Report Number: WUCS-91-51

1991-10-01

A Parser Generator Based on Earley's Algorithm

Jawaid Hakim

Most parser generators are programs that take a context-free grammar specification for a language and generate a parser for that language. Usually, the parsers generated by these parser generators are based on some variations of LL(k) or LR(k) parsing algorithms. The parser generators discussed in this paper creates parsers based on Earlcy's Algorithm. This parser generator creates parsers from any arbitrary context-free grammar specifications; even from ambiguous, cyclic, and unbounded look ahead grammar. These parsers are more powerful than LL(k) and LR(k) parsers and enable the user to create many new applications.

... Read complete abstract on page 2.

A Parser Generator Based on Earley's Algorithm

Jawaid Hakim

Complete Abstract:

Most parser generators are programs that take a context-free grammar specification for a language and generate a parser for that language. Usually, the parsers generated by these parser generators are based on some variations of LL(k) or LR(k) parsing algorithms. The parser generators discussed in this paper creates parsers based on Earlcy's Algorithm. This parser generator creates parsers from any arbitrary context-free grammar specifications; even from ambiguous, cyclic, and unbounded look ahead grammar. These parsers are more powerful than LL(k) and LR(k) parsers and enable the user to create many new applications.

A Parser Generator Based on Earley's Algorithm

Jawaid Hakim

WUCS-91-51

October 1991

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

A project presented to the Sever Institute of Washington University in partial fulfillment of the requirements for the degree of Master of Computer Science. Implemented under the direction of Professor Will D. Gillett.

Abstract

Most parser generators are programs that take a context-free grammar specification for a language and generate a parser for that language. Usually, the parsers generated by these parser generators are based on some variation of LL(k) or LR(k) parsing algorithms. The parser generator discussed in this paper creates parsers based on Earley's Algorithm. This parser generator creates parsers from any arbitrary context-free grammar specification; even from ambiguous, cyclic, and unbounded lookahead grammars. These parsers are more powerful than LL(k) and LR(k) parsers and enable the user to create many new applications.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF GRAMMARS

TABLE OF TABLES

1. Introduction

Context-free grammars (CFGs) have been used extensively for describing the syntax of programming and natural languages. They are often called BNF grammars when used for programming languages. The analysis of the syntax (parsing) of programs or sentences is a crucial part in the implementation of compilers and interpreters for programming languages. Therefore, the development of automatic parser generators (APGs) for CFGs has importance in these areas of computer science. Any computer analysis of natural languages also involves parsing. Therefore, APGs are also useful in the study of natural languages.

An APG is useful in compiler-writing or similar applications that deal extensively with syntax considerations for many reasons. The main reason is the convenience of having to supply only the BNF grammar of the language in order to obtain a parser. The popularity of YACC is an indication of the worth of such tools to programmers.

The algorithm proposed by Jay Clark Earley is a powerful parsing method capable of parsing strings in a context-free language, given any CFG that generates that language. An APG based on Earley's Algorithm provides many new and exciting applications, and will be beneficial in the study of languages.

This paper presents an APG based on Earley's Algorithm, called PEG (Earley's Parser Generator). PEG is a parser generator that accepts any arbitrary CFG, and generates a parser based on that CFG. Like YACC, semantic actions can be attached to the productions of a CFG; the semantic actions of a production are executed when that production is used in the derivation of an input string. Unlike YACC, PEG is able to generate parsers even from ambiguous, cyclic, and disconnected CFGs. Furthermore, PEG can generate parsers from CFGs of any lookahead, including those with unbounded lookahead. The parser generated by PEG is more powerful than parsers based on LL(k) and LR(k) parsing algorithms.

If the CFG on which the PEG parser is based is unambiguous and does not have cycles, then every string in the language defined by the CFG has a degree of ambiguity of 1 (only 1 derivation tree). If the CFG has cycles, then the string being parsed can have an unbounded degree of ambiguity. No matter what the degree of ambiguity of the CFG is, the parser generates a derivation graph (an extension of the concept of a derivation tree) that encodes all the possible derivations trees of a string. Moreover, functions are provided that enable the user to manipulate this derivation graph.

The PEG parser provides special functions to the user so that the CFG on which the parser is based can be changed at execution time. This allows the user to implement syntactically extendible languages (the ability to syntactically change the language at execution time).

This paper is divided into eleven main section. Section 2 gives an introduction to the theory of languages and parsing, with emphasis on CFGs. Section 3 gives an introduction to Earley's Algorithm. It describes how parsing is done in Earley's Algorithm, the idea of state sets and states, and the three functions (predictor, scanner, and completer) used in the algorithm. Section 4 describes the usage of PEG. The structure of the grammar specification that PEG uses to generate a parser is described. Section 5 describes the implementation of PEG. It describes the data structures used to implement the parser. Section 6 presents some applications that illustrate the power of the parsers generated by PEG. Section 7 describes the structure of the derivation graph generated by these parsers. Section 8 describes the derivation graph manipulation routines made available to the user by PEG. Section 9 gives statistics on the execution time performance displayed by a few sample parsers. Section 10 gives the grammar specification for PEG. Finally, Section 11 highlights the issues discussed in the previous sections.

2. Languages and Parsing

This section is an introduction to the computer science concept of languages and parsing. The subject is introduced in a fairly informal manner. For a more detailed exploration of the concepts, please refer to the bibliography.

2.1. Languages and Context-free Grammars (CFGs)

A language is a set of strings (sentences), the strings being composed of the finite concatenation of symbols from some predefined alphabet. The empty string is represented by λ. The symbols of this alphabet are called **terminal** symbols. Languages can be finite (i.e., contain a finite number of strings) or infinite. Languages are syntactic entities; they have no intrinsic meaning by themselves. Of course, meaning (semantics) can be attached to the strings in a language. However, this in an interpretation and is not intrinsic to the language itself; it is instead an external property of the language which the interpreter chooses to attach to the language. In general, the language being described is called the **object language**, and the language by which the object language is described is called the **metalanguage**.

There is a set of symbols that do not appear in the strings of the object language but instead give information about the sequence in which the terminal symbols can appear. These are called the **nonterminal** symbols, and are represented by uppercase letters (e.g. A, B, C). Sequences of terminal and nonterminal symbols (including λ) are represented by Greek letters (e.g. α, β, γ).

Since languages are sets of strings, some mechanism is needed to specify which strings are in these sets. CFGs are one such mechanism. A CFG has a finite set of productions or rewriting rules of the form 'A $\rightarrow$ γ'. The metasymbol '$\rightarrow$' can be read as "goes to". A special nonterminal is designated as the **root** of the grammar; by default this is taken to be the nonterminal on the left-hand side of the first production of the grammar. The productions with a particular nonterminal on the left-hand side are called the **alternatives** of that left-hand side nonterminal. For convenience, the various alternatives of a nonterminal may be written together by separating them with the metasymbol '|'.

As an example of a CFG, Grammar 2.1 generates all strings of well-formed parentheses and only such strings:

$$S \rightarrow (\; S \;) \; S$$

$$S \rightarrow \lambda$$

Grammar 2.1

A CFG may be completely specified by the 4-tuple: <T,N,S,P>. Where T is the set of terminal symbols, N is the set of nonterminal symbols, S is the start/root symbol of the grammar, and P is the set of productions. For example, Grammar 2.1 = $< \{(,)\} , \{S\} , S , \{S \rightarrow (S) S, S \rightarrow \lambda\} >$.

2.1.1. Derivations

A production is treated as a rewriting rule in which the nonterminal on the left-hand side is replaced by the symbols on the right-hand side of the production. Consider a CFG one of whose productions is 'E $\rightarrow$ – E'. It means that a single 'E' can be replaced by '– E'. This action of replacement can be described by writing

$$E \Rightarrow -E$$

which is read as "E derives – E". A sequence of such replacements is called a **derivation**. In a more abstract setting, one can say $\alpha E\beta \Rightarrow \alpha\gamma\beta$ only if 'E $\rightarrow \gamma$' is a production of the grammar. The strings $\alpha E\beta$ and $\alpha\gamma\beta$ are called **working strings**. For convenience, $\Rightarrow^+$ is defined to read as "derives in one or more steps". Taking Grammar 2.1, a derivation for the string '(( ))' would be:

$$\underline{S} \Rightarrow \overline{(\underline{S})S} \Rightarrow (\overline{(\underline{S})S})S \Rightarrow (()\underline{S})S \Rightarrow (())\underline{S} \Rightarrow (())$$

where the underlined nonterminal in a working string is replaced by the overlined symbols in the next working string; if the nonterminal is substituted by the empty string (λ) then that nonterminal is simply eliminated from the next working string.

Usually, the start symbol 'S' of the grammar is taken as the first working string. If a working string consisting of only terminal symbols can be derived from 'S', then that working string is a member of the language generated by the grammar. The derivation given above is called the **left-most derivation** of the string (the left-most nonterminal is chosen for substitution in a working string). In general, a string can have many possible derivations depending on the order in which the nonterminal symbols are chosen for replacement.

2.1.2. Derivation Trees and Ambiguity

It would be useful to have some mechanism to equate similar derivations, i.e., derivations in which the same productions are applied to the same nonterminals in the working strings, the specific order of application being unimportant. The concept of the **derivation tree** is one such mechanism.

The derivation tree filters out or hides the choice regarding replacement order. It supplies an equivalence relation over the set of all derivations of a string. For example, given the language $\{a^m b^n | \; m, n \geq 1\}$ and Grammar 2.2 that generates this language

$$\begin{array}{llllll} E & \rightarrow & A & B & & \\ A & \rightarrow & a & A & | & a \\ B & \rightarrow & b & B & | & b \end{array}$$

Grammar 2.2

the derivation tree for the string 'aabb' is given in Figure 2.1. The string derived can be read from the derivation tree by reading the leaf-nodes of the tree from left to right. Note that this string has one and only one derivation tree under this grammar.

Consider the following grammar which specifies a subset of the arithmetic expressions.

$$E \rightarrow E * E \mid E - E \mid \text{val} \mid (E)$$

Grammar 2.3

Using Grammar 2.3, there are two distinct derivation trees of the string '9–7–5'. These are given in Figure 2.2 and Figure 2.3.

A grammar is called **ambiguous** if there exists a string in the language described by the grammar that has more than one derivation tree. Otherwise, the grammar is called **unambiguous**. Ambiguity that is introduced due to cycles in a grammar (see Section 2.3) will be referred to as **indirect ambiguity** in this paper. On the other hand, ambiguity that is not due to cycles will be referred to as **direct ambiguity**.

Why is ambiguity undesirable? Within a language processing system, the meaning or interpretation is often based on the structure of the derivation tree. Two different derivation trees may imply two different meanings. In the example of Figure 2.2 and Figure 2.3, a compiler would not be able to decide without making assumptions about what the programmer probably intended, whether the semantics of the string '9–7–5' should be the value 7 or –3.

If there is no unambiguous CFG that defines a language, then the language is called

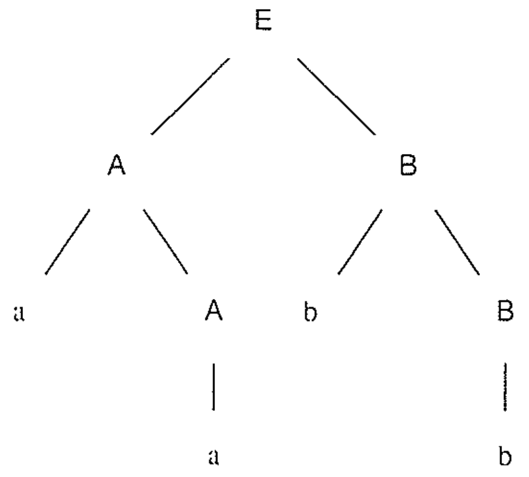


Figure 2.1: Derivation Tree of 'aabb' under Grammar 2.2

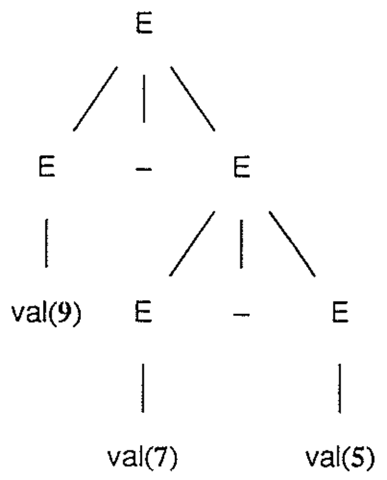


Figure 2.2: First Derivation Tree of '9–7–5' under Grammar 2.3

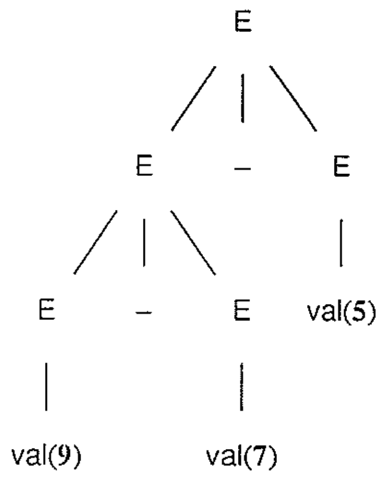
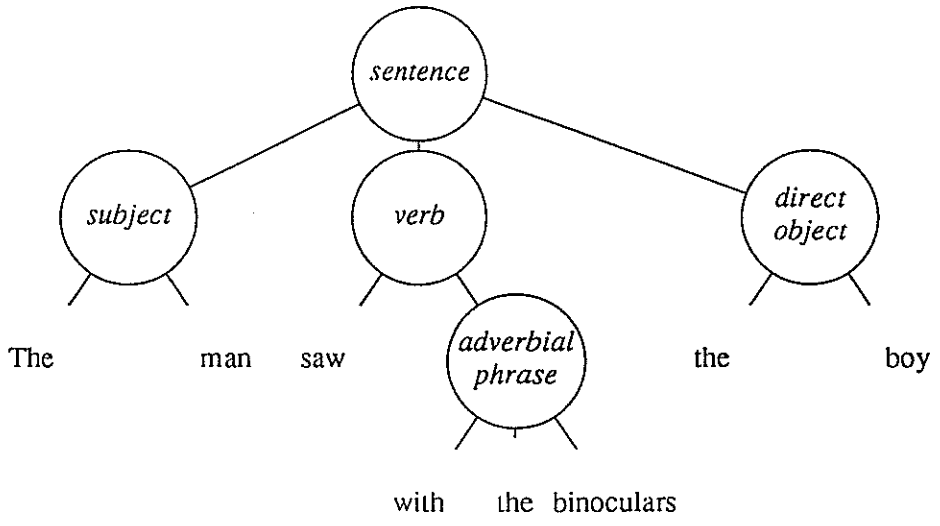
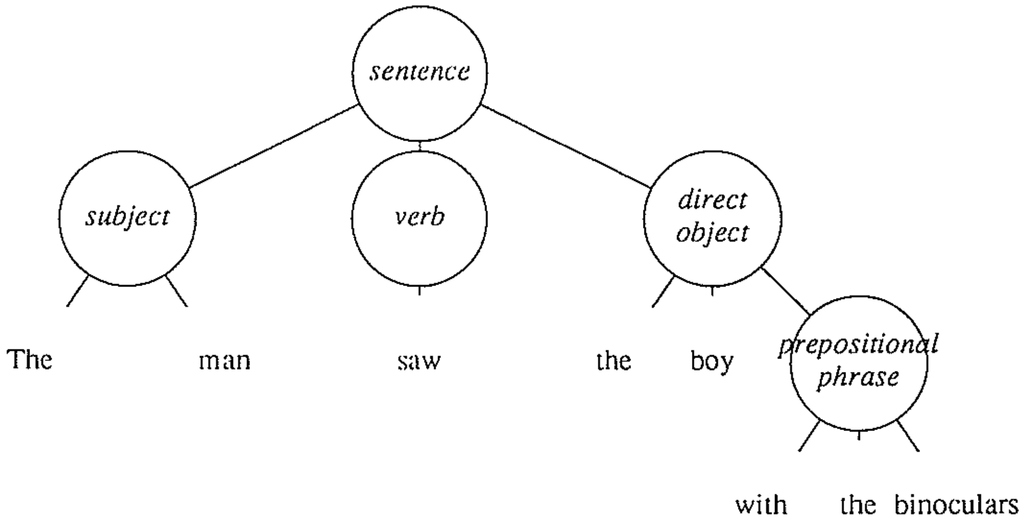


Figure 2.3: Second Derivation Tree of '9–7–5' under Grammar 2.3

inherently ambiguous. Natural languages are inherently ambiguous. As an example of the syntactic analysis performed in human speech processing, consider the English sentence "The man saw the boy with the binoculars". There are at least two interpretations of this sentence, as implied by the abstract syntax structure diagrams shown in Figure 2.4. The syntax structure presented in Figure 2.4(a) implies that the binoculars were the mechanism used to perform the seeing. The syntax structure presented in Figure 2.4(b) implies that the binoculars were possessed by the boy. One of these two syntactic structures must be selected before any semantics or meaning can be extracted from the sentence.



(a)



(b)

Figure 2.4: Abstract Syntax Structures for 'The man saw the boy with the binoculars'

It has been proven that there is no algorithm that can determine whether an arbitrary CFG is ambiguous or not. Therefore, writing a correct grammar to define a language is a matter of experience, and some insist it is an art.

2.2. Lookahead

During parsing, it may be necessary for a parser to look ahead a certain *k* number of symbols, the next *k* input tokens (see Section 4), in order to make its parsing decisions. These *k* symbols that the parser needs are called the **lookahead symbols**. Consider the following grammar:

$$S \rightarrow A \quad a \quad c$$

$$S \rightarrow a \quad a$$

$$A \rightarrow a$$

Grammar 2.4

If an LR(k) parser is constructed based on Grammar 2.4, then it will need at least 2 lookahead symbols for parsing. In other words, an LR(0) or LR(1) parser cannot be constructed based on Grammar 2.4. For example, YACC will not be able to generate a parser for Grammar 2.4 because the parsers generated by YACC use a lookahead of 1. Clearly, a parsing algorithm that is designed to work with any arbitrary CFG will either have to compute additional lookahead symbols "as they are needed" or use some other strategy to "get around" the problem.

