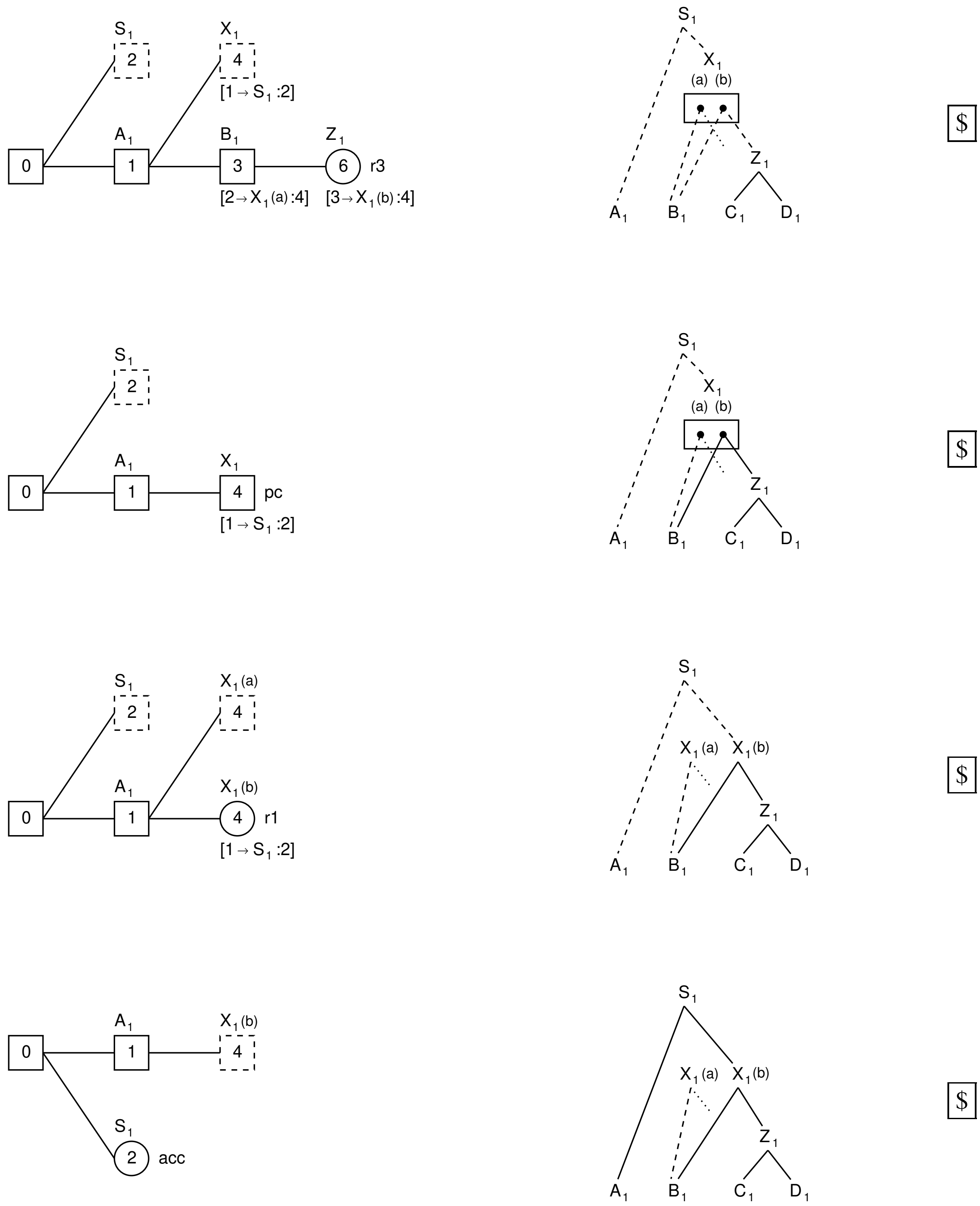


Figure 6.4: Trace of the L∗ algorithm parsing "A B C D" with grammar 6.2.

Completing reductions must also be performed before full reductions. If they are not, derivations created by full reductions at a word will not be packed with derivations that are completed by a completing reduction at the same word.

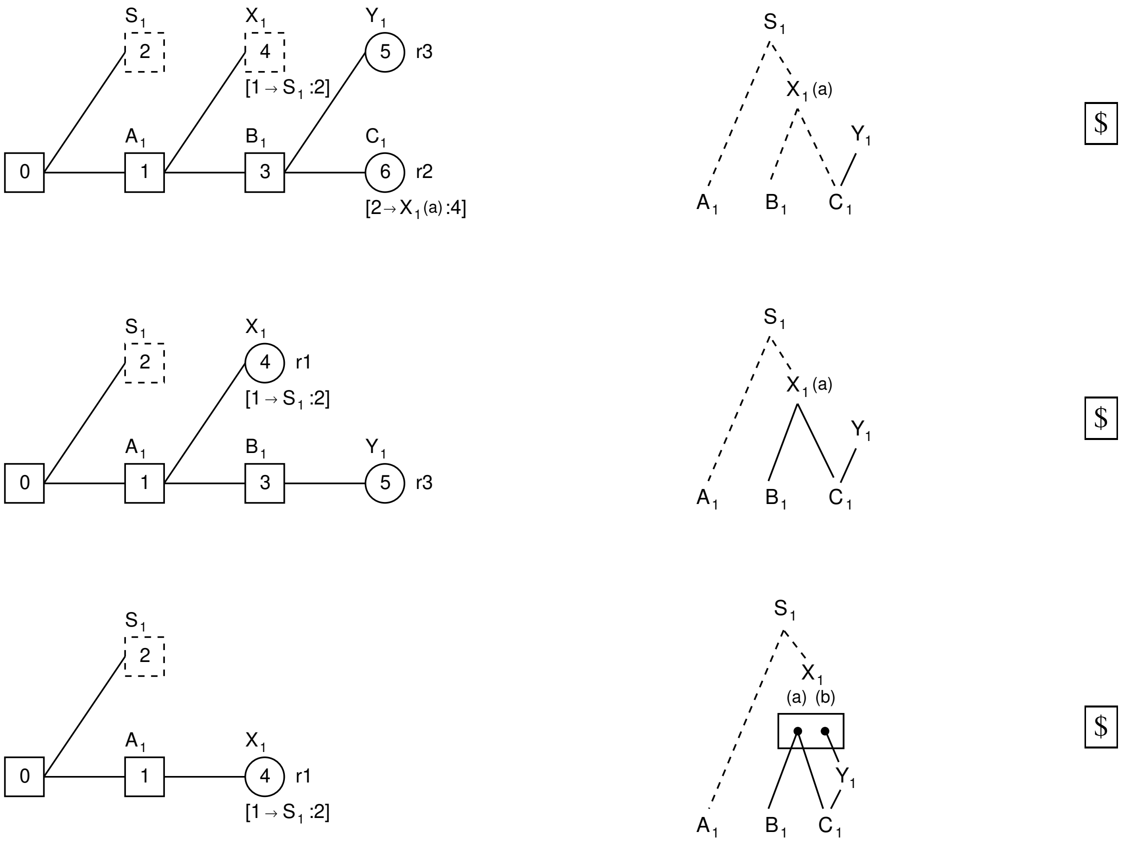


Figure 6.5: Trace of the L∗ algorithm parsing “A B C” with grammar 6.3.

For example, consider parsing the sentence “A B C” using grammar 6.3.

(1)	S	$\rightarrow$	A X_h	
(2)	X	$\rightarrow$	B_h C	Grammar 6.3
(3)	X	$\rightarrow$	B Y_h	
(4)	Y	$\rightarrow$	C_h	

Firstly, the parser shifts “A B” onto the stack. Next, it reduces B to X_1(a) by rule 2, which is then used in a cascaded reduction to S_1 by rule 1. The parser then shifts C onto the stack, combining it into X_1(a), and also reducing it to Y_1 by rule 4. The stack at this point is shown in the first diagram of figure 6.5. If the parser next performs the full reduction by rule 3 at the stack top with state 5, it will incorrectly create a new forest node instead of packing the resulting derivation of X with X_1(a), because X_1(a) is still incomplete. Instead, the parser must first perform the completing reduction by rule 2, marking X_1(a) as complete (second diagram in figure 6.5), so that when the parser performs the full reduction by rule 3, it packs the resulting derivation with the complete X_1(a) (third diagram in figure 6.5).

To ensure correct ordering of reductions, scheduled reductions are ordered according to depth of the vertex that they create in the stack. The *depth* of a vertex is the length of the longest path from the vertex to the root of the stack.

6.3 An example of parsing using the L$*$ parser with packing

To demonstrate the L$*$ algorithm with local ambiguity packing, consider the problem of parsing the sentence "John saw a man in the park" with the grammar 6.4, which is the same as grammar 2.3 used in chapter 2, except the rules of the grammar are annotated with syntactic heads. Table 6.2 shows a parse table for this grammar.

(1)	S	$\rightarrow$	NP VP$_h$	
(2)	S	$\rightarrow$	S$_h$ PP	
(3)	NP	$\rightarrow$	N$_h$	
(4)	NP	$\rightarrow$	Det N$_h$	Grammar 6.4
(5)	NP	$\rightarrow$	NP$_h$ PP	
(6)	PP	$\rightarrow$	Prep$_h$ NP	
(7)	VP	$\rightarrow$	V$_h$ NP	

Throughout the example, the state of the parser is again shown in a diagram with three components:

- *The graph-structured stack.*

  The graph-structured stack is drawn as in the diagrams of chapter 4. There is one new action displayed to the right of a stack top—the packing check action, written `pc`. This action is not stored in the parse table, but is scheduled by the L$*$ parser when necessary.

- *The parse forest.*

  The parse forest is drawn as in the diagrams of chapter 4. Packed nodes are drawn as a box containing the different possible parses for the symbol of the node. A provisional packing of incomplete derivations in a forest node is depicted by the box being drawn with a dotted line. When the provisional packing of a node is checked, the box is changed from a dotted to a solid line. Individual derivations of a packed node are identified by appending a letter to the name of the forest node containing them. For example, two derivations of a packed node X_1 would be labelled X_1(a) and X_1(b).

- *The current word.*

  As in chapter 4, the word the parser is currently processing is shown in a box at the right-hand edge of the diagram.

STATE	ACTION									
	N	Det	Prep	V	$	*EAG*	S	VP	NP	PP
0	s1	s3					g2		g4	
1			r3	r3	r3					
2			s5		acc					g16
3	s15									
4			s5	s6				g8		g7
5	s9, e6-1	s10, e6-1							g14, c6	
6	s9, e7-1	s10, e7-1							g11, c7	
7			r5	r5	r5	e5-2				
8			r1		r1	e1-2				
9			r3	r3	r3					
10	s13									
11			s5, r7		r7	e7-2				g12
12			r5	r5	r5	e5-2				
13			r4	r4	r4					
14			s5, r6	r6	r6	e6-2				g12
15			r4	r4	r4					
16			r2		r2	e2-2				

Table 6.2: L$*$ parse table for grammar 6.4.

The remainder of this section presents a detailed trace of the L$*$ algorithm parsing the sentence "John saw a man in the park". The parser is initialised with a single vertex with state 0 (first diagram of figure 6.6). The parser begins by shifting the word "John" onto the stack, and reducing it to NP_1 by rule 3 (second diagram of figure 6.6). Next, the parser shifts the word "saw" onto the stack, and eagerly reduces it to VP_1 by rule 7. This node is used in a cascaded eager reduction by rule 1, creating the incomplete node S_1 (third diagram of figure 6.6).

Next, the parser shifts the words "a man" onto the stack, and reduces them to NP_2 by rule 4. This NP is combined into the eagerly created node VP_1 (fourth diagram of figure 6.6). The parser then performs a completing reduction by rule 7, which marks the node VP_1 as complete. This is followed by a completing reduction by rule 1, marking S_1 as complete (fifth diagram of figure 6.6).

The parser next rejoins the stack by shifting the word "in" onto the stack, as shown in the first diagram of figure 6.7. Following this, the parser eagerly reduces by rule 6, creating the incomplete node PP_1, and two new vertices—one on each branch of the stack (second diagram of

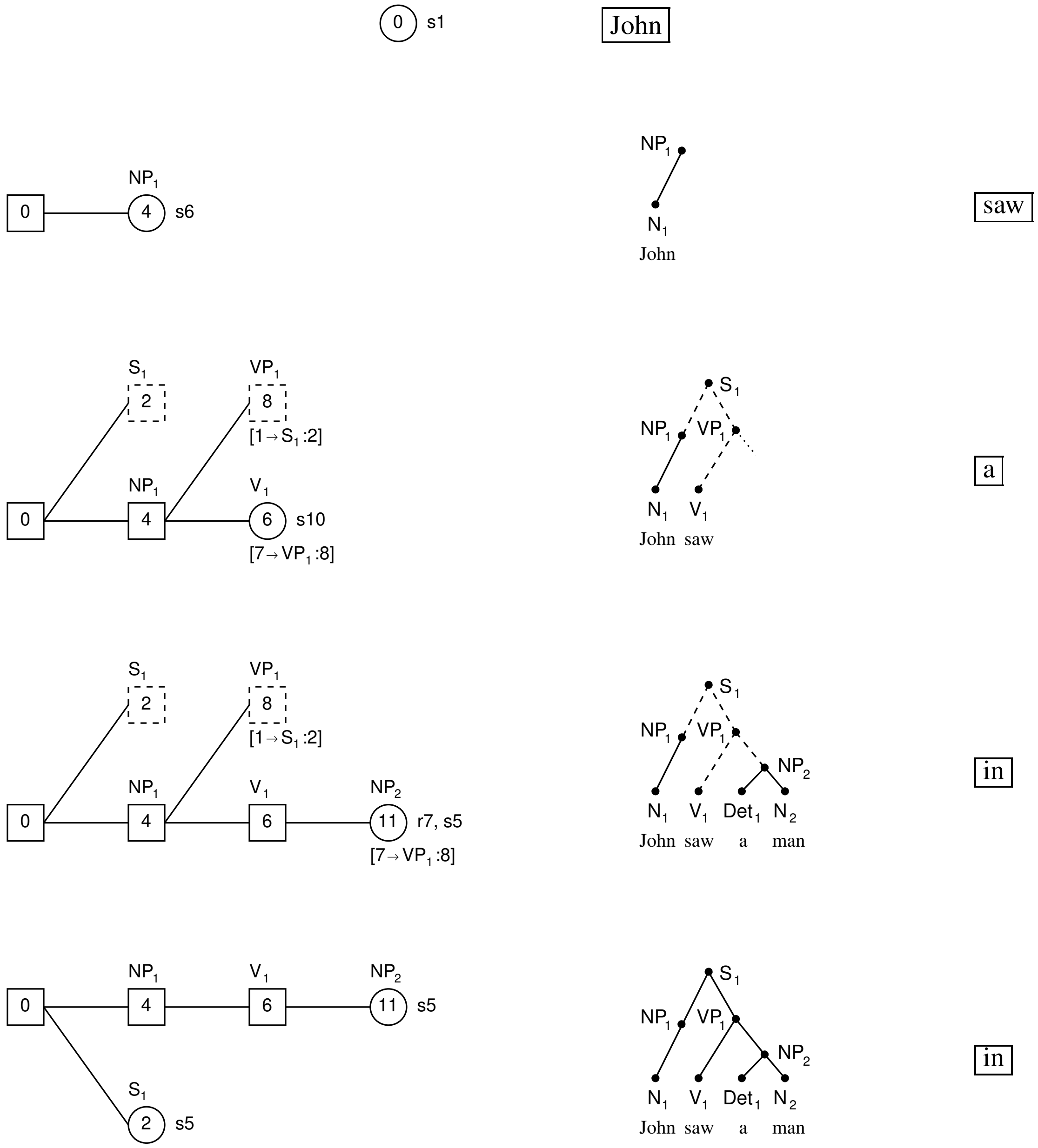


Figure 6.6: Trace of the L∗ algorithm parsing "John saw a man in the park".

figure 6.7). At the vertex with state 12, the parser performs a cascaded eager reduction by rule 5 which creates the node NP_3 from NP_2 and PP_1. This NP is used in a cascaded reduction by rule 7 to create the incomplete node VP_2, which is then used in a further cascaded reduction by rule 1, creating the incomplete node S_2 from NP_1 and VP_2 (third diagram of figure 6.7).

The next action the parser performs is an eager reduction by rule 2, at the stack top with