2.3. Cycles in Grammars

If a CFG has no derivations of the form

$$E \Rightarrow^{+} E$$

for any nonterminal E, then the grammar is said to be **acyclic**. Otherwise, the grammar is said to be **cyclic**. Each derivation of the form $E \Rightarrow^{+} E$, for any nonterminal E, is called a **cycle**. The starting nonterminal of a cycle is called the **head** of the cycle, and the nonterminal preceding the last nonterminal of a cycle is called the **tail** of the cycle. For example, in the cycle

$$E \Rightarrow A\ B \Rightarrow B \Rightarrow E$$

the head is 'E' and the tail is 'B'. A grammar may contain more than one cycle. Cycles are useless in the sense that they do not contribute any terminal symbols towards the derivation of a string. A cyclic grammar may or may not be ambiguous. Consider the following example of a cyclic grammar which is unambiguous:

$$S \rightarrow a \mid B$$
$$B \rightarrow C$$
$$C \rightarrow B$$

Grammar 2.5

Grammar 2.5 has a cycle, $B \Rightarrow^+ B$. This grammar is unambiguous because for the only string in the language generated by this grammar, 'a', there is one and only one derivation tree. The following is an example of an ambiguous grammar with a cycle:

$$S \rightarrow B$$
$$B \rightarrow C$$
$$C \rightarrow S \mid a$$

Grammar 2.6

In fact, Grammar 2.6 generates an infinite number of derivation trees for the string 'a'. This is because the cycle can be followed an arbitrary number of times to generate different derivation trees for the only string in the language.

2.4. Concept of a Derivation Graph

An ambiguous grammar generates more than one derivation tree for certain strings. Furthermore, a cyclic grammar may generate an infinite number of derivation trees for certain strings. Therefore, a mechanism is needed to encapsulate the possibly infinite number of derivation trees (and ambiguity) into a structure that can be stored in a finite amount of space and time. A **derivation graph** is such a mechanism. A derivation graph is an encoding of all the possible derivation trees of a string under a CFG. The structure of a derivation graph is similar to the structure of a derivation tree. However, to encode cycles and ambiguity the derivation graph contains certain additional structures.

2.4.1. Backward Arc Structure to Encode Indirect Ambiguity

To encode the indirect ambiguity (see Section 2.1.2), the derivation graph contains the **backward arc** structure. Given the grammar

$$S \rightarrow A$$

$$A \rightarrow S \mid (\ )$$

Grammar 2.7

the conceptual picture of the derivation graph for the string '( )' is given in Figure 2.5. The backward arc identifies the cyclic construct of the grammar; it connects the tail of the cycle to its head. This graph encapsulates an infinite number of derivation trees.

2.4.2. Alternatives Structure to Encode Direct Ambiguity

As defined so far, the derivation graph is a powerful encapsulation mechanism. However, the backward arc only encapsulates cycles (indirect ambiguity), and for certain grammars may not suffice in encoding all the derivation trees of a string. Consider the following ambiguous and acyclic grammar.

$$\begin{array}{lll}
S & \rightarrow & A\ B \\
A & \rightarrow & C \mid D \\
B & \rightarrow & E \mid F \\
C & \rightarrow & a \\
D & \rightarrow & a \\
E & \rightarrow & b \\
F & \rightarrow & b
\end{array}$$

Grammar 2.8

$$\begin{tikzcd}
 & S \arrow[d] & \\
 & A \arrow[u, bend right] \arrow[dl] \arrow[dr] & \\
(& &)
\end{tikzcd}$$

Figure 2.5: Conceptual Derivation Graph of '( )' under Grammar 2.7

The only string in the language described by Grammar 2.8 is 'ab'.

To encode the direct ambiguity (see Section 2.1.2) of Grammar 2.8, an **alternatives structure** is used. The conceptual derivation graph of the string 'ab' under Grammar 2.8 is given in Figure 2.6. In the internal representation of derivation graphs, every nonterminal node in the derivation graph has an alternatives structure associated with it. The alternatives structure is represented by a list, and the cardinality of this list gives the number of **alternative paths** available from that particular nonterminal node. If the cardinality of an alternatives structure associated with a nonterminal node is 1, then there is only one path available from that nonterminal node. In other words, in a derivation of the input string, only one production can be applied to that particular nonterminal (see Section 2.1.1). On the other hand, if the cardinality of an alternatives structure is greater than 1, then any one of a number of productions could be applied to the associated nonterminal node in a derivation of the input string.

Pictorially, an alternatives structure with a cardinality greater than 1 is represented by dashed lined. In Figure 2.6, the nonterminals 'A' and 'B' are connected to their children nodes by dashed lines. These dashed lines identify the alternative paths that could be followed from 'A' and 'B' in a derivation of the string 'ab'. In other words, in a derivation of the string 'ab', the nonterminal 'A' could be replaced by either 'C' or 'D', and the nonterminal 'B' could be replaced by either 'E' or 'F'. If the cardinality of the alternatives structure associated with a nonterminal is 1, then that nonterminal is connected to its children nodes by solid lines. Notice that Figure 2.6 is an encodement of four derivation trees.

Ignoring the backward arc construct for the moment, a derivation graph may be thought of as an AND/OR tree. AND/OR trees are frequently used in artificial intelligence (AI) for problem solving. The AND construct is used to break down a problem into smaller more manageable sub-problem(s), and the OR construct is used to identify the alternate solutions to a problem. Figure 2.7 shows a pictorial representation of an AND/OR tree.

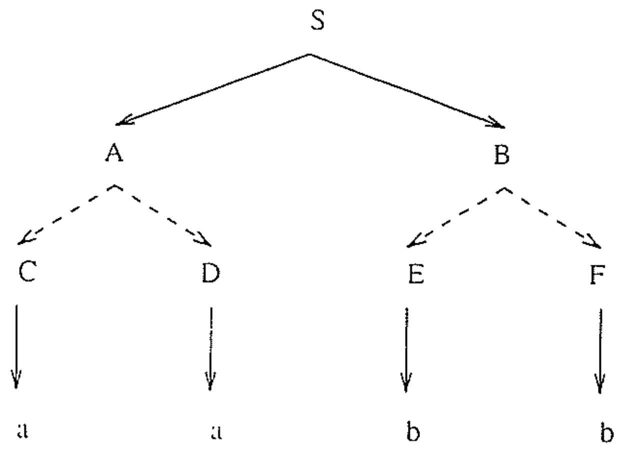


Figure 2.6: Conceptual Derivation Graph of 'ab' under Grammar 2.8

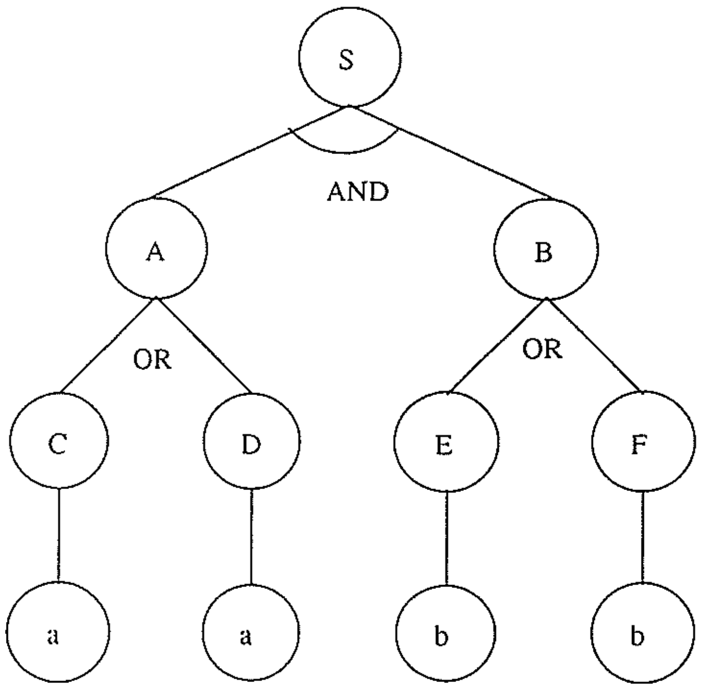


Figure 2.7: AND/OR Tree Corresponding to Derivation Graph of Figure 2.6

In the AND/OR tree of Figure 2.7 the original problem 'S' may be solved by breaking it up into sub-problems 'A' and 'B' and finding solutions to these sub-problems. Problem 'A' is solved by solving sub-problem 'C', or 'D'. Problem 'B' is solved by solving sub-problem 'E', or 'F'. Problems 'C' and 'D' are both solved by solving the sub-problem 'a', and 'E' and 'F' are both solved by solving the sub-problem 'b'. Note that sub-problems 'a' and 'b' can be visualized as being linked to their parent nodes by either AND or OR links (since there is only one link, AND and OR links would be semantically identical).

Notice the correspondence between the derivation graph of Figure 2.6 and the AND/OR tree of Figure 2.7. The derivation graph of Figure 2.6 could be interpreted as an AND/OR tree as follows: the dashed lines represent the OR construct and the solid lines represent the AND construct. It would seem that derivation graphs and AND/OR trees are isomorphic (one-to-one correspondence exists between the two). However, the backward arc construct of a derivation graph makes it conceptually more powerful than AND/OR trees.

2.5. Recognizers and Parsers

It is often of interest to verify whether a given string is in the language under consideration. To put it another way, given a CFG for a language it is of interest to verify whether a string has a derivation under that grammar. A **recognizer** is an algorithm which takes a string as input and either accepts or rejects it, depending on whether or not the string is a sentence in the language generated by the grammar on which the recognizer is based. A **parser**, besides recognizing strings in a language, also outputs the set of all legal derivation trees for the strings that it accepts.

2.6. Parsing Algorithms

A substantial effort has been devoted to developing efficient parsing algorithms. Among others, two important ways to approach parsing are Top-Down and Bottom-Up. Perhaps the most common and well-studied of these algorithms are LL(k) and LR(k) respectively. Both LL(k) and LR(k) algorithms scan the input string from left to right. LL(k) algorithms reconstruct the left-most derivation, and LR(k) algorithms reconstruct the right-most derivation of a string. The *k* in the LR(k) and LL(k) specifies the number of lookahead symbols used by the parser. Of these two parsing algorithms, LR(k) is more powerful because more languages can be described using LR-grammars than LL-grammars.

Most computer programming languages fall into the category that the LR(k) algorithm can parse. However, neither of these algorithms can parse strings using ambiguous grammars, and considerable modification of a grammar may be necessary to remove all ambiguity. Moreover, if the language under consideration is inherently ambiguous, then any CFG written for this language will be ambiguous.

Much work has been done to create universal parsing algorithms that can use any arbitrary CFG to parse strings. **Earley's Algorithm** is one such well-known algorithm.

3. Earley's Algorithm

This section presents an informal description of Earley's Algorithm, used as a recognizer. The algorithm is presented by giving a grammar and showing how a string can be recognized as a member of the language generated by te grammar. The time and space complexity of the algorithm, and its potential uses are also discussed.

3.1. Overview of Earley's Algorithm

Earley's Algorithm is a universal parsing algorithm in the sense that it can take any arbitrary CFG and parse a string to determine whether the string is in the language generated by the grammar. It does not require the input grammar to be in any special form.

Suppose that the input grammar is G = <T,N,S,P> where T is the set of terminal symbols, N is the set of nonterminal symbols, S is the start symbol of the grammar, and P is the set of productions. As a first step, Earley's Algorithm augments G to create G ′ = <T ′,N ′,S ′,P ′>. Two new symbols, \$ and ∇, are selected such that $\{T \cup N\} \cap \{\$,\nabla\} =$. Then grammar G is augmented in the following way to create G ′

$$
\begin{aligned}
T' &= T \cup \{\nabla\} \\
N' &= N \cup \{\$\} \\
S' &= \$ \\
P' &= P \cup \{\$ \rightarrow S'\,\nabla\}
\end{aligned}
$$

Earley's Algorithm scans an input string of terminal symbols $X_1 \ldots X_n$ from left to right, looking ahead some *k* fixed number of symbols. As each X_i is scanned a **state set** S_i is created.

Each state is a 4-tuple. The tuples are (1) a production such that an instance of its right-hand side is potentially being scanned, (2) a point in the production that shows how much of the right-hand side has been scanned so far (pictorially represented by a **dot**), (3) a pointer back to the state set where the search for this production started (where it was predicted) (4) a k-symbol string which can legally occur after that instance of the production (lookahead).

Parsing is done with the help of three functions. Depending on the form of a state, one of these three functions is applicable on it.

The first of these functions is called the **predictor**. Depending upon the state of the parsing activity, it predicts what syntactic classes might follow the string parsed so far. Intuitively, the predictor specifies which nonterminals to look for next. The predictor is applicable on states in which the dot precedes a nonterminal symbol. The predictor adds to the current state set (S_i) one state for each alternative of the nonterminal after the dot.

The second function is the **scanner**. The scanner is applicable on states in which the dot precedes a terminal symbol. For example, suppose the scanner is applied to a state in which the dot is before a terminal symbol. If the terminal symbol after the dot matches the next input symbol, then the scanner moves the dot over the terminal symbol, and adds this modified state to the next state set (S_{i+1}).

The third function is called the **completer**. The completer checks state set S_i for states that indicate that the right-hand side of a production has been derived. The completer is applicable on states in which the dot is after the last symbol on the right-hand side. It adds states to S_i to indicate that the particular instance of the nonterminal on the left-hand side of these productions has been found.

3.2. Earley's Algorithm as a Recognizer

This section presents Earley's Algorithm used as a recognizer. In general, processing on a state set S_i is performed as follows: The states in the state set are scanned in order, and one of the three operations, predictor, scanner, or completer is applied to each state depending on the form of the state. These operations may add more states to S_i and may also put states in a new state set S_{i+1}. A state set can contain only one instance of a particular state (duplicate states are not added). These three operations are described by example. Consider the language $\{a^n | \; n \geq 1\}$. One grammar that describes the language is

$$S \rightarrow a \; S$$

$$S \rightarrow a$$

Grammar 3.1

Suppose that a lookahead of 0 is being used, and the input string is 'a'. The original grammar, Grammar 3.1, is augmented with the Production '$ → S ∇'. The first state set S_0 is initialized with the following state: <$ → S ∇, 0, 0, λ>. Pictorially, this state will be displayed in the following way:

$$\$ \rightarrow .S \quad \nabla \quad 0$$

Note the following about the state given above. The dot '.' is a metasymbol and represents the second component of the state. It is a marker that shows how much of the right-hand side of a production has been scanned so far. The third component of the state is set to 0, to indicate that the search for this production started in state set S_0. Since a lookahead of 0 is being used, there is no k-symbol string contained in the state.

Initially, state set S_0 contains only one state $ → .S ∇ 0, and the predictor operation is applicable to this state because there is a nonterminal (S) to the right of the dot. The predictor operation causes one new state to be added to S_0 for each alternative of S. The dot is put at the beginning of the production in these new states because none of their right-hand side symbols have been scanned yet. The pointer in these new states is set to 0, since they were created (predicted) in S_0. In this example, the two states added to S_0 are:

$$\begin{array}{lllll} S & \rightarrow & .a & S & 0 \\ S & \rightarrow & .a & & 0 \end{array}$$

These two new states are now scanned. The predictor is not applicable to either of these two states because in both of them there is a terminal symbol to the right of the dot. However, the scanner operation is applicable. In this example, the two states added to S_1 by the scanner are:

$$\begin{array}{lllll} S & \rightarrow & a & .S & 0 \\ S & \rightarrow & a. & & 0 \end{array}$$

The predictor is applicable to the first of these new states, and the completer is applicable to the second. From the predictor, the two states added to S_1 are:

$$\begin{array}{lllll} S & \rightarrow & .a & S & 1 \\ S & \rightarrow & .a & & 1 \end{array}$$

The completer is applicable to a state where the dot is at the end of the production. If it finds such a state and the left-hand side of the production is some nonterminal E, it compares the lookahead string of the state with symbols $X_{i+1} \ldots X_{i+k}$ of the input string (k is the number of lookahead symbols being used). If they match, it goes back to the state set S_j indicated by the pointer (j is the third component of a state), and adds to the current state set all states from S_j which have the nonterminal E to the right of the dot. It moves the dot over the E in these new states. Intuitively, S_j is the state set where that E was predicted. Since that K has now been scanned, the dot is moved over the K in all the states of state set S_j which caused that K to be predicted. A state on which the completer is applicable is called a **final state**. Since a lookahead of 0 is being used in this example, no lookahead comparison is done. In this example, the completer adds the following state to S_1

$$\begin{array}{lllll} \$ & \rightarrow & S & .\nabla & 0 \end{array}$$

The only operation now applicable to the states in S_1 is the scanner. Since the input string is exhausted, the next token returned by the lexical analyzer is ∇. Therefore, the only new state added to S_2 is

$$\$ \rightarrow S\ \nabla.\ 0$$

The completer is applicable to this state, but no new states are added to S_2 by the completer. The parsing halts. This final state shows that the S that was predicted in state set S_0 has been successfully scanned, and the input string is exhausted. So the parse is finished, and the string is a sentence of the language generated by Grammar 3.1.

The algorithm described above carries along all the possible parses of a string under the given CFG. That, in essence, is the power of this algorithm. It allows the algorithm to handle any arbitrary CFG. With extensions to the recognizer described above, a parser can be created. With the parser it is possible to produce a derivation graph (see Section 1.4) that encapsulates all the derivation trees (see Section 1.1.2) of a string under a CFG.

3.3. Time and Space Complexity

Given the fact that Earley's Algorithm is a universal parsing algorithm and can handle any arbitrary CFG, it is essential that the issue of time and space requirements is addressed. Unlike LL(k) and LR(k) parsing algorithms, Earley's Algorithm is not table driven. All possible parses of a string are carried along simultaneously at parse time.

Earley proves in his thesis that his algorithm has a bound of the form $Cn^3+O(n^2)$ on the number of steps required to parse a string of length n for any lookahead k. Thus, this algorithm is an $O(n^3)$ recognizer in general. However, a large class on grammars can be parsed in time O(n); they seem to include most unambiguous grammars. Space requirements for Earley's Algorithm are $O(n^2)$ in general. However, as with the time requirements a large class of grammars can be parsed in space O(n), possibly all unambiguous grammars.

Obviously, the actual time and space used would depend on the particular implementation of the algorithm. Earley's thesis merely shows that it is possible to achieve the stated time and space bounds. He also shows that the same time bounds hold for the parser based on his algorithm, though for the parser the space requirements are $Cn^3+O(n^2)$ (to hold the derivation graph).

The tradeoff is between the power and the performance of the various algorithms. The LL(k) and LR(k) algorithms are both time O(n) algorithms (with a lower coefficient of n than Earley's Algorithm). Because of their efficiency they have been used in numerous compiler development tools. However, they are not as powerful as Earley's Algorithm. These algorithms cannot handle ambiguous grammars, or a grammar which requires a lookahead greater than k.

4. Using the Universal Parser Generator

This section provides documentation about the usage of PEG. The syntax for specifying the grammar to PEG is almost identical to that provided by YACC. This syntax was adopted because of its popularity and ease of use.

4.1. Conceptual Input to PEG

PEG provides a general tool for imposing structure on the input to a computer program. The user prepares a specification that includes:

- A set of rules to describe the syntax of the input (productions)
- Code to be invoked when a rule is recognized (semantic actions)

PEG then turns the specifications into a C language routine that examines the input stream. This routine, called a parser, works by calling a lexical analyzer. The lexical analyzer extracts tokens from the input stream. Tokens are the terminal symbols defined by the grammar and used in the productions. When one of the productions is recognized (reduced or determined to be part of the derivation of the input string), the semantic action supplied for this production are executed. Semantic actions are fragments of C language code. They can communicate values of the attributes of terminal and nonterminal symbols, and make use of the values communicated by other semantic actions.

4.2. Basic Specifications

Every specification file theoretically consists of the three sections: the declarations, (grammar) rules, and subroutines. The sections are separated by double percent signs, %%. A full specification file looks like

declarations
%%
rules
%%
subroutines

when all the sections are used. The subroutines section is optional. User comments may appear anywhere in the file. Comments are C language comments limited to a single line.

4.3. Declaration Section

Several things go into the declaration section. PEG maintains a table of tokens and nonterminal symbols along with various information associated with them. These tables are essential to the implementation of PEG. It is possible for the user to expand the size of these tables by including the following statements as the first statement(s) in the declaration section

%termtable *size*

%nonttable *size*

where *size* is an integer number. It is advisable to use prime numbers for the sizes because it makes hashing more efficient (see Section 5.3). These two statements are optional.

Next, the user can include a statement that explicitly tells the parser what symbol to use as the root (start symbol) of the grammar. This is done by the statement

%start *name*

where *name* is a nonterminal symbol. This statement is optional. By default, the nonterminal on the left-hand side of the first production in the grammar is taken to be the root of the grammar.

Next, the user can explicitly choose between using a lookahead of 0 or 1. This is done by the statement

%lookahead *number*

where *number* is either 0 or 1. This statement is optional. By default, a lookahead of 0 is used.

Communication between the parser and the lexical analyzer is done by "agreeing" on a data structure that both routines can access. This data structure is defined to the parser by the statement

```
%union
{
   body of union (as in the C language) ...
}
```

This usage is identical to YACC. The union declaration statement can appear before the lookahead declaration statement in the declaration section. The union declaration is optional; if the union is not explicitly declared by the user then PEG creates a union with one integer field 'ival' in it.

Next, the token (terminal symbols) names are declared. Names representing tokens must be explicitly declared. Associated with each token is an integer number that uniquely identifies that token. Tokens are the means of communication between the lexical analyzer and the parser. The lexical analyzer scans the input stream and returns an integer value (and possibly other values) representing the type of token found to the parser; this integer value tells the parser which token was scanned. The way to declare tokens is the following

%token <*type*> *name number*

in the declaration section. The *name* is the token identifier and the *number* is the integer assigned to that token. Every name used in the productions of the grammar, but not declared as a token, is assumed to be a nonterminal symbol. The *type* is one of the fields from the union declared by the *%union* statement discussed earlier; it is optional. The *type* of the token is used to determine the correct type of the attribute of the tokens (see Section 4.4.1). The *number* may be omitted from the declaration; tokens without a *number* are assigned a default token number, starting at 256.

If more than one token is of the same *type* and default token numbering is being used, then these tokens can be conveniently declared in the following way

%token <*type*> *name1 name2 ...*

where each of the token in the list is assigned a different default token number, and every token declared in this list has the same *type*; the *type* declaration is optional. Any number of statements declaring the tokens may be given in any of the two formats presented above.

The nonterminals used in the productions may have a *type* associated with them. These are declared to PEG in the following manner

%type <*type*> *name1 name2 ...*

where the names are the names of nonterminal symbols, and the *type* is one of the fields declared in the *%union* statement. Any number of *type* statements may be given.

Any other C language code can be inserted at this point into the declaration section. For example, the user may want to define variables to be used by the semantic actions. C Language declarations and definitions can appear in the declarations section. The syntax for specifying these declarations and definitions is the following:

%{ C language declarations and definitions }

These declarations and definitions have global scope, so they are visible inside the semantic actions.

4.4. Rules Section

The rules section is made up of one or more grammar productions. A production has the form

```
A   :   BODY   ;
```

Note that this format has a close correspondence to CFGs. The symbol A represents a nonterminal symbol, the ':' corresponds to the '→' of CFGs, and BODY is a sequence of tokens and nonterminals (possibly separated by the vertical bar '|').

Names of terminals and nonterminals may be of any length and may be made up of letters, digits, and underscores although the first character of a name must be a letter. Uppercase and lowercase letters are distinct. The names used in the body of a production may represent tokens or nonterminal symbols. They are differentiated by explicitly declaring the tokens in the declarations section as previously described.

If there are several productions with the same left-hand side nonterminal symbol, the vertical bar '|' can be used to avoid rewriting the left-hand side. In addition, the semicolon at the end of a production is dropped before a vertical bar. Thus, the grammar productions

```
A   :   B   C   D   ;

A   :   E   F   ;

A   :   G   ;
```

could also be given to PEG as

```
A   :   B   C   D

    |   E   F

    |   G

    ;
```

by using the vertical bar. It is not necessary that all grammar productions with the same left side appear together in the rules section, although it makes the input more readable and easier to change. If a nonterminal 'A' matches the empty string (λ), this can be indicated by the production rule

```
A   :       ;
```

where the right-hand side of the production contains only white spaces followed by a semicolon.

4.4.1. Semantic Actions

Semantic actions may be given as part of a production. If given, they must appear at the end of a production. The attributes of the symbols of a production are accessed in exactly the same manner as in YACC (see below). The semantic actions are simply C language statements that are executed each time the production to which they are attached is reduced during parsing. For example, assume that 'semi' is a token returned by the lexical analyzer during parsing. If the following production is part of the grammar

```
A   :   semi
    {
        printf("Found a Semicolon!");
    }
    ;
```

and this production is reduced, the action routine will be invoked and the message will be printed. At execution time, if the parser generated by PEG detects that the string being parsed will have multiple derivation trees then it stops executing the semantic actions associated with the productions of the grammar. However, the parsing continues.

The dollar sign symbol, $, is used to facilitate communication of information between the semantic actions and the parser. The pseudo-variable $$ represents the value of the attribute of the nonterminal symbol on the left-hand side of a production. The pseudo-variables $1, $2, ...$n represent the values of the attributes of the symbols on the right-hand side of a production. $k refers to the attribute of the kth symbol on the right-hand side of a production. For example, if the production is

```
A  :  B  C  D  ;
```

then $2 refers to the attribute of `C`, and $3 to the attribute of `D`.

Consider a grammar for describing arithmetic expressions. Let one of the productions of the grammar describe addition of two integer numbers. Suppose that the lexical analyzer (see Section 4.9) scans the input and returns to the parser the integer value of the numbers (a '34' in the input returns an attribute of integer 34) as the attribute of the token 'intval'. Furthermore, suppose that the token 'plus' is returned when a '+' is seen in the input. Now, productions that look like

```
E           :   NUMBER plus NUMBER
            {
                printf("Answer = %d",$1+$3) ;
            }
            ;

NUMBER      :   intval
            {
                $$ = $1 ;
            }
            ;
```

will print the result of adding two integers. Notice that if a derivation tree (graph) of the string being parsed was constructed, then the attributes of a node in the tree would be defined in terms of the attributes of its children nodes. The attribute evaluation would be bottom-up, from the leaves to the root. These types of attributes are called **synthesized attributes** and are also the mechanism used by YACC.

PEG has some significant differences in its action routine specifications compared to YACC. In YACC, semantic actions do not have to be placed at the end of a production. They may appear between two symbols on the right-hand side of a production. This sometimes causes parsing problems in YACC. In PEG semantic actions are not allowed in the middle of productions. PEG places one other restriction on semantic actions. No 'return' statement is allowed in the semantic actions. Instead, a special function is provided to accomplish the same effect (see Section 4.4.2).

In YACC, it is legal to have productions of the form

```
A   :   A   '+'   B;
```

where instead of declaring a token for the '+' symbol, it is used directly in the production. PEG does not allow this. In PEG all tokens have to be explicitly declared.

The algorithm used by PEG encourages the so called "left-recursive" productions over "right-recursive" productions. Left-recursive productions have the form

A: A rest_of_rule;

while right-recursive productions have the form

A: rest_of_rule A;

Even though both forms of productions are "legal" (acceptable to PEG), right-recursive rules may result in less efficient parsers (also see Section 9).

4.4.2. Special Functions

PEG provides four special functions that can be part of the semantic actions of any production. These functions enable the user to exploit the special functionality available as a result of using Earley's Algorithm.

earley_stop(): This function is like a 'return' statement. It causes the parser to immediately stop all parsing activities, and make a top-level return to the function that called it.

EarleY_delete_production("p1 p2 ... pn"): This routine deletes productions **p1, p2, ..., pn** from the grammar. Any punctuation can be used as a delimiter between the **p1** through **pn**. The productions and their associated numbers can be referenced from the file earley.tab.h (see Section 4.8). This function returns a 1 if all the deletions are successful, a 2 if some production number is out of range, and a 3 if some production has already been deleted by a previous action.

EarleY_add_production("LHS1: RHS1; ..."): This function adds new production(s) to the grammar. **LHS1** is any nonterminal symbol (new or old). The **RHS1** is like any RHS of a production rule consisting of nonterminals (new or old) and tokens (declared in the original grammar). There can be multiple alternatives in RHS, each separated by the vertical bar '|'. Each alternative may also have **Transfer Semantics** associated with it. Transfer semantics is a specification of the semantics for this new production in terms of the semantics of one of the original productions. The specifications of the transfer semantic actions (if any) of the new production are recast in terms of the semantics of an original production. The semantic specification of new productions always has the format

$$\{Number; \$A_1 = \$A_2; \ldots \$A_n = \$A_{n+1};\}$$

where *Number* is the production number of a production in the original grammar, whose semantic actions will be executed for this new production. A_1 through A_{n+1} are reference reassignments. For example, the transfer semantics '{1;$3 = $2;}' means the following to the parser: For this new production do the same semantic actions as production 1 of the original grammar. However, whenever the attributes of the third symbol were being referenced in the semantic actions, now refer to the attributes of the second symbol. No restriction is placed on these reassignments (see Section 6.1). This function returns a -2 if there is a syntax error in one of the production specifications, a -1 if a production has a token on the left-hand side, a 0 if '$' appears anywhere in the new production (except the semantic actions), and a 1 if no errors occur.

EarleY_delete_add_production(S_1, S_2): This function combines deletion and addition of productions in one function call. S_1 is the string containing the numbers of the productions to be deleted, and S_2 is the string that gives the productions that are to be added (see **EarleY_add_production()**, and **EarleY_delete_production()** above). The values returned by this function are a union of the values returned by its composite functions.

These four special functions may appear anywhere in the semantic action of a production. One final note about these functions. The three functions **EarleY_delete_production()**, **EarleY_add_production()**, and **EarleY_delete_add_production()** alter the original grammar, and if a lookahead of 1 is being used then they cause the lookahead to be recomputed. This lookahead computation can significantly increase the parse time. Some careful programming can help to minimize the adverse effect on the parse time. For example, if one production is to be deleted and another one added at the same point in the parse, then **EarleY_delete_add_production()** is the most efficient way to do it.

Consider the grammar for well-formed parentheses, G2.1. Let the terminal symbols '(' and ')' have the token names 'LP' and 'RP', and let the token numbers be 7 and 8 respectively. Figure 4.1 gives the modification to the data structures if a new production is added or an old production is deleted.

4.5. Subroutines Section

In this last section of the specifications, the user can include any C code that he/she wishes to be available to the parser for use by the semantic actions. There is no restriction placed on the content of this section and any errors are reported at compile time by the compiler (hopefully).

4.6. Error Messages

The error messages produced by PEG are meant to be explicit and self-explanatory. These error messages are divided into two categories, **Warnings** and **Errors**, by level of severity.

Warnings are messages indicating a non-fatal error, and usually parsing continues after a warning. On the other hand, an error message indicates a severe error condition. PEG tries to continue parsing whenever possible, but usually an error terminates the parsing.

4.7. Naming Convention

The parser generated by PEG contains data and supporting routines to implement the parser. As a naming convention, all routines whose names start with the string 'EarleY' are external routines. The user can modify the name of any variable or routine whose name does not start with the string 'EarleY'. To avoid name clashes, the user should also avoid creating new routines, macros, types, or variables with names starting with the strings 'EarleY' or 'earley'. Needless to say, the user should avoid using names that might cause a name clash. See Section 6.3 for more on the naming convention.

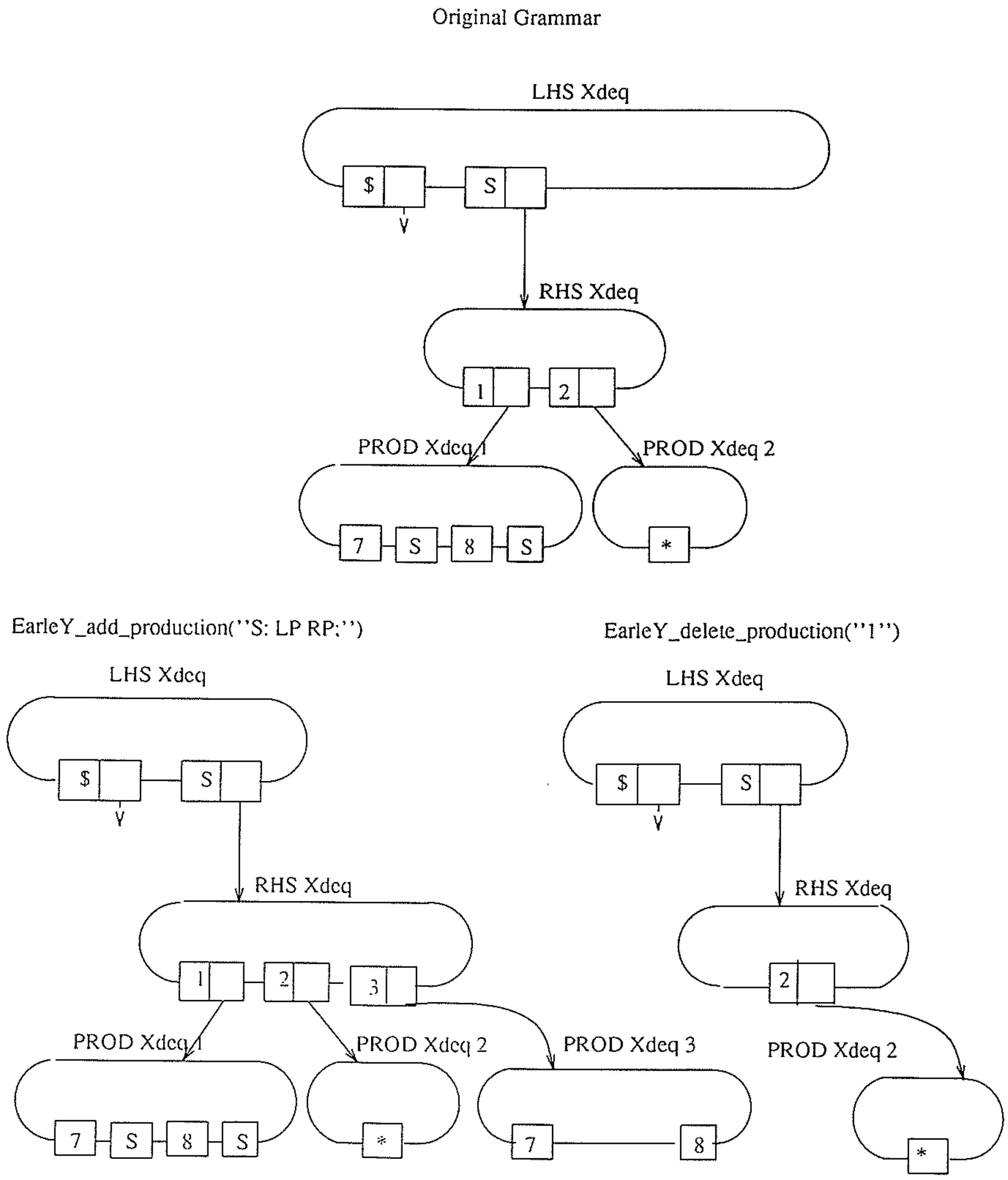


Figure 4.1: Modifications to Original Grammar

4.8. Output Files

PEG scans the grammar specification file and generates two files from the grammar specification.

> **earley.c**: This file contains all the user routines and other supporting routines needed to implement the parser.
>
> **earley.tab.h**: This file contains the various definitions specified by the user. For example, it contains the token definitions, translated to the `#define` statements of the C language. It also contains (as comments) all the productions of the original grammar and their internal numbering; this numbering is essential to using the special functions provided by PEG (see Section 4.4.2).

4.9. Interface to Lexical Analyzer

When the parser created by PEG requires the next token it calls the lexical analyzer supplied by the user. This lexical analyzer is assumed to follow certain conventions. It must return a 0 when it encounters the end-of-file. Otherwise it should return an integer that identifies the token that was scanned. Note that these are the conventions followed by the lexical analyzer created by the program LEX under the Unix operating system.

It may be that the user wants some other attributes of the tokens to be computed by the lexical analyzer and passed to the parser. The union structure declared in the **Declarations Section** is the mechanism used to communicate these attributes to the parser. The values of these attributes are explicitly set by the user in the LEX program that he/she provides to PEG. The file **earley.tab.h** may be consulted to see which are the legitimate tokens expected by the parser. The name of the lexical analyzer must be **earley_userlex()**.