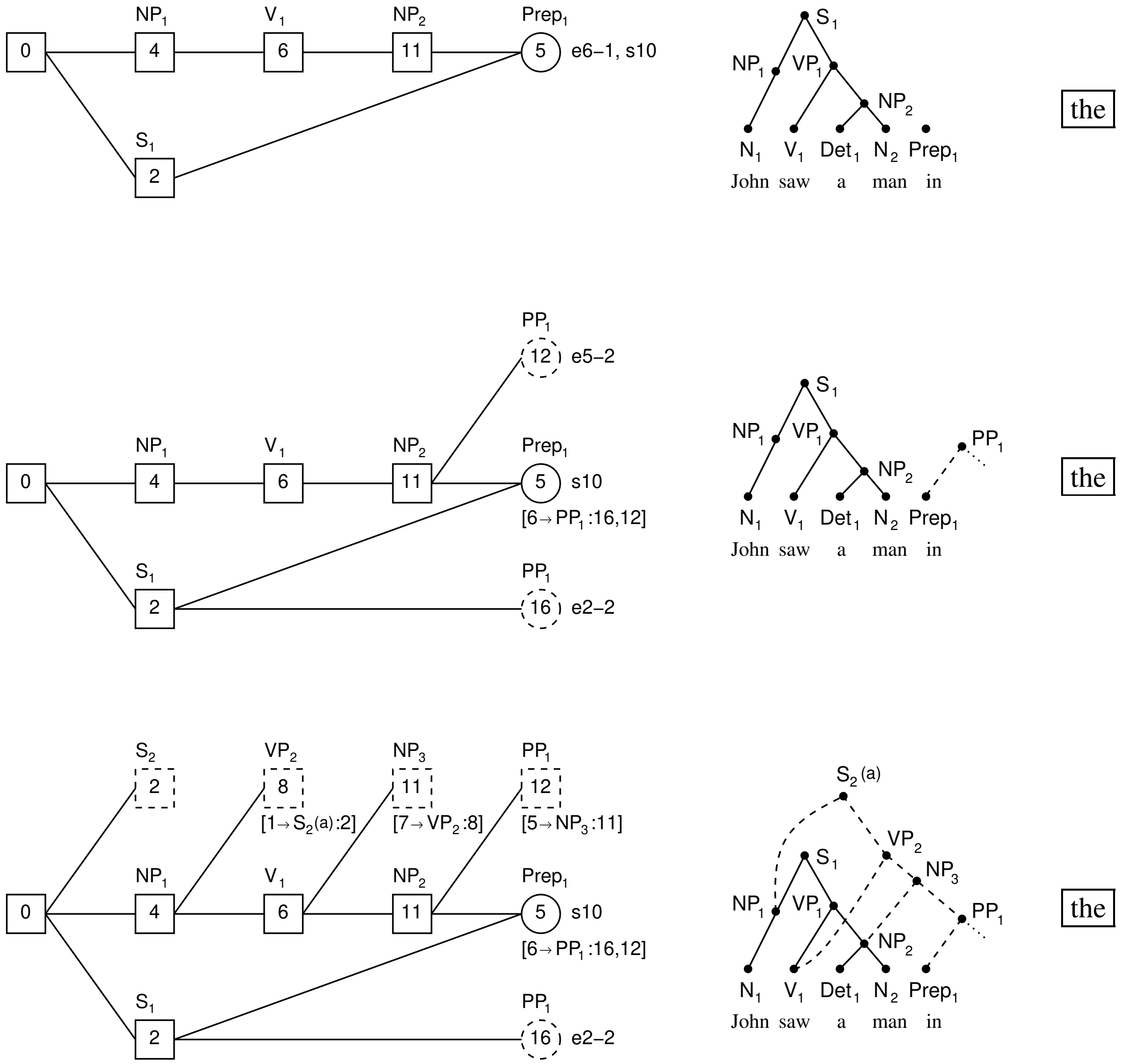


Figure 6.7: Trace of the L$*$ algorithm parsing "John saw a man in the park" (cont.).

state 16. This reduction creates an incomplete derivation of S from S_1 and PP_1. The parse table entry ACTION[0, S] = $\{\mathtt{g2}\}$ specifies that the state of the vertex resulting from the reduction is state 2. There already exists an eager vertex with the same state (state 2) and the same successors (the vertex with state 0) as the one the parser would create from this reduction. Therefore, the parser provisionally packs the new incomplete derivation into the node S_2 (first diagram of figure 6.8).

Having provisionally packed S_2, the parser then shifts the words "the park" onto the stack, and reduces them to NP_4 by rule 4. This NP is combined into PP_1, leaving the stack as shown

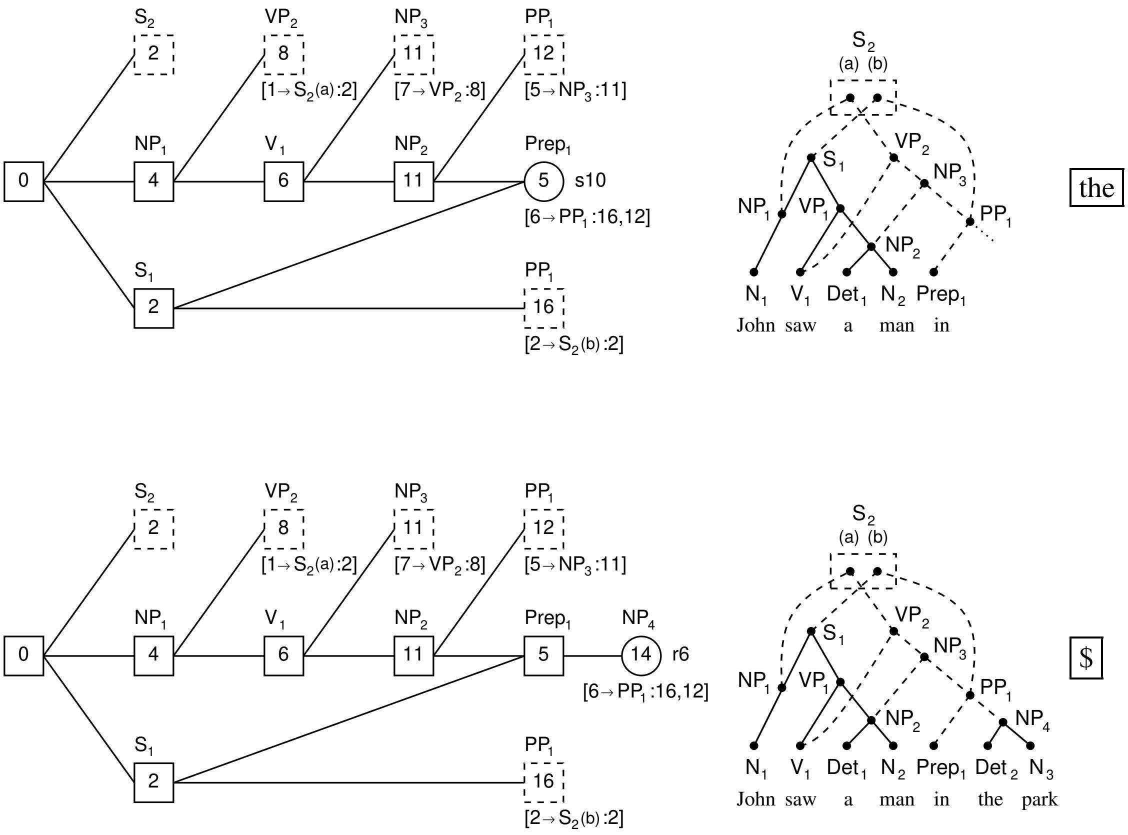


Figure 6.8: Trace of the L$*$ algorithm parsing "John saw a man in the park" (cont.).

in the second diagram of figure 6.8. Next, the parser performs a completing reduction by rule 6, marking PP_1 as complete (first diagram of figure 6.9). This is followed by a completing reduction by rule 5, marking NP_3 as complete, then a further completing reduction by rule 7, marking VP_2 as complete (second diagram of figure 6.9).

Next, the parser performs a completing reduction by rule 1, marking derivation $\mathsf{S}_2(\mathsf{a})$ as complete. This is shown in the first diagram of figure 6.10 by changing the dotted lines connecting the children of $\mathsf{S}_2(\mathsf{a})$ to solid lines. Because one of the derivations of the node S_2 has been completed, the packing of S_2 must be checked, hence a packing check action is scheduled at the stack top with state 2. However, this packing check action is not executed until all other reductions that could affect S_2 have been performed. Thus the completing reduction by rule 2 at the stack top with state 16 is performed first, marking the derivation $\mathsf{S}_2(\mathsf{b})$ as complete (second diagram of figure 6.10). Now that all reductions that can affect S_2 have been performed, the parser checks

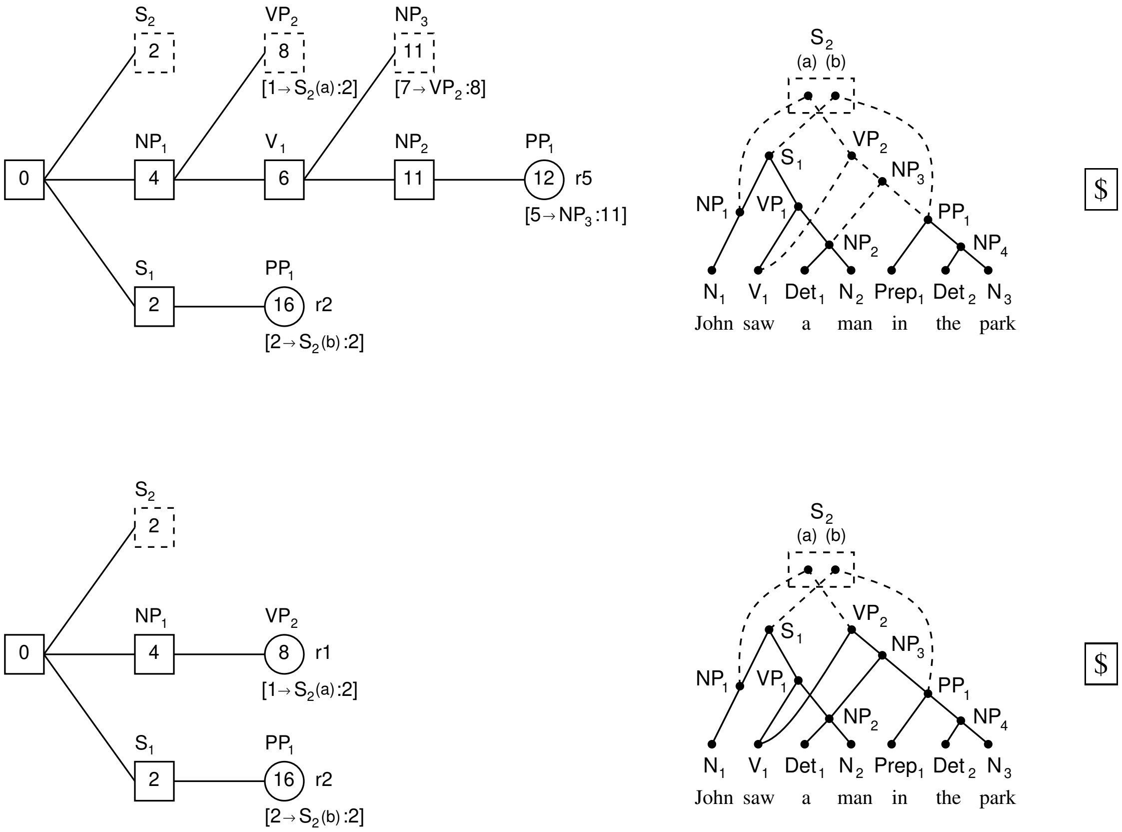


Figure 6.9: Trace of the L$*$ algorithm parsing "John saw a man in the park" (cont.).