The union structure declared in the **Declarations Section** is also declared in the lexical analyzer. Space is allocated for this union in the lexical analyzer and a variable **earley_userlval** is declared to be a union of this type.

4.10. Invoking PEG

PEG is invoked by the following command

```
peg [FLAGS] filename
```

where *filename* is the name of the grammar specification file. The FLAGS allow the user control over the execution and output behavior of the parser generated by PEG. The FLAGS are the following (all of them are optional):

> -CYCLE0: Do not check for cycles at parse time.

-GRAPH: Print the unattributed derivation graph to the default output device.

-PARSE0: Do not perform semantic actions or build the derivation graph; just recognize.

-PRINT0: Do not print anything at the end of the parse.

-PRINT1: Print whether the string is in the language (default).

-PRINT2: Print detailed information about the parse (State Sets and States).

-PRUNE: Perform garbage collection on unneeded states during parse.

By default, the only flag used is '-PRINT1'.

The parser generated by PEG returns a pointer to the attributed derivation graph of the string parsed. If the string had multiple derivations or cycles in the derivation graph, then the derivation graph that is returned is not attributed. The parser also writes out an unattributed version of the derivation graph to the file 'earley.graph'. If the parser name has been changed by global substitution, then the derivation graph file name is also changed accordingly (see Section 6.3). This unattributed derivation graph can be recreated and printed by using two routines supplied with the PEG library (see Section 7.2). Note that if the string being parsed is not a member of the language defined by the CFG on which the parser is based, then the parser returns a 'EarleY_MEM_NULL' pointer (see Section 8) at the end of the parse.

4.11. The Main Procedure

PEG does not provide a main procedure with the parser; the user can write any main procedure that he/she wishes. The name of the parser generated by PEG is **earley_parse()**. Before calling the parser the user must initialize variables and data structures by calling the routine **earley_init()**. For example, suppose that a parser has been created using PEG. Then the following is the smallest main procedure that can be written to call the parser:

```
#include  "earley.tab.h"
main()
{
       EarleY_NODE_TYPE p;

       earley_init();
       p = earley_parse();
       /* EarleY_node_destroy(&p); */
}
```

Note that the main routine given above does not deallocate the space used by the derivation graph; if the commented statement in the main routine was compiled, then the space used by the derivation graph would be deallocated.

4.12. Compiling and Linking

The file, contains the parser generated by PEG. Suppose that the **main** program is in the file 'main.c'. Furthermore, suppose that the lexical analyzer created by the user for the PEG parser is in file 'lexical.c'. The executable file, can be created from the main program, the PEG parser, and the lexical analyzer, by the following compilation steps:

```
cc -c main.c

cc -c earley.c

cc -c lexical.c

cc -o earley main.o earley.o lexical.o -learley
```

Notice that all the compiled modules are linked with the PEG library. The PEG library contains routines that are necessary to implement the parsers generated by PEG. The routines in this library are the **static** routines; they are the same for all parsers generated by PEG. On the other hand, the routines in the file 'earley.c' are the **dynamic** routines; these depend on the grammar which is used to generate the parser. If the user wants the executable to be linked with other libraries or modules, then these must be specified also. The code given above is only meant to illustrate the process that is necessary to successfully compile the code generated by PEG.

5. Implementation of PEG

This section describes the major design features of PEG. The data structures used, and the efficiency measures incorporated into the design of PEG are discussed.

5.1. Abstract Data Types (ADTs)

One of the most important decisions regarding the implementation of any software is the choice of the underlying data structures. In the implementation of PEG, the data structures were chosen on the criteria of reliability, flexibility, and the ability to easily extend and modify the software. This section gives description of the data structures used by PEG, and discusses the advantages of these data structures.

Suppose one wants to implement a stack data structure in a programming language like C. If the stack is implemented using an array, then all the elements that are *pushed* on the array must be of the same *type*. Implementing a generic stack data structure on which arbitrary data types may be pushed becomes difficult. Of course, one may implement the stack using an array whose elements are some pre-defined structure or union. Still, the programmer would have to predict all the possible data types that may be pushed onto the stack. One would like a generic data structure that could hold any data type. The obvious solution is to push not the object onto the stack but a pointer to it. This pointer in turn could point to any arbitrary data object. The left stack in Figure 5.1(a) can hold homogeneous data objects and the one on the right can hold arbitrary data objects (the '?' indicates a single data type and the '*' multiple data types) .

An ADT is a data structure that is implemented using pointers into memory. The programmer describes the data structure to the system by defining the structure of the ADT and the access functions that are applicable on this ADT (Figure 5.1(b). For example, a PAIR_TYPE may be described as an ADT that holds (points to) two objects along with the access functions to access the two objects.

The programmer is required to define certain operations on ADTs to the system. Routines that create, destroy, print, and copy an ADT must be defined. Besides these required routines, the programmer can define any arbitrary functions on an ADT.

At execution time when an ADT is created the system creates a **tag** that identifies the object type and attaches the tag to the ADT. By accessing the tag of an ADT the system can determine how to manipulate the ADT. This tagging mechanism is completely transparent to the user and critical to the implementation of ADTs. Since the tag is created at execution time and not at compile time, the ADTs are generic in the sense that an ADT can reference any other ADT. This notion of defining data objects and the operations application on them is called **object oriented programming** (OOP). The ADT system used by PEG is not a true OOP methodology because **inheritance** is not supported. See Section 8 for more details about specific ADTs.

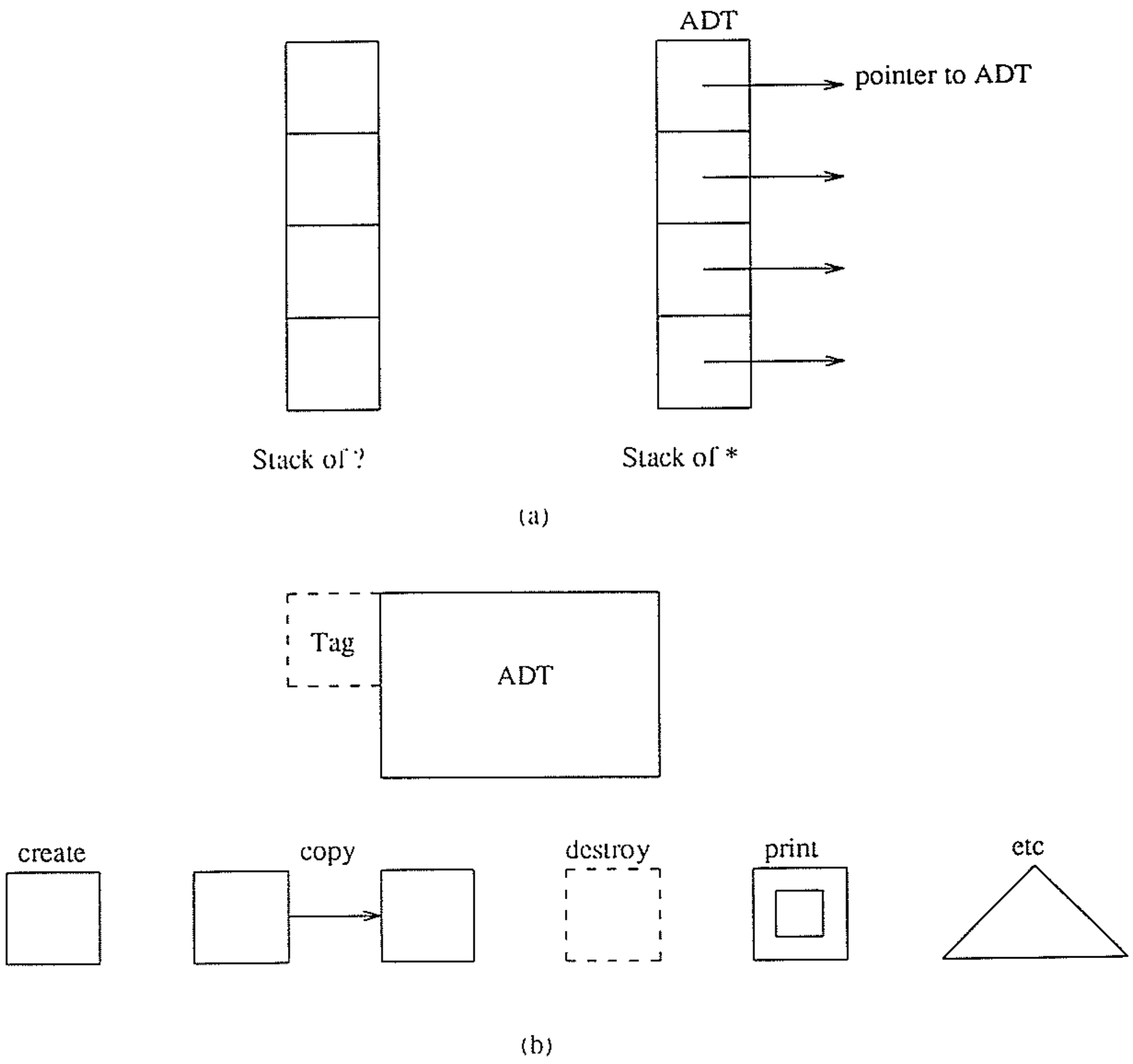


Figure 5.1: Abstract Data Types (ADTs)

5.2. ADT Representation of Grammar

Consider the grammar for well-formed parentheses, G2.1. PEG would augment this grammar by adding the production '$\$ \rightarrow S\ \nabla$' as the first step in the generation of the parser. Therefore, the grammar that would be used to create the parser would be the following

$$\$ \rightarrow S\ \nabla$$

$$S \rightarrow (\ S\ )\ S$$

$$S \rightarrow \lambda$$

Grammar 5.1

The conceptual representation of Grammar 5.1 in the ADT paradigm is given in Figure 5.2. In **LHS Xdeq** (Xdeq stands for "extended double ended queue"), all the nonterminals of the grammar are represented by the left element of one and only one **Pair**. The left element of a Pair in LHS Xdeq contains the string representation of a nonterminal, and the right element is a pointer to a **RHS Xdeq**. In RHS Xdeq a list of Pairs is held, the left element of a Pair contains the integer production number of the alternative that the Pair represents, and the right element is a pointer to a **PROD Xdeq**. A PROD Xdeq represents the right-hand side of a specific production, and its elements are made up of terminal (**Term**) and nonterminal (**Nont**) symbols.

Suppose that the terminal symbols '(' and ')' are represented by the tokens 7 and 8 respectively. That is to say, the user has explicitly declared these tokens in the **Declarations Section** of the PEG specification file. By convention, the end-of-file is represented by the reserved token number 0 (see Section 4.9). Also by convention, in PEG the empty string (λ) is represented by a nonterminal whose string representation is "*". Given these conventions, the actual representation of Grammar 5.1 is given in Figure 5.3. This is how PEG stores the input grammars in the ADT paradigm. In this representation, the placing of the elements in the various Xdeqs depends entirely on the order in which they are encountered in the grammar file. For example, if the alternatives of a nonterminal 'A' are written before the alternatives of another nonterminal 'B', then 'A' will appear before 'B' in LHS Xdeq. During parsing if a new nonterminal is inserted into the grammar, then it will be inserted at the right end in the LHS

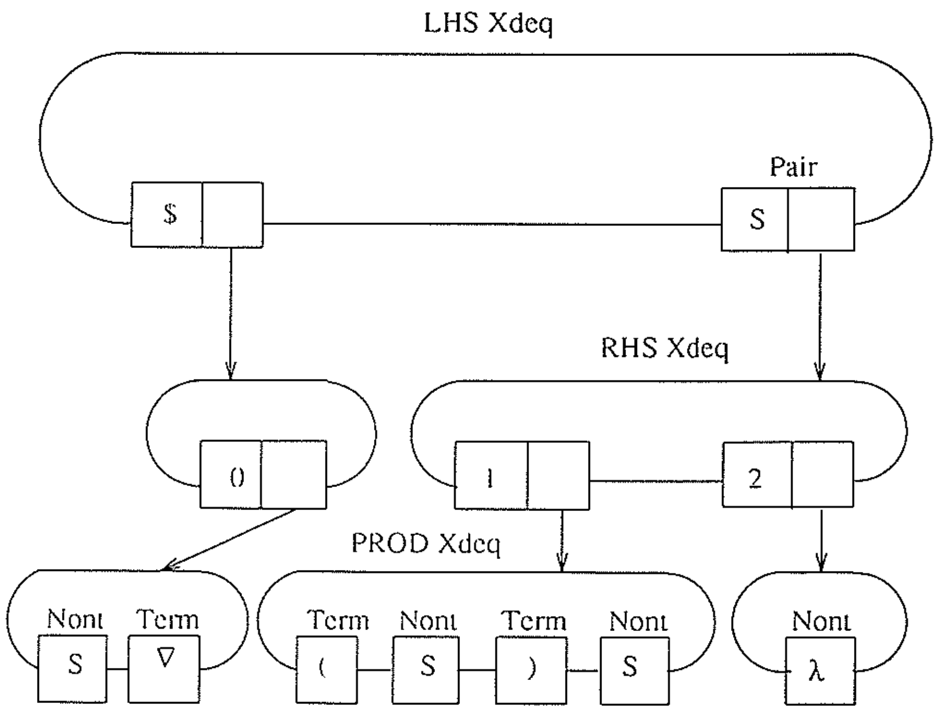


Figure 5.2: ADT Representation of Grammar 5.1 (Conceptual)

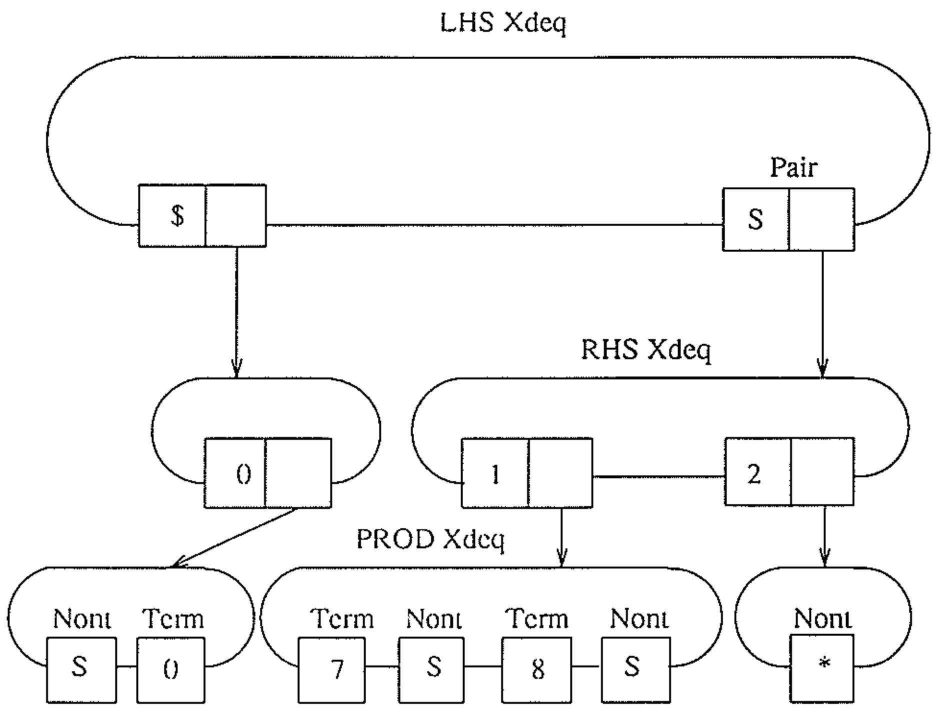


Figure 5.3: ADT Representation of Grammar 5.1 (Actual)

Xdeq. If a new alternative is added for an existing nonterminal, it is added at the end of the corresponding RHS Xdeq. This provides a natural representation of the grammar as a data structure.

5.3. Efficiency Considerations

The three operations (Predictor, Scanner, and Completer) that implement Earley's Algorithm access the productions in the grammar being used (see Section 3). To cut down the search time required to access a nonterminal and its alternatives or to access a particular production, a **hash table** is maintained. The associated hashing function takes the string representation of a nonterminal and returns the integer hash table position. Among other information this hash table contains a pointer to the nonterminal in the LHS Xdeq. Accessing a particular nonterminal in the LHS Xdeq during parsing, is done in two steps. In Step 1 the hash function is applied to the nonterminal name, giving the index into the hash table. In Step 2 the pointer to the nonterminal in LHS Xdeq is read from the hash table. This procedure eliminates searching for the nonterminal in LHS Xdeq, making the required search time much less dependent on the size of the grammar. A conceptual picture of this two step search procedure is given in Figure 5.4. A similar hash table is maintained for the terminal symbols.