the packing of node S_2 and discovers that all derivations of the provisionally packed node are complete, therefore the provisional packing is correct, and S_2 is marked as complete. This is depicted by changing the dotted box to a solid one in the third diagram of figure 6.10. Having determined that the packing of S_2 is correct, the parser accepts the input, with S_2 as the root of the resulting parse forest.

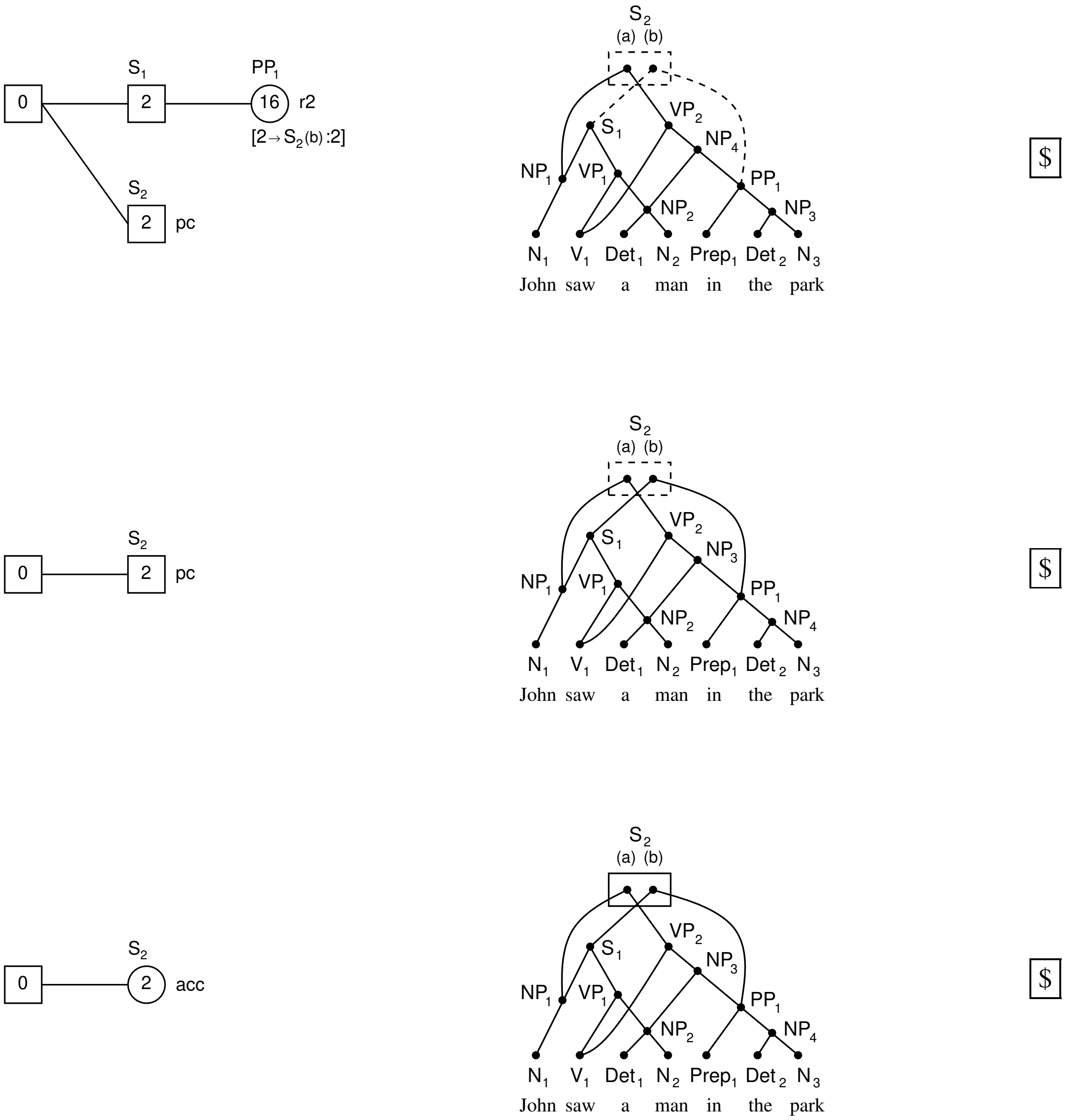


Figure 6.10: Trace of the L∗ algorithm parsing "John saw a man in the park" (cont.).

6.4 Formal Algorithm

Input

A parse table ACTION[*state*, *symbol*] for the context-free grammar $G = \langle N, T, R, S\rangle$ and an input string $z \in T^*$. Entries in the parse table are sets of parsing actions. Each action has the form "`shift` s", "`reduce` r", "`eager-reduce` r–k", "`goto` s", "`combine` r", or "`accept`". N is a set of nonterminals, T is a set of terminals, R is a set of grammar rules of the form $X \rightarrow \alpha$, where $X \in N$ and $\alpha \in (N \cup T)^+$, and S is the start symbol. The state s_0 is designated as the start state.

Output

The root node of a packed shared parse tree for z if $z \in L(G)$, otherwise an error indication.

Data Structures

A *vertex* in the graph-structured stack is a tuple $\langle s, n, l, \mathcal{C}, \mathcal{S}\rangle$, where s is a state, n is a forest node, l is the length of the longest path back to the base of the stack, $\mathcal{C}$ is a set of combine pointers, and $\mathcal{S}$ is the set of successor vertices in the stack. For notational convenience, the elements of a vertex v can be referenced using the functions State(v), Node(v), Depth(v), Combine-Ptrs(v), and Successors(v).

A *node* in the shared forest is a tuple $\langle X, \mathcal{D}\rangle$, where $X \in N$, and $\mathcal{D}$ is a set of derivations. The elements of a node n can be referenced using the functions Nonterm(n) and Derivs(n).

A *derivation* is a pair $\langle c, x\rangle$, where c is a list of child forest nodes and $x \in \{$`complete`, `incomplete`$\}$ indicates whether the derivation is complete or not. The elements of a derivation d can be referenced using the functions Children(d), and Status(d).

A *combine pointer* is a tuple $\langle r, d, \mathcal{V}\rangle$, where $r \in R$, d is a derivation and $\mathcal{V}$ is a set of vertices. The elements of a combine pointer c can be referenced using the functions Rule(c), Deriv(c), and Vertices(c).

A *path* is a contiguous sequence of vertices $v_1, \ldots, v_k$ in the stack. That is, for $i = 2, \ldots, k$, $v_i \in$ Successors(v_{i-1}).

RFRONTIER and SFRONTIER store sets of pairs $\langle v, a\rangle$, where v is a vertex and a is a parse action yet to be performed at v. The vertices of these pairs form the active stack tops. `Shift`, `combine`, and `accept` actions are stored in SFRONTIER. `Reduce`, `eager-reduce`, and `pack-check` actions are stored in RFRONTIER. RFRONTIER is a priority queue. Items on

RFRONTIER are ordered by the depth of the vertices that their actions create in the stack. Actions that create vertices deeper in the stack have a higher priority and are run earlier.

The sets CURRENT-V and EAGER-V are used to keep track of possible packing opportunities. CURRENT-V is a set of all vertices whose associated forest node contains at least one derivation that was completed at the current input word. EAGER-V is a set of all vertices created by eager reduction whose associated forest node as yet contains no complete derivation.

ω denotes the current input symbol.

Main Loop

· Add a terminator symbol \$ to the end of the input string z	
· $\omega \leftarrow$ The first symbol of z	
· $v_0 \leftarrow \langle s_0, \text{NIL}, 0, \{\}, \{\}\rangle$	· Initialise the stack
· CURRENT-V $\leftarrow \{v_0\}$	
· EAGER-V $\leftarrow \{\}$	
· Call **Schedule**(v_0, ω)	
· Loop	· Perform reductions followed by shifts until acceptance or rejection.
· Call **Reduce**()	
· Call **Shift**()	
· If SFRONTIER = $\{\langle v, \text{“}\texttt{accept}\text{”}\rangle\}$ then halt and return Node(v)	
· If SFRONTIER = $\{\}$ then halt and signal an error.	