In a RHS Xdeq the productions are stored in ascending order by the production number. This information is utilized when searching for a specific production. In Figure 5.5, suppose that production 5 is to be deleted from the grammar at some point during the parse. The special function **EarleY_delete_production** can stop searching RHS Xdeq 2 as soon as production 8 is

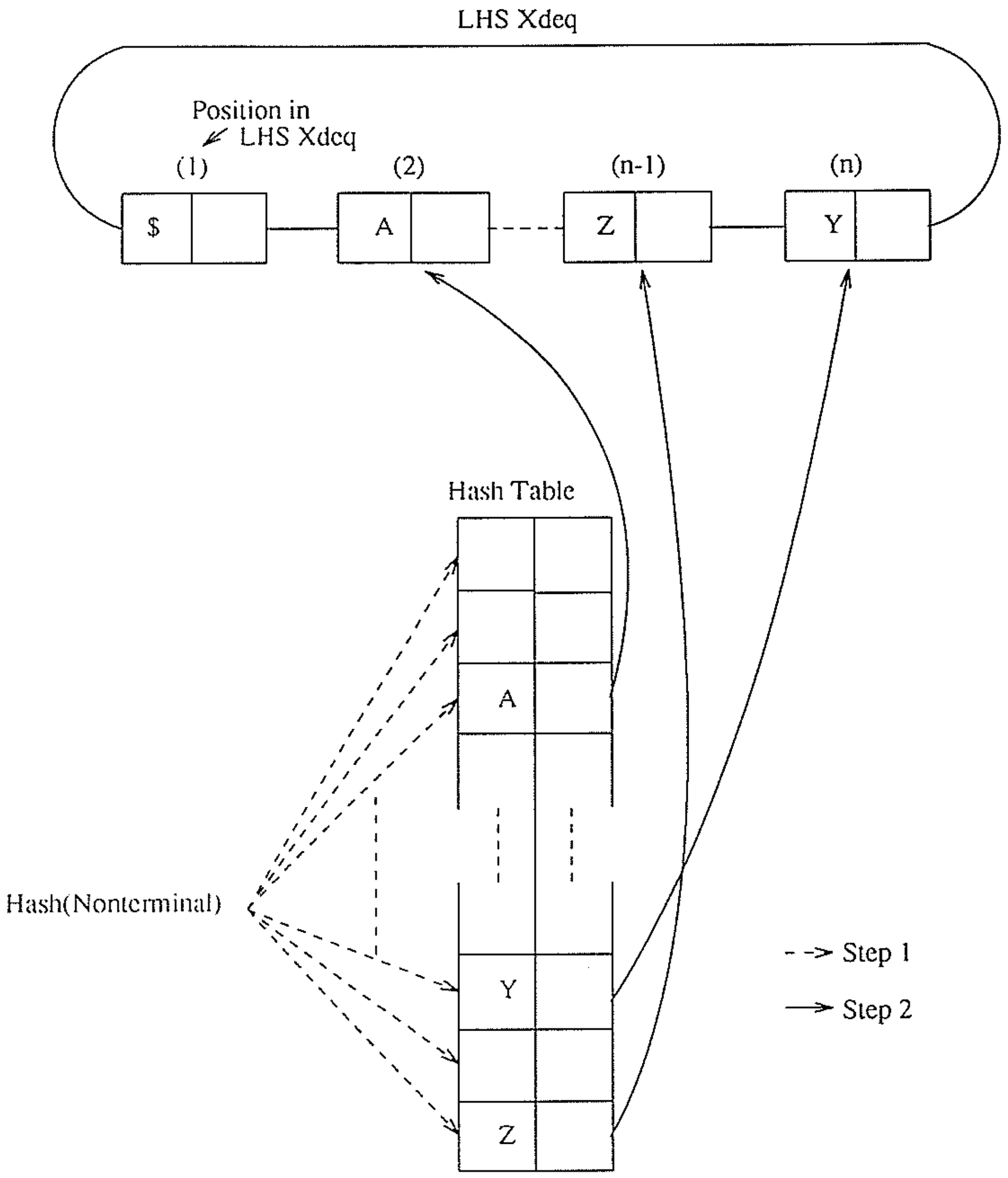


Figure 5.4: Accessing a Nonterminal and Its Alternatives

encountered (searching from left to right), and proceed to the next RHS Xdeq.

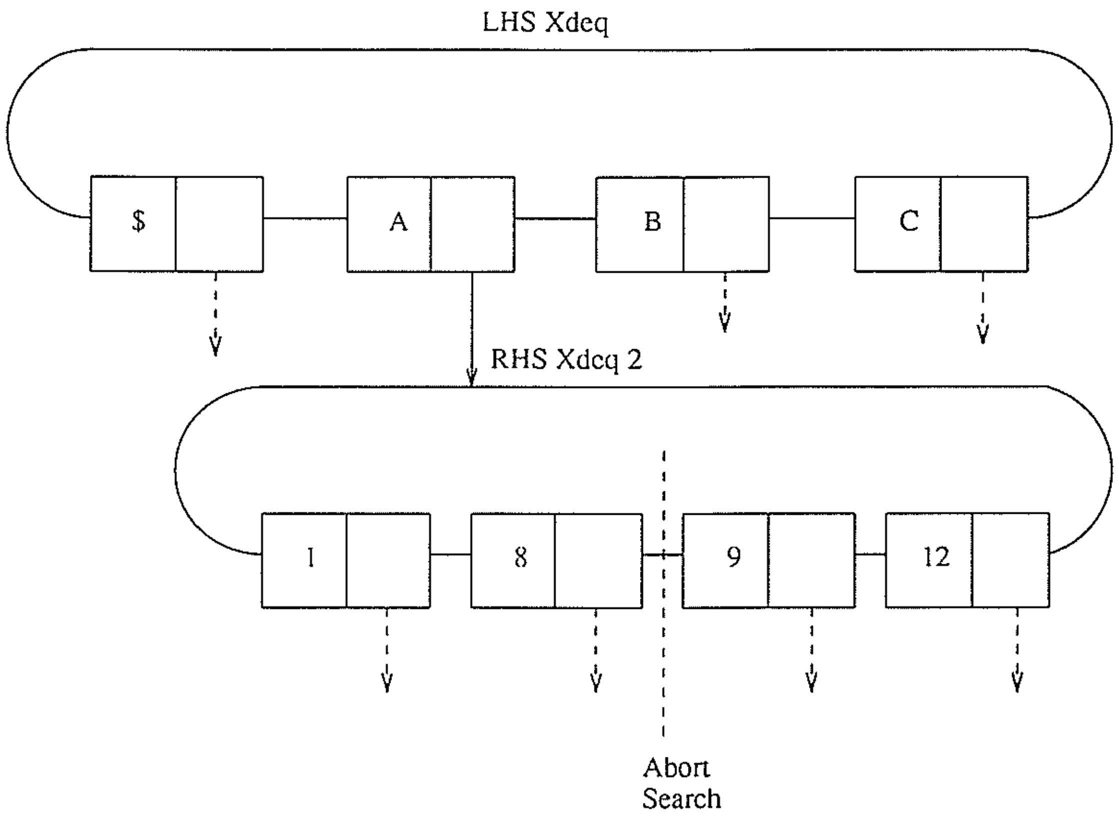


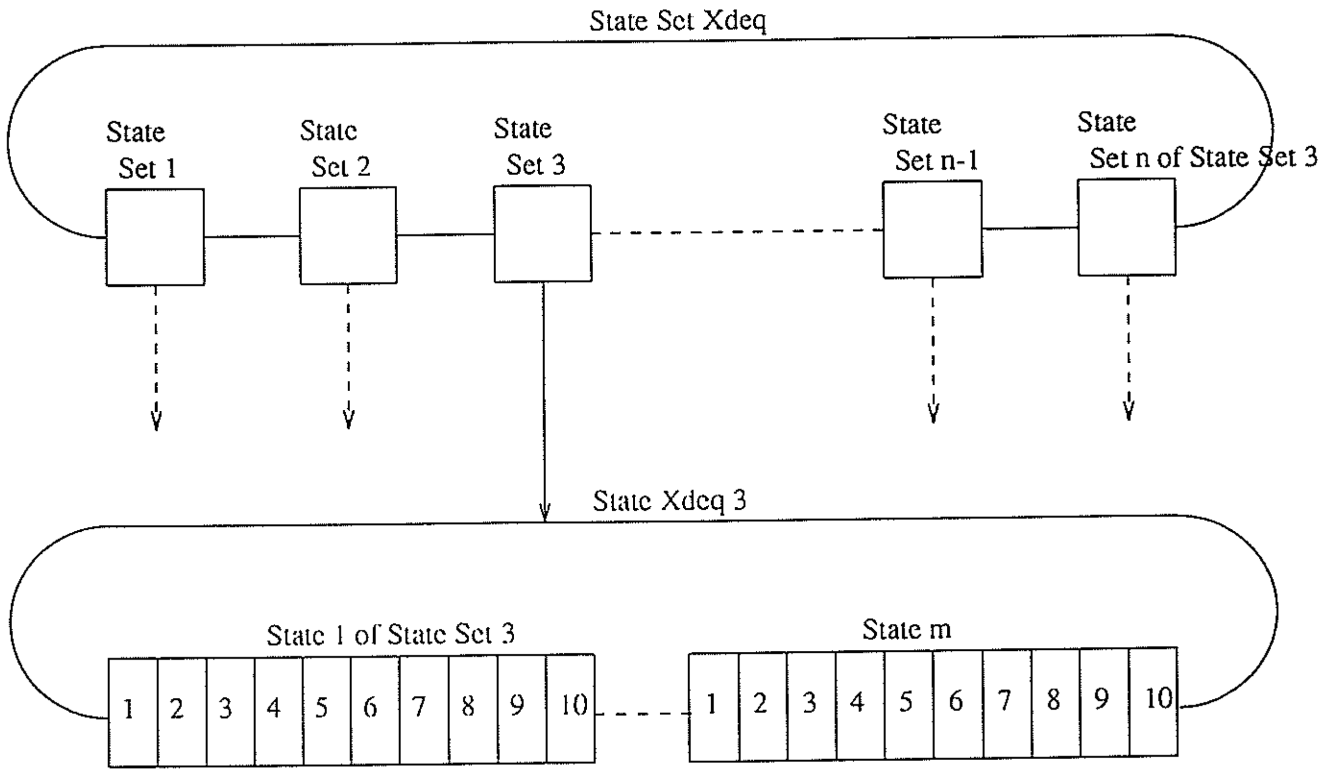
Figure 5.5: Searching for a Specific Production

5.4. ADT Representation of State Sets and States

As dictated by Earley's Algorithm, State Sets and States are created as the input string is parsed. Referring to Figure 5.6, **State Set Xdeq** contains all the State Sets that are created during parsing. As a new State Set is created, it is inserted at the right end of State Set Xdeq. Each State Set is a pointer to a **State Xdeq**, which in turn contains all the States belonging to that particular State Set.

In a recognizer based on Earley's Algorithm, each State contains four entries. On the other hand, a parser based on Earley's Algorithm requires more space (e.g. for the derivation graph). In PEG a State has ten components, as shown in Figure 5.6. Components 1-4 are the same as specified by Earley for the recognizer. Components 5 and 7 are used by PEG to implement synthesized attributes, and Component 6 is used to determine which semantic action to execute (see Section 4.4.1). The other components are used to generate the derivation graph, and achieve efficiency in parsing.

Each of the three operations (Predictor, Scanner, Completer) require a State to be in a particular form before the operation can be applied to the State. To find the States that are of the correct form in a State Set requires a linear search of the State Set. Instead of this costly search, PEG does the following: whenever a new State is inserted into a State Set, depending upon the form of this State it's position in the State Set is inserted into one of three lists, **Predictor Xdeq**,



1:
Production number

2:
Position in production:
number of RHS symbols scanned

3:
Pointer to State Set where predicted

4:
Xdeq of lookahead symbols

5:
Xdeq of Attribute of RHS symbols

6:
Pointer to Completer state generating this state

7:
Attribute of LHS symbol

8:
Position of this state
(State Set, State)

9:
Applicable operation
(Predictor, Scanner, Completer)

10:
Indicator for cycle

Figure 5.6: State Sets and States

Scanner Xdeq, or **Completer Xdeq**. The three operations check the appropriate list to find States that are of the correct form: search time is reduced (compared to a linear search).

5.5. Deferred Semantic Actions

Suppose that during parsing a particular production has to be reduced (i.e. a Completer action). In LL(k) and LR(k) parsers the semantic actions associated with this production can immediately be executed. In LL(k) and LR(k) parsers every time a production is reduced it is indeed going to be in the final parse. On the other hand, Earley's Algorithm carries out all the possible parses of a string under a grammar. Therefore, it is possible that the parser generated by PEG will make many parsing decision (Completer operations) which later turn out to be extraneous. For this reason, some semantic actions may have to be deferred until the parser can determine whether the parsing decision is indeed going to be part of the final parse.

If a state set contains only one final state (see Section 3), then that state is indeed going to be part of the final parse. Otherwise, if there are two or more final states, then the states that they produce by the completer is marked and the parsing continues. Later, as soon as a state set is reached that contains only states which were produced by one of those final states, then that final state is the one which represented the correct parsing decision, and is going to be in the final parse. As soon as it is determined that a final state is going to be in the final parse, the semantic action associated with it can be executed. Of course, it may be that many semantic actions have been deferred, beginning at the point in the parse where local ambiguity first appeared. All these semantic actions are executed in the correct order once that ambiguity is resolved. If the local ambiguity is never resolved, or as soon as it is determined that the string being parsed has multiple derivation trees, semantic actions are not executed.

This feature gives PEG a unique capability. Deferred semantic actions mean that a parsing decision is postponed until later; this is the same (conceptually) as using an unbounded lookahead (see Section 2.2). Therefore, the parser generated by PEG has the advantage of having an unbounded lookahead without actually computing any lookahead sets.

6. Examples of the Unique Features of PEG

This section gives three examples that illustrate the unique features of PEG. The first example illustrates the implementation of syntactically extendible languages. The second example shows how deferring semantic action enables the parsers generated by PEG to parse CFGs with unbounded lookahead. The third example shows how multiple parsers generated by PEG can be linked together in a single executable. In all the examples, the lexical analyzer specification is given using LEX.

6.1. Syntactically Extendible Languages

The following example illustrates the ability to change the syntax of the language being parsed at execution time in a parser generated by PEG. It also shows how one can transfer the semantics of an original production of the grammar to a new production that is added to the grammar (see Section 4.2).

Suppose a language contains infix arithmetic expressions. Furthermore, suppose the application should allow the user to change the language at execution time so that the language contains only postfix arithmetic expressions. For example, if the following input is given to the application:

```
13+5*7;
CHANGE;
13 5 7 * +;
```

then the desired output would be:

```
48
Switching to Postfix
48
```

That is, when the keyword 'CHANGE' is encountered in the input then the language should be changed from infix arithmetic expressions to postfix arithmetic expressions.

The above application can be easily implemented in PEG. The following grammar specification file:

```
%lookahead 0
%union {
char *sval;
int ival;
}
%token <sval> DECNUM
%token LP
%token RP
```

```
%token SEMI
%token PLUS
%token MINUS
%token MULT
%token IDIV
%token CHANGE
%type <ival> Prog State Exp Fact Term Change Top
%%
Prog:State Prog | ;

State:Top SEMI | Change SEMI ;

Exp:Exp PLUS Term { $$ = $1 + $3; }
  | Exp MINUS Term { $$ = $1 - $3; }
  | Term { $$ = $1; }
  ;

Term:Term MULT Fact { $$ = $1 * $3; }
  | Term IDIV Fact { $$ = $1 / $3; }
  | Fact { $$ = $1; }
  ;

Fact:DECNUM { $$ = earley_eval_int($1,10,"0123456789"); }
  | LP Exp RP { $$ = $2; }
  ;

Change:CHANGE
  {
  printf("\nSwitching to Postfix");
  earley_delete_production("5 6 7 8 9 10 11 12");
  earley_add_production("Exp: Exp Exp PLUS {5; $3 = $2;};");
  earley_add_production("Exp: Exp Exp MINUS {6; $3 = $2;};");
  earley_add_production("Exp: Exp Exp MULT {8; $3 = $2;};");
  earley_add_production("Exp: Exp Exp IDIV {9; $3 = $2;};");
  earley_add_production("Exp: DECNUM {11; $1 = $1;};");
  }
  ;

Top:Exp { printf("\n%d",$1); }
  ;
%%
```

and the following lexical analyzer:

```
D [0-9]
%{
```

```
#include "earley.tab.h"
%}
%%
"(" {return(LP);}
")" {return(RP);}
"+" {return(PLUS);}
"-" {return(MINUS);}
"*" {return(MULT);}
"/" {return(IDIV);}
";" {return(SEMI);}
{D}+ {earley_userlval.sval=concat(yytext,"");return(DECNUM);}
"CHANGE" {return(CHANGE);}
[ \t\n]+ {}
%%
```

implement the desired application. In the specification files given above, function **concat(s1,s2)** makes a copy of string s1, appends a copy of string s2 at its end, and returns a pointer to the resulting string. Function **earley_eval_int(s1,base,bdigits)** takes a string of digits s1 and computes the numeric value of the string by interpreting the digits in a user-specified number base. The second parameter to the function specifies the number base, and the third parameter gives the digits of that number base.

Some caution has to be exercised in using the extendible language feature of the parser generated by PEG; deferred semantic actions (see Section 5.5) means that some semantic actions may not get executed at the time (parse time) that one would intuitively expect them to.

The example presented here is static in the sence that the new productions to be added to the grammar, and the productions to be deleted from the grammar, are fixed by the writer of the PEG specification file. A more robust application would allow the user to specify what the changes to the syntax of the language should be. This can be accomplished through the use of productions that build the call to the special functions (see Section 4.4.2) by getting tokens from the input stream. Through appropriate setting of the attributes (character strings) of these tokens, the parameters to the special functions could be constructed.

6.2. Deferred Semantic Actions

In general, if a CFG is lookahead m (m>k) then LL(k) and LR(k) parser generators will not be able to generate parsers to parse it. Moreover, if the CFG has unbounded lookahead then it cannot be parsed by any LL(k) or LR(k) parsers. The example in this section presents a grammar that has unbounded lookahead.

Suppose that an application accepts a list of numbers and interprets them as decimal, octal, or hexadecimal depending on the keyword that follows the list of numbers. Specifically, if the list of numbers is followed by the keyword 'd+' then the numbers in the list are interpreted as decimal, and their sum is printed. Similarly, the keyword 'o+' results in interpreting the numbers as octal, and the keyword 'h+' results in hexadecimal addition. For example, if the input to the application is the following:

```
4 35 d+
4 35 o+
4 35 h+
```

then the desired output is:

```
39
33
57
```

An implementation of this application is now shown. The following grammar specification:

```
%start prog
%union {
char *sval;
int  ival;
}
%token <sval> NUM
%token OPLUS
%token DPLUS
%token HPLUS
%type <ival> prog evals eval oladd dladd hladd ol dl hl
%%
prog:evals ;

evals:evals eval {printf("\n%d",$2);}
  | eval {printf("\n%d",$1);}
  ;

eval:oladd { $$ = $1;}
  | dladd { $$ = $1; }
  | hladd { $$ = $1; }
  ;

oladd:ol OPLUS { $$ = $1; } ;

dladd:dl DPLUS { $$ = $1; } ;

hladd:hl HPLUS { $$ = $1; } ;

ol:ol NUM { $$ = $1 + earley_eval_int($2,8,"01234567"); }
  | NUM { $$ = earley_eval_int($1,8,"01234567"); }
  ;
```

```
dl:dl NUM { $$ = $1 + earley_eval_int($2,10,"0123456789"); }
  | NUM { $$ = earley_eval_int($1,10,"0123456789"); }
  ;

hl:hl NUM { $$ = $1 + earley_eval_int($2,16,"0123456789abcdef");
  | NUM { $$ = earley_eval_int($1,16,"0123456789abcdef"); }
  ;
%%
```

and the following lexical analyzer specification:

```
D [0-9]
%{
#include "earley.tab.h"
%}
%%
"o+" {return(OPLUS);}
"d+" {return(DPLUS);}
"h+" {return(HPLUS);}
{D}+ {earley_userlval.sval=concat(yytext,"");return(NUM);}
[ \t\n]+ {}
%%
```

implement the desired application.