Reduce()

Perform all outstanding reductions by calling the subroutine appropriate to each reduce action. Actions with the same priority are processed in the following order: completing reduces, full reduces, packing checks, then eager reduces.

- While RFRONTIER $\neq \{\}$
  - $\mathcal{B} \leftarrow$ The set of all items in RFRONTIER with the highest priority
  - If $\exists x \in \mathcal{B}$ s.t. $x = \langle v,$"`reduce` $X \rightarrow \alpha$"$\rangle$
    - $\mathcal{C} \leftarrow \{c \in \textsf{Combine-Ptrs}(v) \mid \textsf{Rule}(c) = X \rightarrow \alpha\}$ — Each element of $\mathcal{C}$ corresponds to a previous eager reduction that is now complete.
    - If $\mathcal{C} \neq \{\}$
      - Call **Completing-Reduce**$(v, \mathcal{C})$

      Else
      - $\mathcal{P} \leftarrow \{p \mid p$ is a path of length $|\alpha|$ starting at v in the stack$\}$
      - $\forall p \in \mathcal{P}$, call **Full-Reduce**$(p, X \rightarrow \alpha)$

    Else If $\exists x \in \mathcal{B}$ s.t. $x = \langle v,$ "`pack-check`"$\rangle$
    - Call **Packing-Check**$(\{v \mid \langle v,$"`pack-check`"$\rangle \in \mathcal{B}\})$

    Else If $\exists x \in \mathcal{B}$ s.t. $x = \langle v,$"`eager-reduce` r–k"$\rangle$
    - $\mathcal{P} \leftarrow \{p \mid p$ is a path of length k starting at v in the stack$\}$
    - $\forall p \in \mathcal{P}$, call **Eager-Reduce**$(p, r)$

Completing-Reduce$(v, \mathcal{C})$

Perform a completing reduction at vertex v. As this reduction covers the same ground as a previous eager reduction, there is no new parse structure to create, and all that needs to be done is schedule any actions at the vertices associated with the result of the completing reduction.

- $\forall c \in \mathcal{C}$ — $\mathcal{C}$ is a set of combine pointers that point to the vertices created earlier by an eager reduction that is now being completed.
  - $\textsf{Status}(\textsf{Deriv}(c)) \leftarrow$ `complete`
  - Remove c from $\textsf{Combine-Ptrs}(v)$
  - $\forall v' \in \textsf{Vertices}(c)$
    - Remove v' from EAGER-V
    - Add v' to CURRENT-V
    - Add $\langle v',$"`pack-check`"$\rangle$ to RFRONTIER with priority $\textsf{Depth}(v')$ — Check that the packing of $\textsf{Node}(v')$ is still correct.

Full-Reduce$(v_1, \ldots, v_k, X \rightarrow \alpha)$

Non-eagerly reduce by the rule $X \rightarrow \alpha$, creating a derivation of X whose children are the nodes of vertices $v_1, \ldots, v_k$. Combine this new node into other partial derivations created previously by eager reduction, and schedule any further actions triggered by this reduction.

· $d \leftarrow \langle(\mathsf{Node}(v_k), \ldots, \mathsf{Node}(v_1)), \texttt{complete}\rangle$	
· $\Pi \leftarrow$ A partition of Successors(v_k) by goto value on symbol X	· v belongs to the set $\pi_s \in \Pi$ if and only if "`goto` s" $\in$ ACTION[State(v),X]. See figure 6.11(a).
· If $\exists v \in$ CURRENT-V s.t. Successors(v) $\in \Pi \wedge$ Nonterm(Node(v)) = X · Add d to Derivs(Node(v))	· Pack using an existing vertex in CURRENT-V whose nonterminal and left context match the current reduction.
Else · $n' \leftarrow \langle X, \{d\}\rangle$	· Create a new forest node.

· $\forall \pi_s \in \Pi$
    · $l \leftarrow$ Max({Depth(v) | $v \in \pi_s$}) + 1
    · $v' \leftarrow \langle s, n', l, \{\}, \pi_s\rangle$
    · Add v' to CURRENT-V
	· Create new vertices containing this node, one for each element of Π.

· $\forall v \in \pi_s$
      · $\forall a \in$ ACTION[State(v), ω] s.t. a = "`combine` r"
        · Call **Combine**(v, v', r)
    · Call **Schedule**(v', ω)
	· Combine the newly created n' into previous eager reductions by rule r whose corresponding partial derivations have Node(v) as their rightmost element.

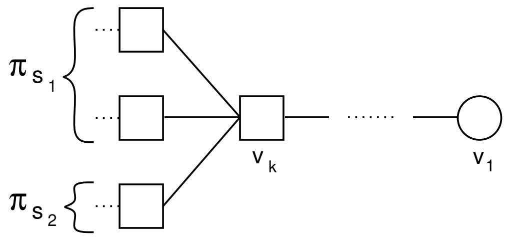


Figure 6.11(a): Stack before full reduction

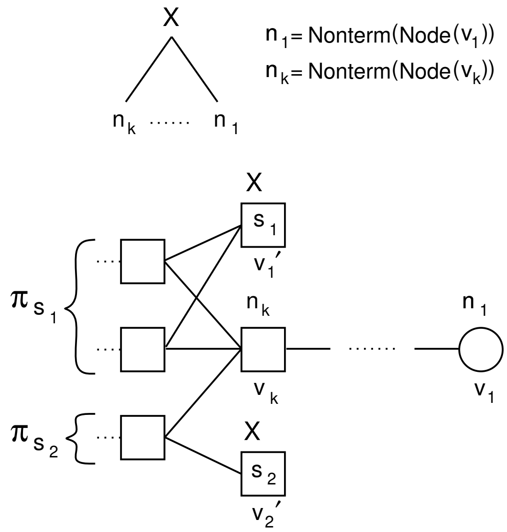


Figure 6.11(b): Stack after full reduction

Eager-Reduce($v_1, \ldots, v_k, X \rightarrow \alpha$)

Eagerly reduce by the rule $X \rightarrow \alpha$, creating a new incomplete derivation of X whose children are the nodes of vertices $v_1, \ldots, v_k$. Combine this new node into other partial derivations created previously by eager reduction, and schedule any further actions triggered by reduction.

· $d \leftarrow \langle(\mathsf{Node}(v_k), \ldots, \mathsf{Node}(v_1)), \texttt{incomplete}\rangle$	· The new derivation is incomplete because it is created by an eager reduction.
· $\Pi \leftarrow$ A partition of Successors(v_k) by goto value on symbol X	· v belongs to the set $\pi_s \in \Pi$ if and only if "goto s" $\in$ ACTION[State(v),X]. See figure 6.12(a).
· $c \leftarrow \langle X \rightarrow \alpha, d, \{\}\rangle$	· Create a new combine pointer.

· $\forall v \in$ EAGER-V
  · If Successors(v) $\in \Pi \wedge$ Nonterm(Node(v)) = X
    · Add v to Vertices(c)
	· Collect the set of existing eager vertices whose nonterminal and left context match the current reduction.

· If Vertices(c) $\neq \{\}$ then	· Pack using an existing vertex created by a previous eager reduction.
· $v \leftarrow$ An arbitrary element of Vertices(c) · Add d to Derivs(Node(v))	· All elements of Vertices(c) have the same forest node.
Else	
· $n' \leftarrow \langle X, \{d\}\rangle$	· Create a new forest node.

· $\forall \pi_s \in \Pi$
    · $l \leftarrow$ Max($\{$Depth(v) $\mid v \in \pi_s\}$) + 1
    · $v' \leftarrow \langle s, n', l, \{\}, \pi_s\rangle$
    · Add v' to EAGER-V
    · Add v' to Vertices(c)
	· Create new vertices containing this new node, one for each element of Π.

· $\forall v \in \pi_s$
      · $\forall a \in$ ACTION[State(v), ω] s.t. a = "combine r"
        · Call **Combine**(v, v', r)
	· Combine the newly created n' into previous eager reductions by rule r whose corresponding partial derivations have Node(v) as their rightmost element.

· Call **Schedule**(v', *EAG*)	· Use *EAG* because appropriate lookahead symbols are not available.
· Add c to Combine-Ptrs(v_1)	

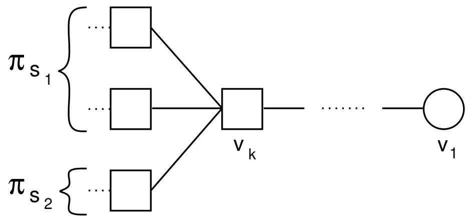


Figure 6.12(a): Stack before eager reduction.