As the parser gets the 'NUM' token from the input stream (through the lexical analyzer), it cannot decide which of the three lists it is scanning, decimal, octal, or hexadecimal. Every state set, except the first, has more than one final state (see Section 3). As a consequence, semantic actions are deferred (see Section 5.5). Finally, when the parser scans one of the three keywords, then it can determine how to interpret the numbers in the list. At that point in the parse the local ambiguity is resolved, and the deferred semantic actions are executed. The reader can verify that prior to the keyword being scanned every state set has more than one final state, by creating the parser based on the PEG specification file given in this section, using the '-PRINT2' flag (see Section 4.10), and parsing any one of the three string given in this example.

6.3. Linking Multiple Parsers

Section 4.7 discussed the naming convention followed by PEG. The motivation for imposing a naming convention is to enable the user to link multiple parsers in a single executable. Suppose two parsers are generated by PEG. Both these parsers will have routines and variables with identical names. Obviously, an attempt to link these two parsers in a single executable will fail due to the multiple declarations.

Having a uniform naming convention makes it easy to link multiple parsers together. To link two parsers in a single executable the following steps might be followed:

- Change all names starting with the string 'earley' to 'earley1' in the first parser.
- Change all names starting with the string 'earley' to 'earley2' in the second parser.
- Change the name of the lexical analyzer provided with the first parser to 'earley1_userlex' and the name of the union in this lexical analyzer to 'earley1_userlval' (see Section 4.9).
- Change the name of the lexical analyzer provided with the first parser to 'earley2_userlex' and the name of the union in this lexical analyzer to 'earley2_userlval'.

Now the two parsers may be linked together without any naming conflicts. This simple procedure easily generalizes to the linkage of more than two parsers.

Consider the grammar specification file given below:

```
%token A
%token C
%%
prog: sub A C {printf("\nSeen 'aac'");}
  | A A {printf("\nSeen 'aa'");}
  ;

sub: A ;
%%
```

Suppose the following lexical analyzer is provided with this grammar:

```
%{
#include "earley.tab.h"
%}
%%
"a" {return(A);}
"c" {return(C);}
[\n] {return(0);}
[ \t] {}
%%
```

Notice that the above lexical analyzer returns an end-of-file on encountering a newline character (see Section 4.9).

Now suppose two parsers are created, one using the grammar specification given in this section and the other using the grammar specification given in Section 6.1. Let these two parsers be in files 'earley1.c' and 'earley2.c' respectively. Furthermore, let the lexical analyzers (generated by LEX) for the two parsers be in files 'earley1_userlex.c' and 'earley2_userlex.c', respectively. To make the changes (to the two parsers and lexical analyzers) specified at the beginning of this section the following shell file could be executed, where 'tmp' is a temporary file:

```
cp earley1.c tmp
sed -e "s/earley/earley1/g" tmp > earley1.c
cp earley2.c tmp
sed -e "s/earley/earley2/g" tmp > earley2.c
cp earley1_userlex.c tmp
sed -e "s/yy/earley1_user/g" tmp > earley1_userlex.c
cp earley2_userlex.c tmp
sed -e "s/yy/earley2_user/g" tmp > earley2_userlex.c
```

Suppose the main procedure is the following:

```
main ()
{
/*
extern FILE *earley1_userin, *earley2_userin;
      earley1_userin = fopen("earley1.in","r");
      earley2_userin = fopen("earley2.in","r");
*/
      earley1_init();
      earley1_parse();

      earley2_init();
      earley2_parse();
}
```

If the two parsers and lexical analyzers are linked together with the main procedure and the following input is provided:

```
aac
13 + 5 * 7;
CHANGE;
13 5 7 * +;
```

then the output from the program is the following:

```
Seen 'aac'
48
Switching to Postfix
48
```

It may be the case that the user wants the input to come from file(s). In that case, the file pointer(s) of the lexical analyzer(s) must be set to the desired input files. In the example presented in this section, if the commented lines of code were compiled, then the input for the first parser (earley1_parse) would come from the file 'earley1.in', and the input for the second parser (earley2_parse) would come from the file 'earley2.in'. This particular method of redirecting input is a feature of lexical analyzers generated by LEX.

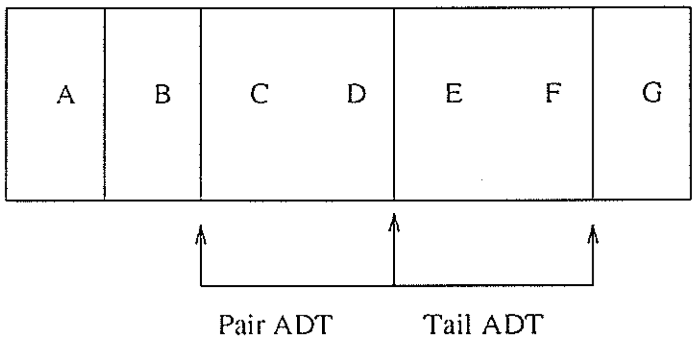
7. Derivation Graph

This section illustrates the derivation graph generation features of the parsers generated by PEG. During parsing a derivation graph is created by the parsers. This derivation graph encapsulates all the possible derivations of the string being parsed. These parsers write an unattributed version of the derivation graph to a file (see Section 4.10), and return a pointer to an attributed version at the end of the parse. See Section 8 for a description of the ADTs and routines necessary to manipulate the derivation graph.

7.1. Internal Structure of the Derivation Graph

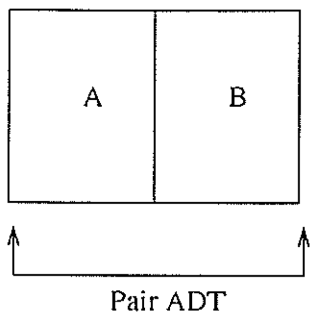
The derivation graph created by the PEG parsers contains two types of nodes: **nonterminal** and **terminal**. The structure of the nonterminal nodes is given in Figure 7.1, and Figure 7.2 gives the structure of the terminal nodes.

If the string being parsed has only one derivation tree then the parser creates an **attributed** derivation graph of the string. That is, the nonterminal nodes contain the attribute information that was computed by the semantic action routines (see Section 4.4.1). Otherwise, the parser creates an **unattributed** derivation graph. The unattributed derivation graph is identical to the attributed derivation graph, except that the unattributed derivation graph does not contain attribute information for the nonterminal nodes.



A: Nonterminal symbol	E: 1 if node is at tail of a cycle
B: A pointer to xdeq of alternatives	F: Pointer to cycle head if E = 1
C: 1 if node is at head of a cycle	G: Attributes of nonterminal symbol
D: Label of cycle head if C = 1	

Figure 7.1: Nonterminal Node



A: Terminal symbol B: Attributes of terminal symbol

Figure 7.2: Terminal Node

Consider the following grammar specification file for PEG (see Section 4):

```
%start Exp
%token PLUS
%union {
char *s;
}
%%
Exp: A B;

A: C;

C: A | PLUS;

B: D;

D: B | PLUS;
%%
```

This grammar is cyclic. The only string in the language is '++'. Assume that the associated lexical analyzer returns the token PLUS when it sees a '+' in the input, along with the character string '+' as the attribute of the token. Then, the derivation graph created by the PEG parser based on this grammar is given in Figure 7.3.

Suppose that a parser was generated by PEG, based on a specification file corresponding to Grammar 2.8. Furthermore, suppose that this parser was used to parse the string 'ab'. Then, Figure 7.4 gives the internal representation of the derivation graph of the string 'ab' under G2.8. Note that the cardinality of the alternatives xdeqs for the nonterminal nodes 'A' and 'B' is 2, corresponding to the two distinct paths that may be followed from each of these two nodes in the derivation of the string 'ab'.

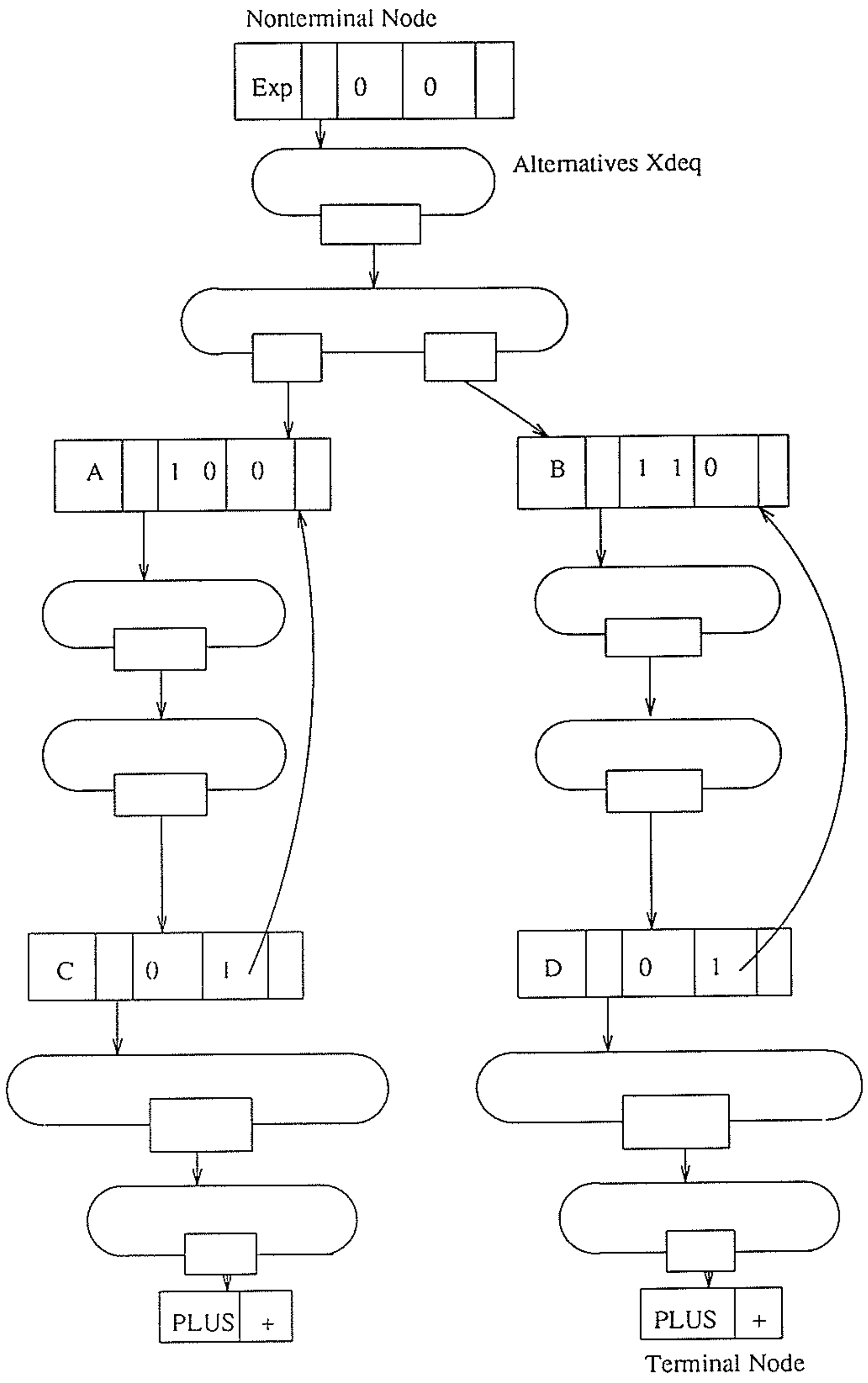


Figure 7.3: Internal Representation of Derivation Graph of '++'

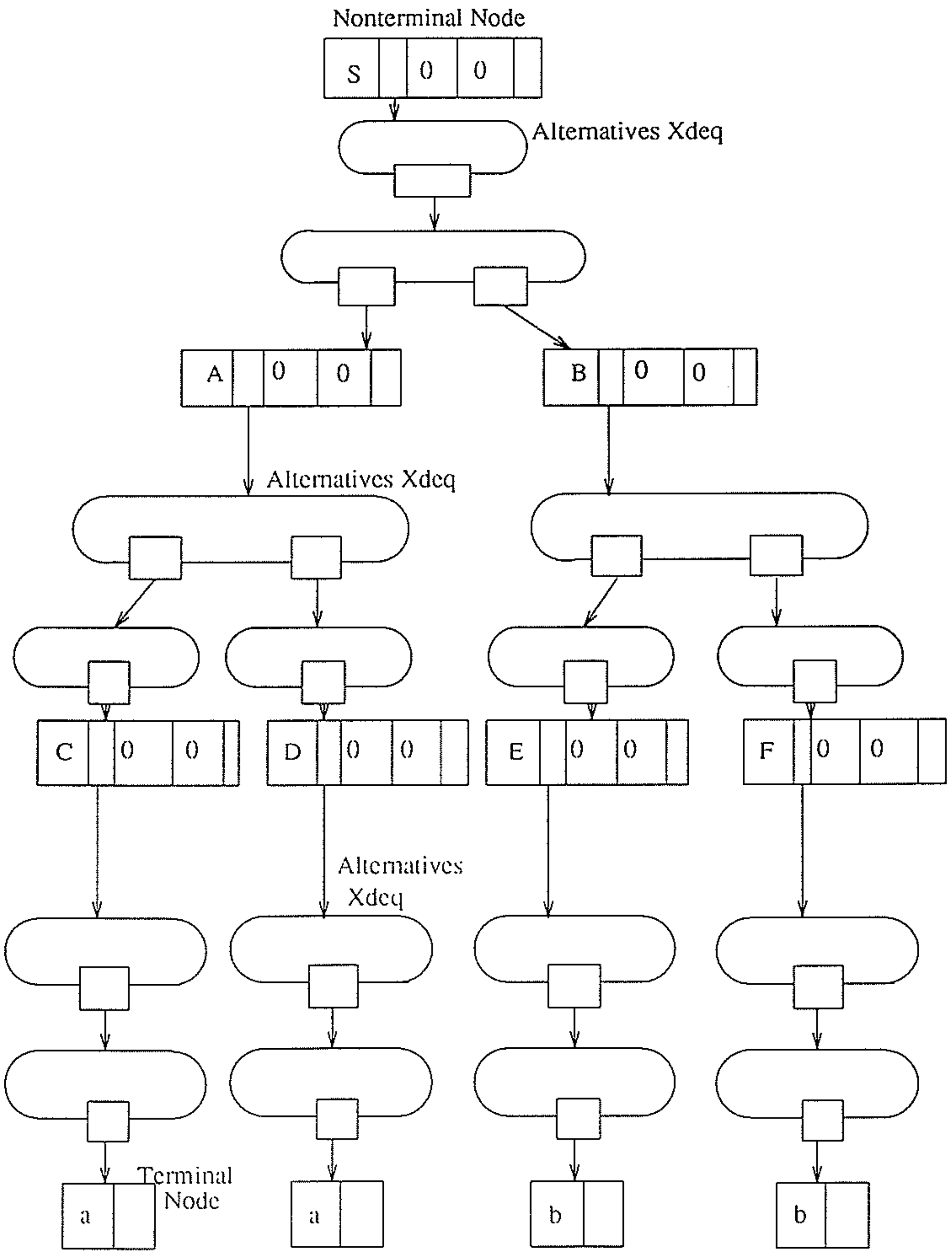


Figure 7.4: Internal Representation of Derivation Graph of 'ab' under G2.8

Notice that in Figure 7.3 and Figure 7.4 some of the fields in the nodes of the derivation graphs are empty. This is meant to indicate that the information held in those fields is "garbage". For example, if the pair ADT in a nonterminal node (see Figure 7.1) has a 0 in it and nothing else, then it means that the nonterminal is not at the head of a cycle and that the field for the label of the cycle head contains "garbage".

7.2. Printing the Derivation Graph

The PEG library provides routines to manipulate the derivation graph created by the PEG parsers (see Section 8). Specifically, the routine **EarleY_display_parse_graph** prints the derivation graph returned at the end of the parse, using spacing and metasymbols to display the structure of the derivation graph.

All nonterminal symbols are bracketed between '<' and '>'. If a nonterminal has multiple derivations, then each of its derivations is bracketed between '[' and ']'. If a nonterminal is at the head of a cycle, then an integer label is printed along with the nonterminal to uniquely identify the cycle. For example, if the nonterminal 'A' is at the head of a cycle then it will be printed as '<A>:label'. If a nonterminal is at the tail of a cycle, then the label of the corresponding head of the cycle is printed along with the nonterminal. For example, if the nonterminal 'C' is at the tail of a cycle then it will be printed as '!:<c>:label'. The empty string (λ) is printed as '*'.

The label for the head and the tail of a cycle is an encoding of the backward arc discussed earlier (see Section 2.4.1). The alternatives xdeq is the internal representation of the concept of the alternatives structure (see Section 2.4.2). For the grammar of the previous section, a conceptual picture of the derivation graph created by the parser for the string '++' is given in Figure 7.5. Suppose the following **main** program was provided for the parser based on the grammar of the previous section:

```
#include "earley.tab.h"
main ()
{
  earley_init();
  EarleY_display_parse_graph(earley_parse());
}
```

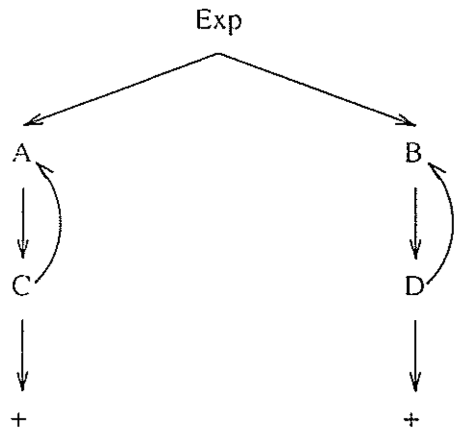

$$\begin{tikzcd}
 & Exp \arrow[dl] \arrow[dr] & \\
A \arrow[d] & & B \arrow[d] \\
C \arrow[u, bend right] \arrow[d] & & D \arrow[u, bend right] \arrow[d] \\
+ & & +
\end{tikzcd}$$

Figure 7.5: Conceptual Representation of Derivation Graph of '++'

The main program (given above) prints the derivation graph that is returned at the end of the parse, using the encoding scheme given in the previous paragraph. Notice the close correspondence between the derivation graph printed by the parser, given in Figure 7.6, and the structure of the conceptual derivation graph, given in Figure 7.5.

```
<Exp>
    <A>:0
        <C>
            [
            PLUS
            ]
            [
            !:<A>:0
            ]
    <B>:1
        <D>
            [
            PLUS
            ]
            [
            !:<B>:1
            ]
```

Figure 7.6: Printout of Derivation Graph of '++'

7.3. Interpreting the Derivation Graph Printout

By "looking at" the printout of the derivation graph one should be able to determine the structure of the derivation tree(s) encoded by the derivation graph. In other words, by "looking at" the derivation graph the user should be able to determine the string that was parsed. If the CFG on which the PEG parser is based, is unambiguous, then "reading" the printout of the derivation graph for any string that is parsed is easy; the derivation graph is basically a single derivation tree. However, if the derivation graph encodes direct or indirect ambiguity (see Section 2.1.2) then it may not be easy to determine what (legal) derivation trees are encoded by the derivation graph.

Consider Grammar 7.1. This grammar is ambiguous, and the conceptual derivation graph for the input string 'a' is given in Figure 7.7. The derivation graph in Figure 7.7 encodes four derivation trees. However, two of the derivation trees encoded by this derivation graph are not valid (for the input strings λ and 'aa'). The printout of this derivation graph also displays four derivation trees. Clearly, some algorithm is needed to assist the user in "decoding" the printout of the derivation graph to determine the string that was actually parsed.

Due to certain implementation characteristics of PEG, the rule to determine the string that was parsed, from the printout of a derivation graph, is very simple. A **depth-first traversal of**

$$\begin{array}{lllll} S & \rightarrow & A & B & \\ A & \rightarrow & a & | & \lambda \\ B & \rightarrow & a & | & \lambda \end{array}$$

Grammar 7.1

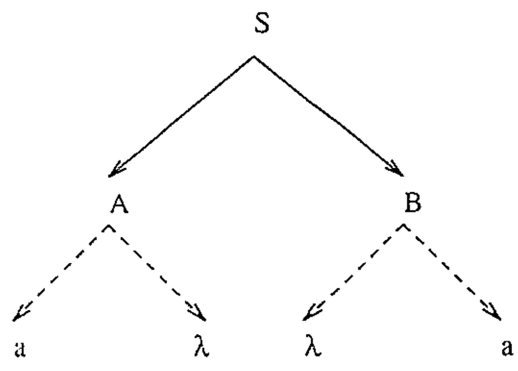


Figure 7.7: Conceptual Derivation Graph of 'a' under Grammar 7.1

the printout of the derivation graph is performed, cycles in the derivation graph are ignored, and for each nonterminal node in the derivation graph the first of its alternatives is chosen. This algorithm is outlined in Figure 7.8.

The algorithm given in Figure 7.8 identifies the string that was parsed (the sequence of tokens recognized during parsing). Once the sequence of the tokens that were recognized during parsing is determined the user can easily prune out the invalid derivation trees from the printout of the derivation graph.

```
procedure visit(n:node)
begin
     if n is a terminal node then
          print n;
          return;
     else
          pick the first non-cyclic alternative of n, m;
          for each child l of m (from top-to-bottom)
               visit(l);
end
```

Figure 7.8: Algorithm to Determine the String Parsed from Printout of Derivation Graph

8. Abstract Data Types (ADTs) Used in Derivation Graph

This section gives the structure of some important ADTs and the operations applicable on them. If the user wants to manipulate the derivation graph generated by the parser then some knowledge of the ADTs used to create the derivation graph is necessary. The ADTs necessary to manipulate the derivation graph are described in this chapter.

8.1. Extended Double Ended Queue (Xdeq)

An xdeq is essentially a list. The elements of this list are pointers to other ADTs. Elements may be accessed at either end of this list, or at any position within the list. Three pointers are maintained to access the elements of an xdeq: one pointer points to the left end of the xdeq, the second pointer points to the right end of the xdeq, and the third pointer points to the current position within the list. The important operations applicable on an xdeq are the following:

EarleY_PNT_XDEQ_NODE_TYPEEarleY_xdeq_create(parent)
EarleY_MEM_TYPE parent;
This function creates an empty xdeq. The pointer 'parent' is the parent of this newly created xdeq (EarleY_MEM_NULL if no parent information is available).

voidEarleY_xdeq_move_to_left_end(p)
EarleY_PNT_XDEQ_NODE_TYPE p;
This function moves the current position pointer to the left end of the xdeq p.

EarleY_MEM_TYPEEarleY_xdeq_get_data(p)
EarleY_PNT_XDEQ_NODE_TYPE p;
This function returns the element pointed to by the current pointer of the xdeq p.

EarleY_MEM_TYPEEarleY_xdeq_get_data_go_right(p)
EarleY_PNT_XDEQ_NODE_TYPE p;
This function returns the element pointed to by the current pointer of the xdeq p and move the current pointer one position to the right.

EarleY_MEM_TYPEEarleY_xdeq_extract_data(p)
EarleY_PNT_XDEQ_NODE_TYPE p;
This function returns the element pointed to by the current pointer of the xdeq p and removes the element from the xdeq.

int EarleY_xdeq_cardinality(p)
EarleY_PNT_XDEQ_NODE_TYPE p;
This function returns the number of elements in the xdeq p.

voidEarleY_get_data_at_pos_i(p,i)
EarleY_PNT_XDEQ_NODE_TYPE p;
int i;

This function returns the ith element of the xdeq p.

voidEarleY_insert_data_at_pos_i(p,i,data)
EarleY_PNT_XDEQ_NODE_TYPE p;
int i;
EarleY_MEM_TYPE data;
This function insert the data at the ith position in the xdeq p.

voidEarleY_insert_data_at_left_end(p,data)
EarleY_PNT_XDEQ_NODE_TYPE p;
EarleY_MEM_TYPE data;
This function inserts the data at the left end of the xdeq p.

voidEarleY_insert_data_at_right_end(p,data)
EarleY_PNT_XDEQ_NODE_TYPE p;
EarleY_MEM_TYPE data;
This function inserts the data at the right end of the xdeq p.

voidEarleY_xdeq_destroy(p)
EarleY_PNT_XDEQ_NODE_TYPE *p;
This function destroys the xdeq p.

EarleY_PNT_XDEQ_NODE_TYPEEarleY_xdeq_copy(parent,p)
EarleY_MEM_TYPE parent;
EarleY_PNT_XDEQ_NODE_TYPE p;
This functions creates a copy of the xdeq p and returns a pointer to the copy. The parent argument should be 'EarleY_MEM_NULL' if no parent information is available.

voidEarleY_xdeq_pprint(p)
EarleY_PNT_XDEQ_NODE_TYPE p;
This function prints the xdeq p.

To step through the elements of the xdeq p the following C language code might be used:

```
EarleY_xdeq_move_to_left_end(p);
j = EarleY_xdeq_cardinality(p);
for(i=1;i<=j;i++) {
  element = EarleY_xdeq_get_data_go_right(p);
  /* Other statements */
}
```

8.2. EarleY_PAIR_TYPE ADT

An EarleY_PAIR_TYPE is an ADT that hold two data objects, a left object and a right object. The following operations are defined on this ADT:

EarleY_PAIR_TYPEEarleY_pair_create(left,right)
EarleY_MEM_TYPE left, right;
This function creates a pair ADT and sets the left and right elements of this newly created pair.

EarleY_MEM_TYPEEarleY_pair_get_left_element(p)
EarleY_PAIR_TYPE p;
This function returns the left element of the pair p.

EarleY_MEM_TYPEEarleY_pair_get_right_element(p)
EarleY_PAIR_TYPE p;
This function returns the right element of the pair p.

EarleY_MEM_TYPEEarleY_pair_set_left_element(p,data)
EarleY_PAIR_TYPE p;
EarleY_MEM_TYPE data;
This function sets the left element of the pair p.

EarleY_MEM_TYPEEarleY_pair_set_right_element(p,data)
EarleY_PAIR_TYPE p;
EarleY_MEM_TYPE data;
This function sets the right element of the pair p.

voidEarleY_pair_destroy(p)
EarleY_PAIR_TYPE *p;
This function destroys the pair p.

EarleY_PAIR_TYPEEarleY_pair_copy(parent,p)
EarleY_MEM_TYPE parent;
EarleY_PAIR_TYPE p;
This function makes a copy of the pair p and returns a pointer to the copy. The parent argument should be 'EarleY_MEM_NULL' if no parent information is available.

voidEarleY_pair_pprint(p)
EarleY_PAIR_TYPE p;
This function prints the pair p.

8.3. EarleY_TAIL_TYPE ADT

An EarleY_TAIL_TYPE is an ADT that hold two data objects. The following operations are defined on this ADT:

EarleY_TAIL_TYPEEarleY_tail_create(left,right)
EarleY_MEM_TYPE left, right;
This function creates a tail ADT and sets the left and right elements of this newly created tail.

EarleY_MEM_TYPEEarleY_tail_get_left_element(p)
EarleY_TAIL_TYPE p;
This function returns the left element of the tail p.

EarleY_MEM_TYPEEarleY_tail_get_right_element(p)
EarleY_TAIL_TYPE p;
This function returns the right element of the tail p.

EarleY_MEM_TYPEEarleY_tail_set_left_element(p,data)
EarleY_TAIL_TYPE p;
EarleY_MEM_TYPE data;
This function sets the left element of the tail p.

EarleY_MEM_TYPEEarleY_tail_set_right_element(p,data)
EarleY_TAIL_TYPE p;
EarleY_MEM_TYPE data;
This function sets the right element of the tail p.

voidEarleY_tail_destroy(p)
EarleY_TAIL_TYPE *p;
This function destroys the tail p (the right element of a tail ADT is not destroyed).

EarleY_TAIL_TYPEEarleY_tail_copy(parent,p)
EarleY_MEM_TYPE parent;
EarleY_TAIL_TYPE p;
This function makes a copy of the tail p and returns a pointer to the copy. This routine does not copy the right element of the tail p; it might lead to infinite copying. The parent argument should be 'EarleY_MEM_NULL' if no parent information is available.

voidEarleY_tail_pprint(p)
EarleY_TAIL_TYPE p;
This routine prints the tail p. Since the right element of a tail is (possibly) a pointer that could result in infinite looping, only the left element of the tail is printed.

Notice that two of the access functions provided with this ADT, EarleY_tail_copy and EarleY_tail_pprint, are not functionally complete. The reason for this drawback is that a tail ADT has (possibly) a pointer that could induce an infinite looping in the access functions. The user can supply his/her own copy and print routines for this ADT, and maybe a second version of PEG will remedy this drawback.

8.4. EarleY_NODE_TYPE ADT

A node is an ADT that holds five elements. The nonterminal nodes in the derivation graph are created using the EarleY_NODE_TYPE ADT (see Section 7). The operations applicable on an EarleY_NODE_TYPE ADT are the following:

EarleY_NODE_TYPEEarleY_node_create(p1, p2, p3, p4, p5)
EarleY_MEM_TYPE p1, p2, p3, p4, p5;
This function creates a node and sets its five elements. A pointer to this newly created node is returned.

voidEarleY_node_destroy(p)
EarleY_NODE_TYPE *p;
This function destroys the node p.

EarleY_MEM_TYPEEarleY_node_get_ith_element(i,p)
int i;
EarleY_NODE_TYPE p;
This function returns the ith element of the node p.

voidEarleY_node_set_ith_element(i,p,data)
int i;
EarleY_NODE_TYPE p;
EarleY_MEM_TYPE data;
This function sets the ith element of the node p to data.

EaleY_NODE_TYPEEarleY_node_copy(parent, p)
EarleY_MEM_TYPE parent;
EarleY_NODE_TYPE p;
This routine makes a copy of the node p, and returns a pointer to the copy. The parent argument should be 'EarleY_MEM_NULL' if no parent information is available.

voidEarleY_node_pprint(p)
EarleY_NODE_TYPE p;
This outine prints the node p.

8.5. Integer and Character ADTs

In the ADT paradigm all data types are coded as ADTs. That means, even basic data types such as integer and character have a corresponding ADT defined for them. The following is a list of functions and macros that are used to manipulate these basic data types:

EarleY_PNT_INT_TYPEEarleY_int_create(i)
int i;
This function creates an integer ADT, sets its value to i, and returns a pointer to this newly created ADT.

EarleY_PNT_STRING_TYPEEarleY_string_create(s)
char *s;
This function creates an string ADT, sets its value to s, and returns a pointer to this newly created ADT.

voidEarleY_int_destroy(p)
EarleY_PNT_INT_TYPE *p;
This function destroys the integer ADT p.

voidEarleY_string_destroy(p)
EarleY_PNT_STRING_TYPE *p;
This function destroys the string ADT p.

int EarleY_INT_VAL(p)
EarleY_PNT_INT_TYPE p;
This macro returns the integer stored in the integer ADT p.

char*EarleY_STNG_VAL(p)
EarleY_PNT_STRING_TYPE p;
This macro returns the character string stored in the string ADT p.

8.6. Accessing the Tag of an ADT

An ADT is a tagged data structure (see Section 5.1). This tag identifies the type of the ADT. To access the tag the following macro may be used:

EarleY_TAG_TYPEEarleY_TAG_OF(p)
EarleY_MEM_TYPE p;
This function returns the tag of an ADT. EarleY_TAG_TYPE are the following:

EarleY_xdeq_tag
EarleY_node_tag
EarleY_pair_tag
EarleY_tail_tag

```
EarleY_int_tag
EarleY_string_tag
```

8.7. earley_SEM_VAL Macro

Every parser generated by PEG has an unique semantic ADT defined for it. This ADT is used to handle the attributes of the symbols of the grammar during parsing. The attributed derivation graph gives the user access to these attributes. For each parser generated by PEG a 'earley_SEM_VAL' macro is defined. This macro returns a pointer to the union of attributes for a node in the derivation graph (see Section 4.3). For example, if the user wanted to access the attributes of a terminal node 'anode', then the following code would suffice to get a pointer to the union structure of this terminal node:

```
earley_sem_typeunion_p;

union_p = earley_SEM_VAL((earley_SEM_TYPE) EarleY_pair_get_right_element(anode));
```

Notice that the macro name starts with the string 'earley'. If multiple parsers are linked together then each parser will have its own macro, and type, for accessing the attributes of the nodes of its derivation graph. Therefore, this macro name is not static (see Section 6.3).

8.8. Derivation Graph Manipulation Routines

The following routines are provided to manipulate the unattributed derivation graph that is written to a file (see Section 4.10):

```
voidEarleY_initialize_parse_graph_routines()
```

If the derivation graph is to be manipulated from a different executable than the parser, then this routine must be called before any other graph manipulation routine.

```
struct EarleY_graph_info  EarleY_recreate_parse_graph(file)
char *file;
typedef struct EarleY_graph_info {
    int tokens_parsed;
    EarleY_NODE_TYPE graph;
};
```

This routine reads in the unattributed derivation graph from a file and builds the data structures necessary to store the graph. The structure returned by this routine holds the number of tokens that were parsed, and the derivation graph.

```
voidEarleY_display_parse_graph(p)
EarleY_NODE_TYPE p;
```

This routine prints out the unattributed derivation graph whose starting node is p.

9. Performance of PEG Parsers

In this section statistics about the parse times of some parsers generated by PEG are given. The objective is to give some idea of the time complexity of this implementation. In particular, the objective is to verify that this implementation's timing behavior is no more than $O(n^3)$, because Earley's Algorithm is, in general, a $O(n^3)$ parsing algorithm (see Section 3.3). These tests were done on a Sun SPARC-2. All the parsers were created using the '-PARSE0' flag (see Section 4.10). Grammars G9.1 and G9.2 are the grammars of Section 6.1 and Section 6.2 respectively. Grammars G9.3 and G9.4 are taken from Earley's thesis. Notice, G9.3 is a very ambiguous grammar, and G9.4 is lookahead 2.

$$\begin{array}{lllllll} S & \rightarrow & A & B & & & \\ A & \rightarrow & a & \mid & A & b & \\ B & \rightarrow & b & c & \mid & b & B \end{array}$$

Grammar G9.3

$$S \rightarrow a \mid a \; S \; a$$

Grammar G9.4

The tables and graphs on the following pages give the statistics of the timings displayed by the parsers based on the above four grammars. All the times are in seconds and were obtained using the **time** command available under the UNIX operating system (the **user** and **system** times were added to get the figures given in the tables). Note that superscripts in the **String** column of the tables are used to indicated repetition of a character string. For example, $[3+]^{250}$ means that the string '3+' is repeated 250 times in the input, and 3^{250} means that the string '3' is repeated 250 times in the input. Each Figure consists of two graphs, one for the parser using a lookahead of 0, and the other for the parser using a lookahead of 1.