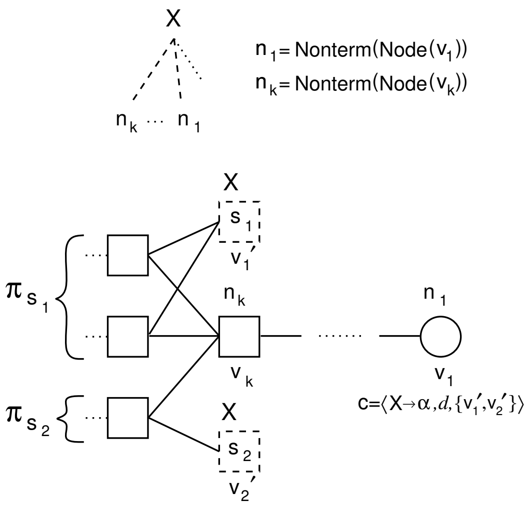


Figure 6.12(b): Stack after eager reduction.

Combine(v_f, v_t, r)

Combine Node(v_t) into partial derivations created by eager reduction whose rightmost element is currently Node(v_f). These derivations are identified by the combine pointers associated with v_f.

- $\forall c \in$ Combine-Ptrs(v_f) s.t. Rule(c) = r
  - Add Node(v_t) to the end of Children(Deriv(c))
  - Remove c from Combine-Ptrs(v_f)
  - Add c to Combine-Ptrs(v_t)

- Move the combine pointers forward to v_t so that further combines (or completing reductions) can be performed there.

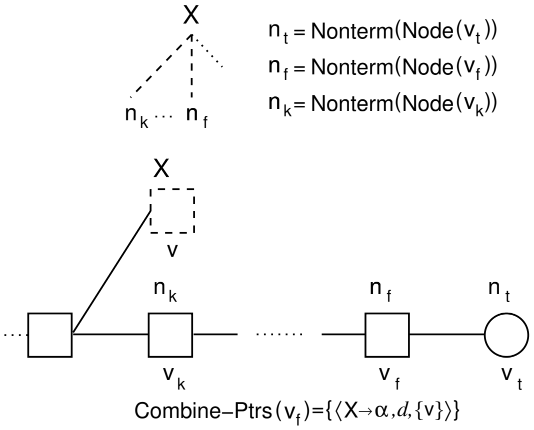


Figure 6.13(a): Stack before combine.

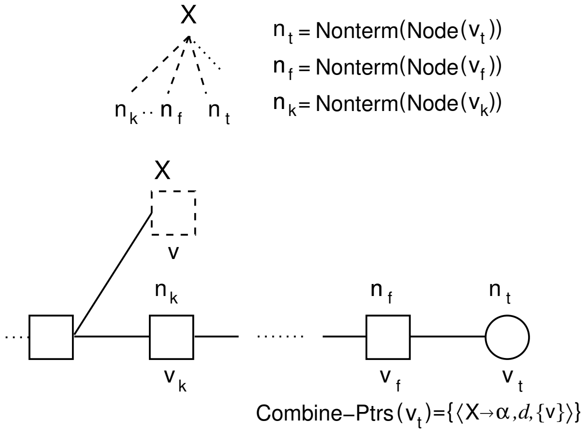


Figure 6.13(b): Stack after combine.

Packing-Check($\mathcal{L}$)

Check the provisional packing of nodes in the parse forest. If there are both complete and incomplete derivations packed together in the same node, then the packing was incorrect and the incomplete and complete derivations must be separated and unpacked into two distinct forest nodes. The complete derivations were created by completing reductions at the current input word, so schedule any actions these completing reductions trigger at the vertices whose associated forest node contain the complete derivations.

· $\Pi \leftarrow$ A partition of $\mathcal{L}$ according to associated forest node	· Each v belongs to the set $\pi_n \in \Pi$ if and only if $\mathsf{Node}(v) = n$.
· $\forall \pi_n \in \Pi$	
· $\mathcal{C} \leftarrow \{d \in \mathsf{Derivs}(n) \mid \mathsf{Status}(d) = \texttt{complete}\}$	
· $\mathcal{I} \leftarrow \{d \in \mathsf{Derivs}(n) \mid \mathsf{Status}(d) = \texttt{incomplete}\}$	
· If $\mathcal{C} \neq \{\}$ then	
· If $\mathcal{I} \neq \{\}$ then	· Forest node n contains both complete and incomplete derivations, so it is necessary to unpack.
· $n' \leftarrow \langle \mathsf{Nonterm}(n), \mathcal{C} \rangle$ · $\mathsf{Derivs}(n) \leftarrow \mathcal{I}$	· Remove the complete derivations from n and put them into a new node n'. See figure 6.14(b).
· $\forall v \in \pi_n$	
· Remove v from CURRENT-V	
· Add v to EAGER-V	
· $v' \leftarrow \langle \mathsf{State}(v), n', \mathsf{Depth}(v), \{\}, \mathsf{Successors}(v) \rangle$	· Create new vertices associated with the newly created node.
· Add v' to CURRENT-V	
· Call **Schedule**(v', ω)	· $\mathsf{Node}(v')$ contains the complete derivations, so schedule further actions at v'
Else	
· $\forall v \in \pi_n$	· All $v \in \pi_n$ have forest node n, and all derivations of n are complete; therefore schedule further actions at these vertices.
· Call **Schedule**(v, ω)	

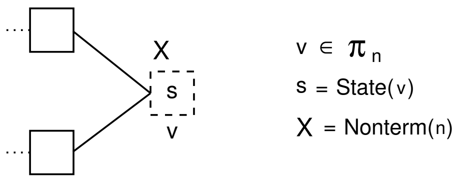


Figure 6.14(a): Stack before unpacking.

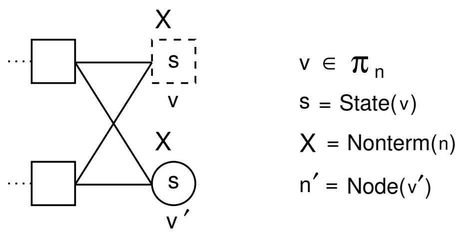


Figure 6.14(b): Stack after unpacking.

Shift()

Shift the next terminal symbol onto all the stack tops and create a new node for it in the parse forest. Combine this new node into other partial derivations created previously by eager reduction, and schedule any actions triggered by the shift.

- $n \leftarrow \langle \omega, \{\} \rangle$
- $\omega \leftarrow$ The next symbol of the input string
- CURRENT-V $\leftarrow \{\}$
- $\mathcal{S} \leftarrow \{\langle v, a\rangle \in$ SFRONTIER $\mid a =$ "`shift` s"$\}$
- SFRONTIER $\leftarrow$ SFRONTIER $- \mathcal{S}$
- $\Pi \leftarrow$ A partition of $\mathcal{S}$ according to goto state of the `shift` actions
- $\forall \pi_s \in \Pi$
  - $\mathcal{V} \leftarrow \{v \mid \langle v, a\rangle \in \pi_s\}$
  - $l \leftarrow$ Max$(\{\mathsf{Depth}(v) \mid v \in \mathcal{V}\}) + 1$
  - $v_s \leftarrow \langle s, n, l, \{\}, \mathcal{V}\rangle$
  - Add v_s to CURRENT-V
  - $\forall x \in$ SFRONTIER s.t. $x = \langle v,$"`combine` r"$\rangle \wedge v \in \mathcal{V}$
    - Remove x from SFRONTIER
    - Call **Combine**(v, v_s, r)
  - Call **Schedule**(v_s, ω)

- Create a new node for the shifted input symbol.
- $\mathcal{S}$ is the set of all shift actions to perform.
- Each $\pi_s \in \Pi$ consists of elements of the form $\langle v,$ "`shift` s"$\rangle$.
- Create a new vertex for each member of Π.
- Combine the newly created n into previous eager reductions by rule r whose corresponding partial derivations have $\mathsf{Node}(v)$ as their rightmost element.

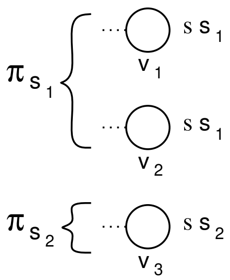


Figure 6.15(a): Stack tops with outstanding shift actions.

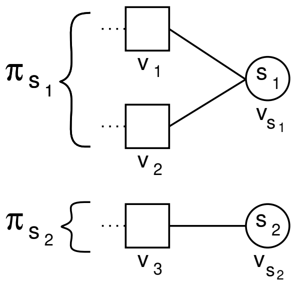


Figure 6.15(b): Stack after shifting.

Schedule(v, L)

Add to RFRONTIER and SFRONTIER all possible actions to be performed at vertex v.

- $\forall a \in$ ACTION[$\mathsf{State}(v), L$] s.t. $a =$ "`eager-reduce` r–k" $\wedge$ $\neg\exists c \in \mathsf{Combine\text{-}Ptrs}(v)$ s.t. $\mathsf{Rule}(c) = r$
  - $p \leftarrow \mathsf{Depth}(v) - k + 1$
  - Add $\langle v, a\rangle$ to RFRONTIER with priority p
- $\forall a \in$ ACTION[$\mathsf{State}(v), L$] s.t. $a =$ "`reduce` $X \rightarrow \alpha$"
  - $p \leftarrow \mathsf{Depth}(v) - |\alpha| + 1$
  - Add $\langle v, a\rangle$ to RFRONTIER with priority p
- $\forall a \in$ ACTION[$\mathsf{State}(v), L$] s.t. a is a `shift`, `combine`, or `accept` action
  - Add $\langle v, a\rangle$ to SFRONTIER

- Filter out reductions that would repeat an eager reduction carried out earlier.

□

Chapter 7

Conclusion

This report has described L$*$ parsing—a method of parsing designed to improve the efficiency of a natural language processing system by facilitating the early resolution of ambiguity. L$*$ parsing defines a general framework for specifying parsers with different control strategies.

7.1 Summary of L$*$ parsing

L$*$ parsing facilitates early resolution of ambiguity by allowing grammar rules to be applied whenever they are likely to provide syntactic information that permits an NLP system to perform useful semantic or pragmatic work.

L$*$ parsing is a table-driven algorithm. It has been implemented by extending the bottom-up GLR parser to include two new parse actions—the *eager-reduce* and *combine* actions. These actions are stored in the parse table. The eager-reduce action allows an L$*$ parser to perform a reduction by a grammar rule before all symbols of the rule's RHS have been parsed, creating an incomplete derivation. The combine action then incorporates the missing RHS symbols into the incomplete derivation when they are derived from later input.

Control strategies for the L$*$ parser are expressed by specifying the circumstances in which eager reductions are to be performed. One particular approach to performing eager reductions is to eagerly reduce only when doing so generates a syntactic attachment. Syntactic attachment provides a way of determining whether or not a reduction is likely to allow useful semantic processing. A method of compiling this approach into an L$*$ parse table has been designed and implemented.

To make the L$*$ algorithm practical, a method for removing contextual distinctions encoded in states of the parse table has been designed and implemented. Contextual distinctions reduce opportunities for the L$*$ parser to share and pack parses. This makes the parser inefficient because it unnecessarily repeats work. Contextual distinctions are removed by the use of equivalence classes. An equivalence class contains all states that are differentiated only by contextual distinctions. When parsing, all states of the same equivalence class at the top of the stack are merged together. Equivalence classes can also be used with LR(k) grammars to remove the contextual distinctions introduced by lookahead.

The L$*$ parser is intended to be used as part of a larger natural language processing system. A prototype interface for the interaction of the parser and an oracle that evaluates parses has been designed and implemented. When the oracle rejects parses created by eager reduction, the parser stops all work on completing the parse. To implement this, an extra table storing the number of kernel items in each state is built in conjunction with building the parse table.

7.2 Work in progress

This report describes an ongoing research project. Current lines of research are summarised in the following sections.

7.2.1 Integration of equivalence classes, oracle, and packing

A version of the L$*$ parser that combines the algorithms described in chapters 5 and 6 has been implemented. While there appear to be no problems with the integration of the two algorithms, further testing is needed.

7.2.2 Extending L$*$ parsing to a larger class of context-free grammars

The current version of the algorithm does not function correctly with grammars involving null rules (grammar rules that specify that a nonterminal derives the empty string). Any context-free grammar with null rules can be transformed to a grammar with no null rules (Aho and Ullman, 1972). However, the number of rules resulting from such a transformation can be impractically high. Also, transformations of the grammar are generally undesirable, because the relationships between symbols in the grammar may have implications for semantic processing that are altered by transforming the grammar.

Tomita's version of the GLR algorithm did not process null rules correctly either. Although the GLR parser could parse with grammars involving null rules, the method for dealing with them produced inefficient parse forests, and could not cope with cyclic grammars. A better method of parsing grammars with null rules is to allow cycles in the graph-structured stack (Nozohoor-Farshi, 1991; Rekers, 1992).

A version of the L$*$ algorithm without packing that processes null rules has been implemented, though not fully tested. Also, the implications of null rules when using the L$*$ parser with packing have not yet been investigated. In particular, reductions by null rules may cause problems for determining the order in which reductions should be performed.

7.2.3 Extending L$*$ parsing to other grammar formalisms

Although context-free grammars can model a useful subset of natural language, they cannot express all features. Also, describing some features of natural language with a context-free grammar may take many grammar rules, and fail to capture underlying structure and regularities. For example, to express subject-verb agreement in a context-free grammar requires different rules for singular and plural subjects.

A common method of addressing these problems is to augment context-free grammars with some form of parameter mechanism. This is the approach of constraint-based grammar formalisms (Shieber, 1992), examples of which include FUG (Kay, 1982), LFG (Kaplan and Bresnan, 1982), and PATR (Shieber, 1992). Tomita has addressed issues in parsing augmented context-free grammars with the GLR algorithm (Tomita, 1987a).

The L$*$ algorithm has been extended to use affix grammars over a finite lattice (AGFLs) (Nederhof and Sarbo, 1993), which are a restricted form of affix grammars (Koster, 1991). However, the implementation does not perform local ambiguity packing, and has not been fully tested. Sample AGFLs of English and Turkish have been obtained from the University of Nijmegen, The Netherlands, through Mark-Jan Nederhof, for testing purposes.

Parsing with parameters presents additional problems for L$*$ parsing. In particular, an eager reduction may be create an incomplete derivation with the value of a parameter unbound. Cascaded reductions may then place constraints on the possible values of this unbound parameter.

When performing a later combine action, the parser may discover inconsistencies between these constraints and the value of the parameter for the derivation being combined. The currently implemented solution to this problem is to split forest nodes when performing combine actions to remove any inconsistencies.

7.2.4 Experimentation and evaluation of L$*$ parsing

The L$*$ algorithm has currently been tested on a number of grammars specially designed to exploit the various mechanisms of L$*$ parsing. However, no evaluation of the effects of L$*$ parsing has been performed on a wide coverage grammar of natural language, with the parser interacting with a larger NLP system.

Work is in progress to perform such an evaluation. The PUNDIT natural language processing system (Lang and Hirschman, 1988) has been obtained under an educational licence for this purpose. PUNDIT is an NLP system written in Quintus Prolog. It analyses the syntax of sentences according to a restriction grammar of natural language (Hirschman and Puder, 1986, 1982). Work to date has seen PUNDIT ported to Sicstus Prolog.

7.3 Future work

There are a number of possible extensions to the work described in this report.

7.3.1 Changing the way the L$*$ parser pursues parses

An important modification for use in natural language would be to change the current breadth-first pursuit of all parses to a best-first search. The most promising parse would always be considered, without expending effort on parsing less likely interpretations. The parser would receive goodness ratings on the current parses, and always pursue the one with the highest goodness rating. This would involve some interesting changes, because input can no longer be processed in a strict left-to-right fashion. A promising parse might be pursued for a number of words before it is discovered to be wrong (such as in a garden-path sentence), at which time the parser might back up in the input and pursue a different interpretation.

7.3.2 Determining where to eagerly reduce

Another major area for future research is determining eager reduction points in the grammar. This report has presented one strategy for producing eager reductions from a grammar. This is far from the only possibility, however, and there is scope for investigating other possible methods of determining eager reduction points. One interesting possibility is to explore probabilistic, corpus-based approaches.

7.3.3 Dealing with ungrammatical input

Dealing with ungrammatical input is a serious problem for real NLP systems. Work in this area addressing adapting the GLR algorithm to handle ungrammatical input (Malone and Felshin, 1991) could be extended to the L$*$ parsing framework.

7.4 Summary

This report has described a new parsing algorithm for natural language processing that is intended to increase the efficiency of an NLP system by facilitating early resolution of ambiguity. The initial design and implementation has been completed. However, many issues still remain to be addressed.

References

Abney, S. P. and Johnson, M. (1991) Memory requirements and local ambiguities of parsing strategies. *Journal of Psycholinguistic Research*, *20*(3), 233–250.

Aho, A. V. and Ullman, J. D. (1972) *The Theory of Parsing, Translation, and Compiling*, Vol. 1: Parsing. Prentice-Hall.

Aho, A. V. and Ullman, J. D. (1977) *Principles of Compiler Design*. Addison-Wesley Publishing Co.

Billot, S. and Lang, B. (1989) The structure of shared forests in ambiguous parsing. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, pp. 143–151. Vancouver, British Columbia. Also published as INRIA Rapports de Recherche 1038.

Birnbaum, L. (1986) *Integrated Processing in Planning and Understanding*. Ph.D. thesis, Yale University, New Haven.

Crain, S. and Steedman, M. (1985) On not being led up the garden path: The use of context by the psychological syntax processor. In Dowty, D., Karttunen, L., and Zwicky, A. (Eds.), *Natural Language Parsing: Psychological, Computational, and Theoretical Perspectives*, chap. 10, pp. 320–358. Cambridge University Press.

DeRemer, F. L. (1971) Simple LR(k) grammars. *Communications of the ACM*, *14*(7), 453–460.

Earley, J. (1970) An efficient context-free parsing algorithm. *Communications of the ACM*, *13*(2), 94–102. Reprinted in Readings in Natural Language Processing, Grosz et. al (Eds.).

Frazier, L. (1987) Theories of sentence processing. In Garfield, J. L. (Ed.), *Modularity in Knowledge Representation and Natural Language Understanding*, chap. 15, pp. 291–307. MIT Press.

Gazdar, G., Klein, E., Pullum, G. K., and Sag, I. A. (1985) *Generalised Phrase Structure Grammar*. Basil Blackwell.

Gazdar, G. and Mellish, C. (1989) *Natural Language Processing in Prolog*. Addison-Wesley Publishing Co.

Hirschman, L. and Puder, K. (1982) Restriction grammar in prolog. In Van Canegham, M. (Ed.), *Proceedings of the First International Logic Programming Conference*, pp. 85–90. Marseilles. Association pour la Diffusion et le Developpement de Prolog.

Hirschman, L. and Puder, K. (1986) Restriction grammar: A prolog implementation. In Warren, D. and Van Canegham, M. (Eds.), *Logic Programming and its Applications*, pp. 244–261. Ablex Publishing Corporation.

Hopcroft, J. E. and Ullman, J. D. (1979) *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Co.

Jacobs, P. S., Krupka, G. R., and Rau, L. F. (1991) Lexico-semantic pattern matching as a companion to parsing in text understanding. In *Proceedings of the Speech and Natural Language Workshop*, pp. 337–341. Pacific Grove, CA. Morgan Kaufmann Publishers, Inc.

Jones, E. K. and Miller, L. M. (1992) Eager GLR parsing. In *First Australian Workshop on Natural Language Processing and Information Retrieval*, pp. 1–8. Melbourne, Australia.

Jones, E. K. and Miller, L. M. (1993) The L$*$ parsing algorithm. Technical Report CS-TR-93/9, Victoria University of Wellington.

Jones, E. K. and Miller, L. M. (1994a) L$*$ parsing: A general framework for syntactic analysis of natural language. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-94)*. Washington. In press.

Jones, E. K. and Miller, L. M. (1994b) Table-driven parsing with flexible control. In Gupta, G. (Ed.), *Proceedings of the Seventeenth Annual Computer Science Conference*, pp. 105–113. Christchurch, New Zealand.

Kaplan, R. M. (1973) A general syntactic processor. In Rustin, R. (Ed.), *Natural Language Processing*, pp. 193–241. Academic Press.

Kaplan, R. M. and Bresnan, J. (1982) Lexical-functional grammar: A formal system for grammatical representation. In Bresnan, J. (Ed.), *The Mental Representation of Grammatical Relations*, pp. 172–281. MIT Press.

Kay, M. (1982) Parsing in functional unification grammar. In Dowty, D. R., Karttunen, L., and Zwicky, A. M. (Eds.), *Natural Language Parsing*, pp. 251–278. Cambridge University Press.

Kay, M. (1986) Algorithm schemata and data structures in syntactic processing. In Grosz, B. J., Jones, K. S., and Webber, B. L. (Eds.), *Readings in Natural Language Processing*, pp. 35–70. Morgan Kaufmann Publishers, Inc.

Koster, C. H. A. (1991) Affix grammars for natural language. In *Attribute Grammars, Applications and Systems, International Summer School SAGA*, Vol. 545 of *Lecture Notes in Computer Science*, pp. 358–373. Springer-Verlag.

Lang, B. (1974) Deterministic techniques for efficient non-deterministic parsers. In Loeckx, J. (Ed.), *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*, Vol. 14 of *Lecture Notes in Computer Science*, pp. 255–269. Saarbrücken. Springer-Verlag.

Lang, F. M. and Hirschman, L. (1988) Improved portability and parsing through interactive aquisition of semantic information. In *Proceedings of the Second Conference of Applied Natural Language Processing*, pp. 49–57. Austin, Texas. ACL.

Lankhorst, M. M. (1991) An empirical comparison of Generalised LR tables. In *Proceedings of the Workshop on Tomita's Algorithm—Extensions and Applications*, pp. 92–98. P. O. Box 217, Enschede, The Netherlands. University of Twente, Computer Science Department.

Leermakers, R. (1989) How to cover a grammar. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, pp. 135–142. Vancouver, British Columbia.

Malone, S. and Felshin, S. (1991) GLR parsing for erroneous input. In Tomita, M. (Ed.), *Generalized LR Parsing*, chap. 9, pp. 129–139. Kluwer Academic Publishers.

Maruyama, H. (1990) Structural disambiguation with constraint propagation. In *Proceedings of the 28th Annual Meeting of the Association for Computational Linguistics*, pp. 31–38. Pittsburgh, Pennsylvania.

Miller, L. M. (1994) Flexible table-driven parsign for natural language processing. Master's thesis, Victoria University of Wellington.

Miller, L. M. and Jones, E. K. (1992) Eager GLR parsing. In *Proceedings of the New Zealand Computer Science Research Students' Conference*, pp. 175–178. Hamilton, New Zealand.

Nederhof, M.-J. (1993) Generalised left-corner parsing. In *Proceedings of the 6th Conference of the European Chapter of the Association for Computational Linguistics*, pp. 305–314. Utrecht, The Netherlands.

Nederhof, M.-J. and Sarbo, J. J. (1993) Efficient decoration of parse forests. In Trost, H. (Ed.), *Feature Formalisms and Linguistic Ambiguity*, chap. 4, pp. 53–78. Ellis Horwood.

Nirenburg, S., Carbonell, J., Tomita, M., and Gooman, K. (1992) *Machine Translation: A Knowledge-Based Approach*. Morgan Kaufmann Publishers, Inc.

Nozohoor-Farshi, R. (1991) GLR parsing for ϵ-grammars. In Tomita, M. (Ed.), *Generalized LR Parsing*, chap. 5, pp. 61–75. Kluwer Academic Publishers.

Palmer, M. S., Passonneau, R. J., Weir, C., and Finin, T. (1993) The KERNEL text understanding system. *Artificial Intelligence*, *63*, 17–68.

Pereira, F. C. N. and Warren, D. H. D. (1980) Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artifical Intelligence*, *13*, 231–278.

Perrault, C. R. (1984) On the mathematical properties of linguistic theories. *Computational Linguistics*, *10*, 165–176. Reprinted in Readings in Natural Language Processing, Grosz et. al (Eds.).

Pollard, C. J. and Sag, I. A. (1987) *Information-based Syntax and Semantics*. No. 13 in CSLI Lecture Notes. Center for the Study of Language and Information.

Rau, L. F. and Jacobs, P. S. (1988) Integrating top-down and bottom-up strategies in a text processing system. In *Proceedings of the Second Conference of Applied Natural Language Processing*, pp. 129–135. Austin, Texas. ACL.

Rekers, J. (1992) *Parser Generation for Interactive Environments*. Ph.D. thesis, University of Amsterdam.

Sager, N. (1981) *Natural Language Information Processing: A computer grammar of English and its applications*. Addison-Wesley Publishing Co.

Schabes, Y. (1991) Polynomial time and space shift-reduce parsing of arbitrary context-free grammars. In *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics*, pp. 106–113. Berkeley, CA.

Shann, P. (1991) Experiments with GLR and chart parsing. In Tomita, M. (Ed.), *Generalized LR Parsing*, chap. 2, pp. 17–34. Kluwer Academic Publishers.

Shieber, S. M. (1987) Evidence against the context-freeness of natural language. In Savitch, W., Bach, E., Marsh, W., and Safran-Naveh, G. (Eds.), *The Formal Complexity of Natural Language*, Vol. 33 of *Studies in Linguistics and Philosophy*, pp. 320–335. D. Reidel Publishing Co.

Shieber, S. M. (1992) *Constraint-Based Grammar Formalisms: Parsing and type inference for natural and computer languages*. MIT Press.

Steele Jr., G. L. (1990) *Common Lisp—The Language* (second edition). Digital Press.

Stowe, L. A. (1991) Ambiguity resolution: Behavioural evidence for a delay. In *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society*, pp. 257–262. Cognitive Science Society.

Taraban, R. and McClelland, J. L. (1988) Constituent attachment and thematic role assignment in sentence processing: Influences of content based expectations. *Journal of Memory and Language*, *27*, 597–632.

Tomita, M. (1985) An efficient context-free parsing algorithm for natural language. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, pp. 756–764. Los Angeles, California. Morgan Kaufmann Publishers, Inc.

Tomita, M. (1986) *Efficient Parsing for Natural Language*. Kluwer Academic Publishers.

Tomita, M. (1987a) An efficient augmented-context-free parsing algorithm. *Computational Linguistics*, *13*(1-2), 31–46.

Tomita, M. (1987b) The universal parser architechture for knowledge-based machine translation. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, pp. 718–721.

Tomita, M. (1988) Graph-structured stack and natural language parsing. In *Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics*, pp. 249–256. Buffalo, New York. ACL.

Tomita, M. and Ng, S. (1991) The generalized LR parsing algorithm. In Tomita, M. (Ed.), *Generalized LR Parsing*, chap. 1, pp. 1–16. Kluwer Academic Publishers.

Tyler, L. K. and Marslen-Wilson, W. (1977) The on-line effects of semantic context on syntactic processing. *Journal of Verbal Learning and Verbal Behaviour*, *16*, 683–692.

Winograd, T. (1983) *Language as a Cognitive Process. Volume 1: Syntax*. Addison-Wesley Publishing Co.