Table 1 (Lookahead 0)		
Grammar	String	Time
G1	$[3+]^{125}3;$	1.9
G1	$[3+]^{250}3;$	3.7
G1	$[3+]^{500}3;$	7.2
G1	$[3+]^{1000}3;$	14.9
G2	3^{250} d+	2.3
G2	3^{500} d+	4.5
G2	3^{1000} d+	9.0
G2	3^{2000} d+	18.2
G3	$ab^{162}c$	1.1
G3	$ab^{322}c$	2.3
G3	$ab^{642}c$	5.5
G3	$ab^{1282}c$	14.1
G4	a^{21}	0.2
G4	a^{43}	0.8
G4	a^{89}	4.2
G4	a^{179}	22.7

Table 9.1
Parse Times (Lookahead 0)

Table 2 (Lookahead 1)		
Grammar	String	Time
G1	$[3+]^{125}3;$	2.3
G1	$[3+]^{250}3;$	4.6
G1	$[3+]^{500}3;$	9.1
G1	$[3+]^{1000}3;$	18.1
G2	3^{250} d+	2.7
G2	3^{500} d+	5.5
G2	3^{1000} d+	10.9
G2	3^{2000} d+	21.9
G3	$ab^{162}c$	1.3
G3	$ab^{322}c$	2.9
G3	$ab^{642}c$	6.6
G3	$ab^{1282}c$	15.8
G4	a^{21}	0.2
G4	a^{43}	0.9
G4	a^{89}	4.6
G4	a^{179}	24.9

Table 9.2
Parse Times (Lookahead 1)

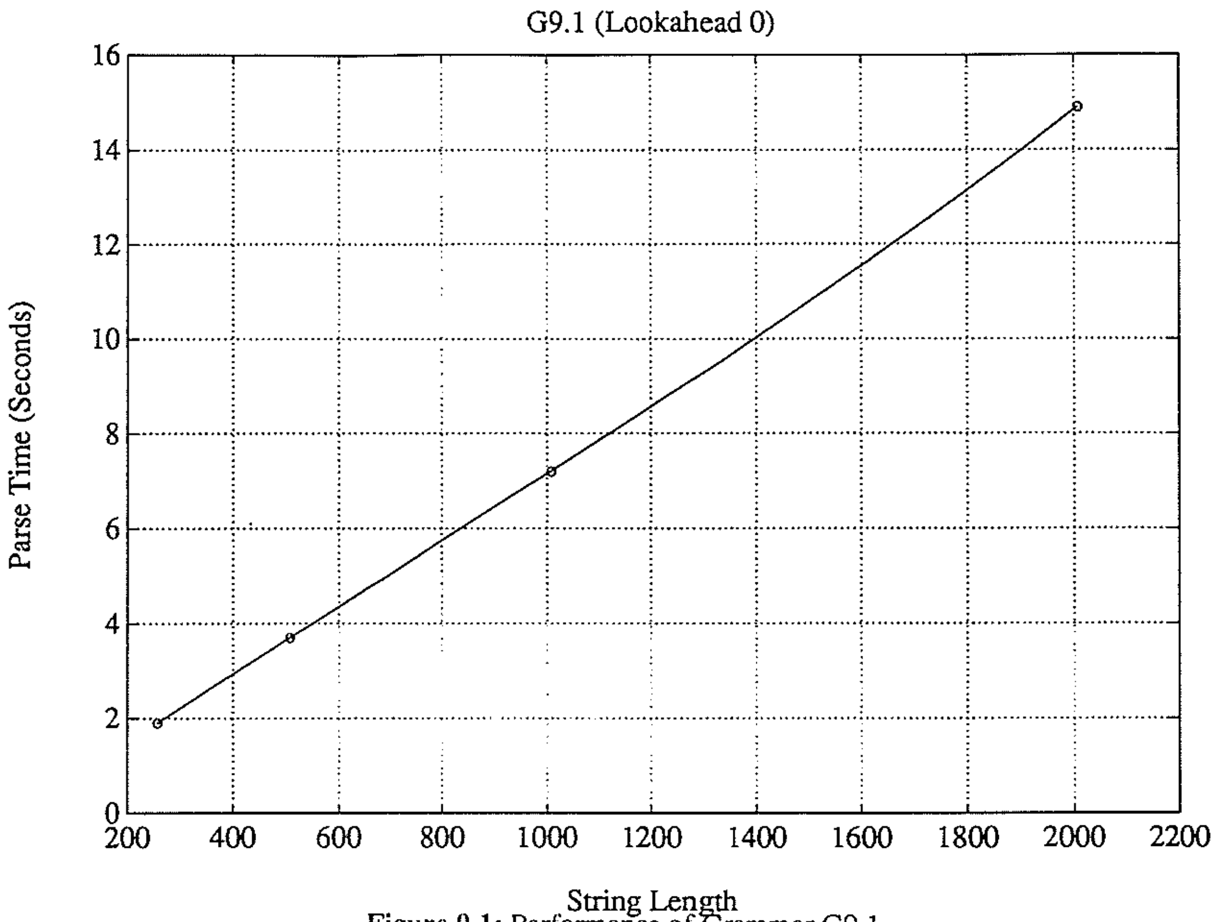
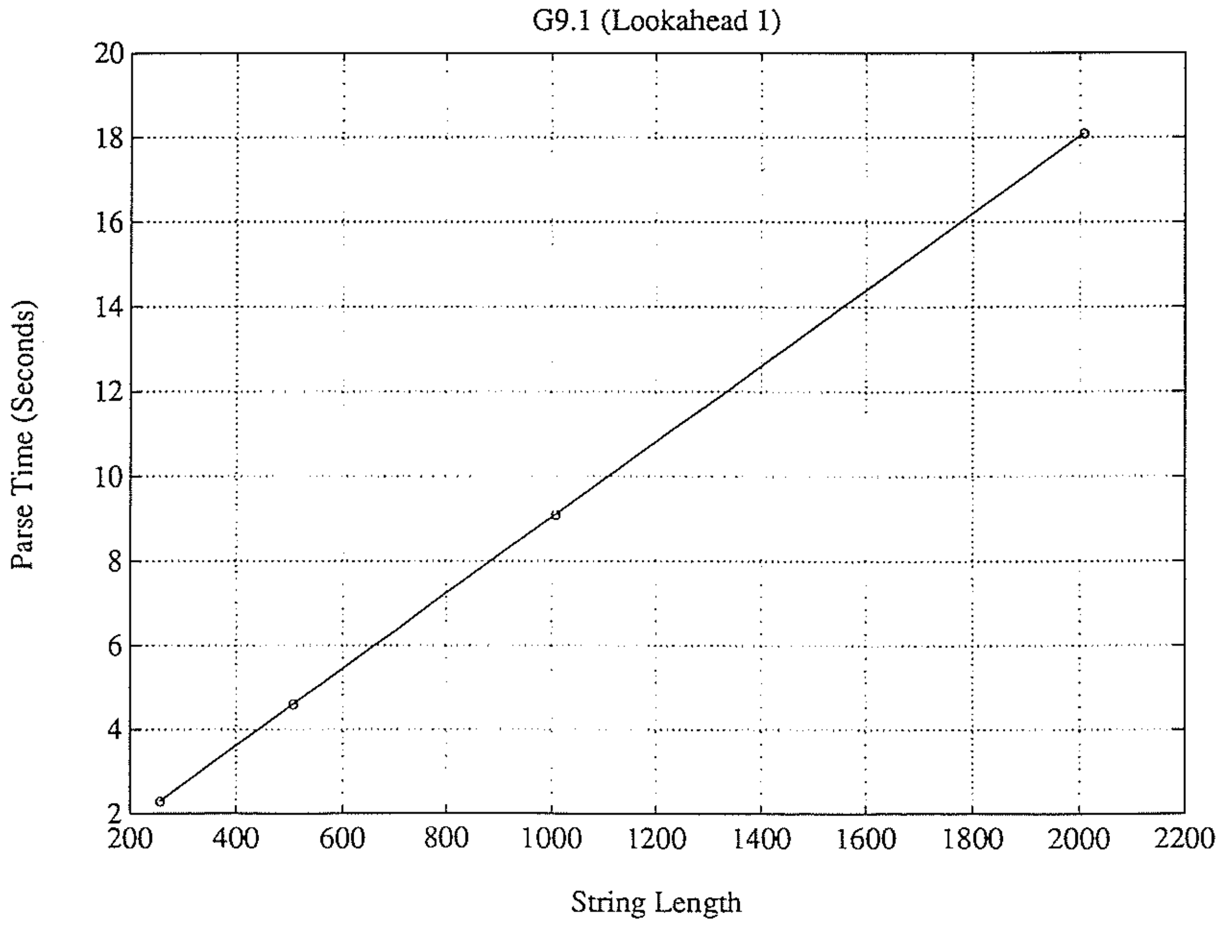


Figure 9.1: Performance of Grammar G9.1



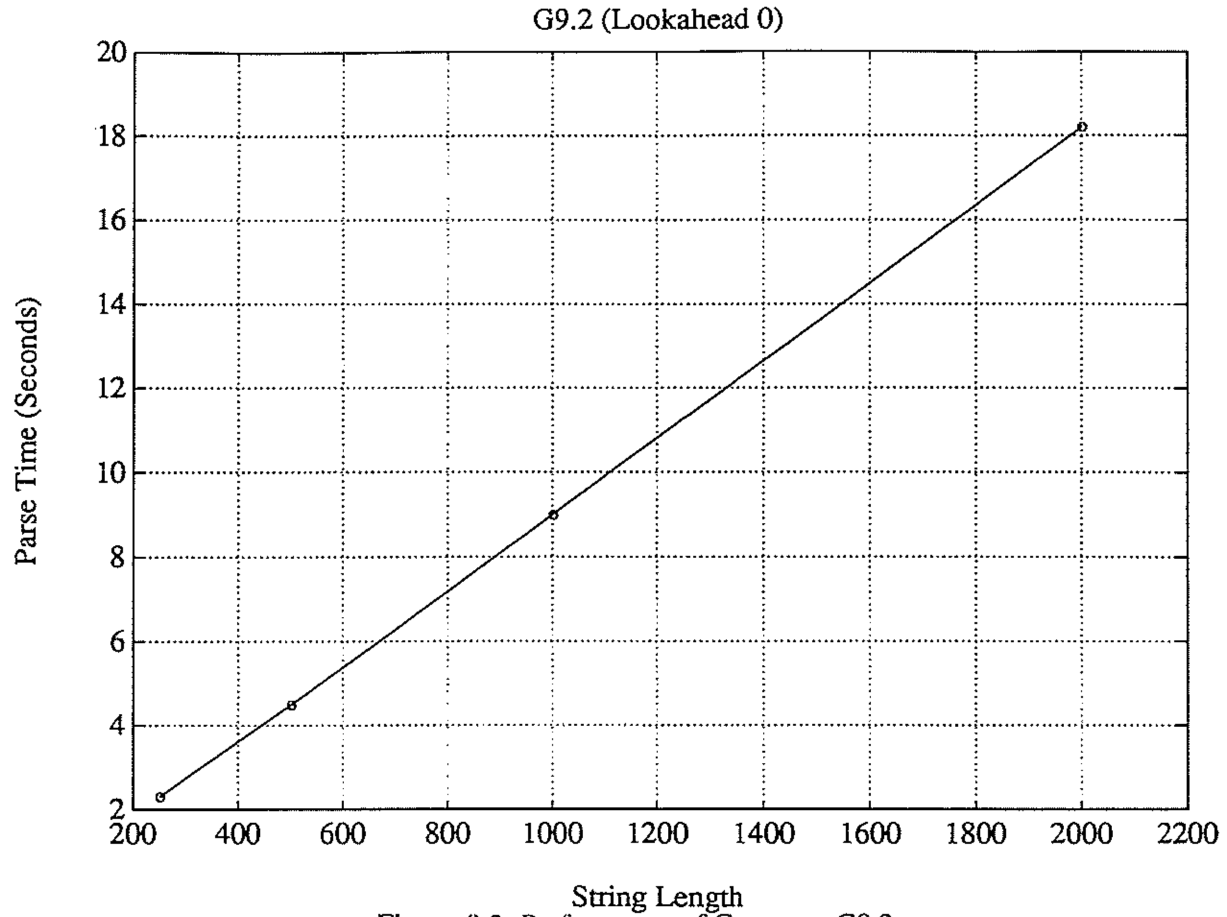
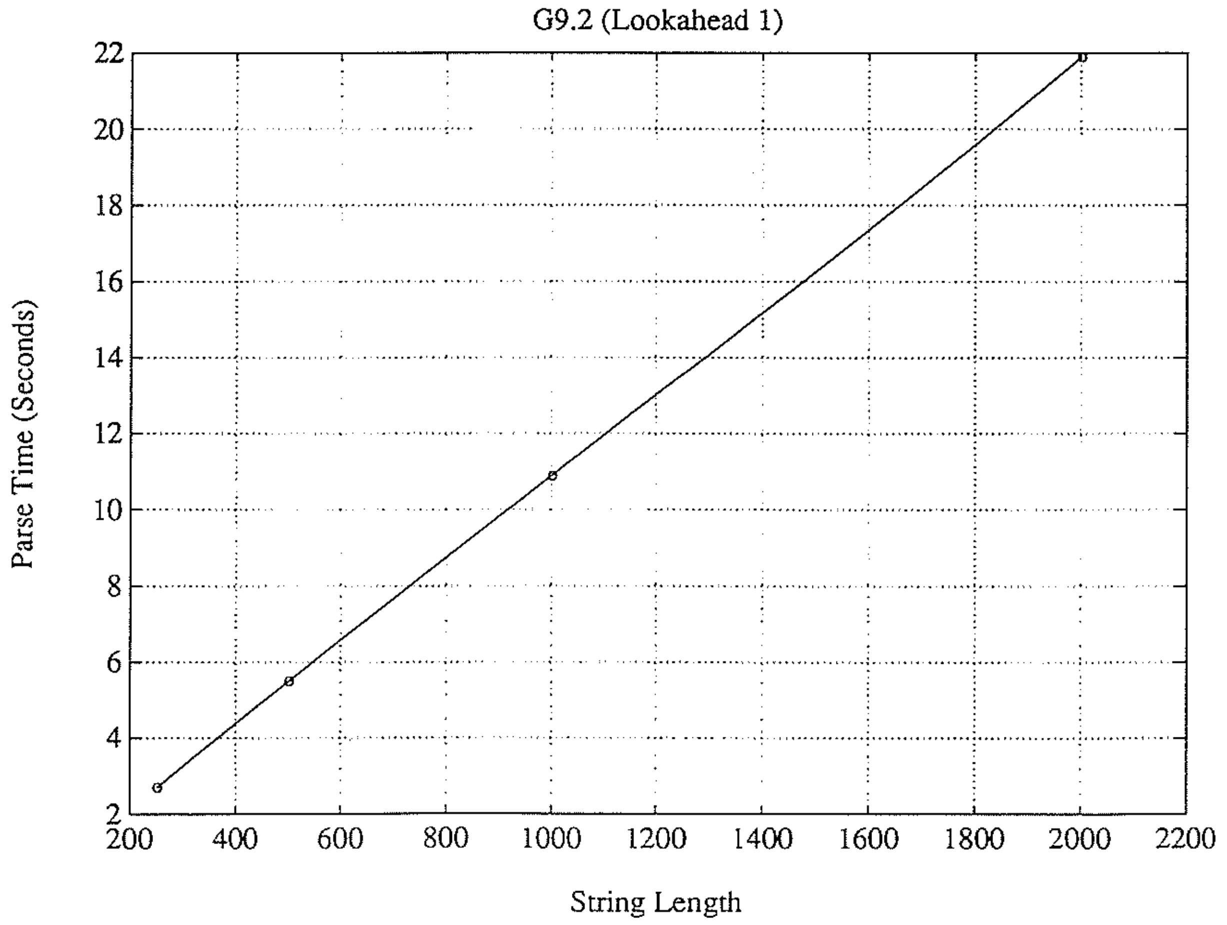


Figure 9.2: Performance of Grammar G9.2



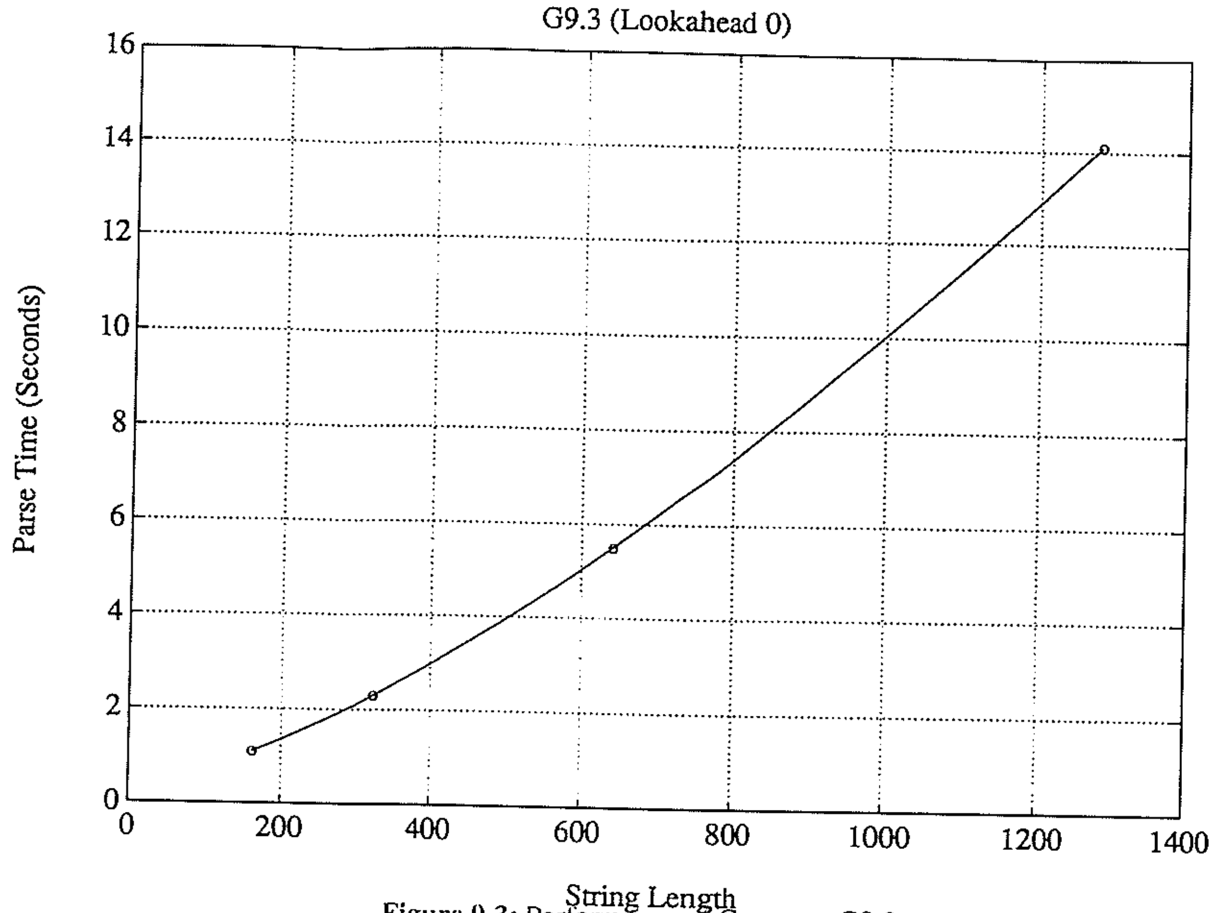
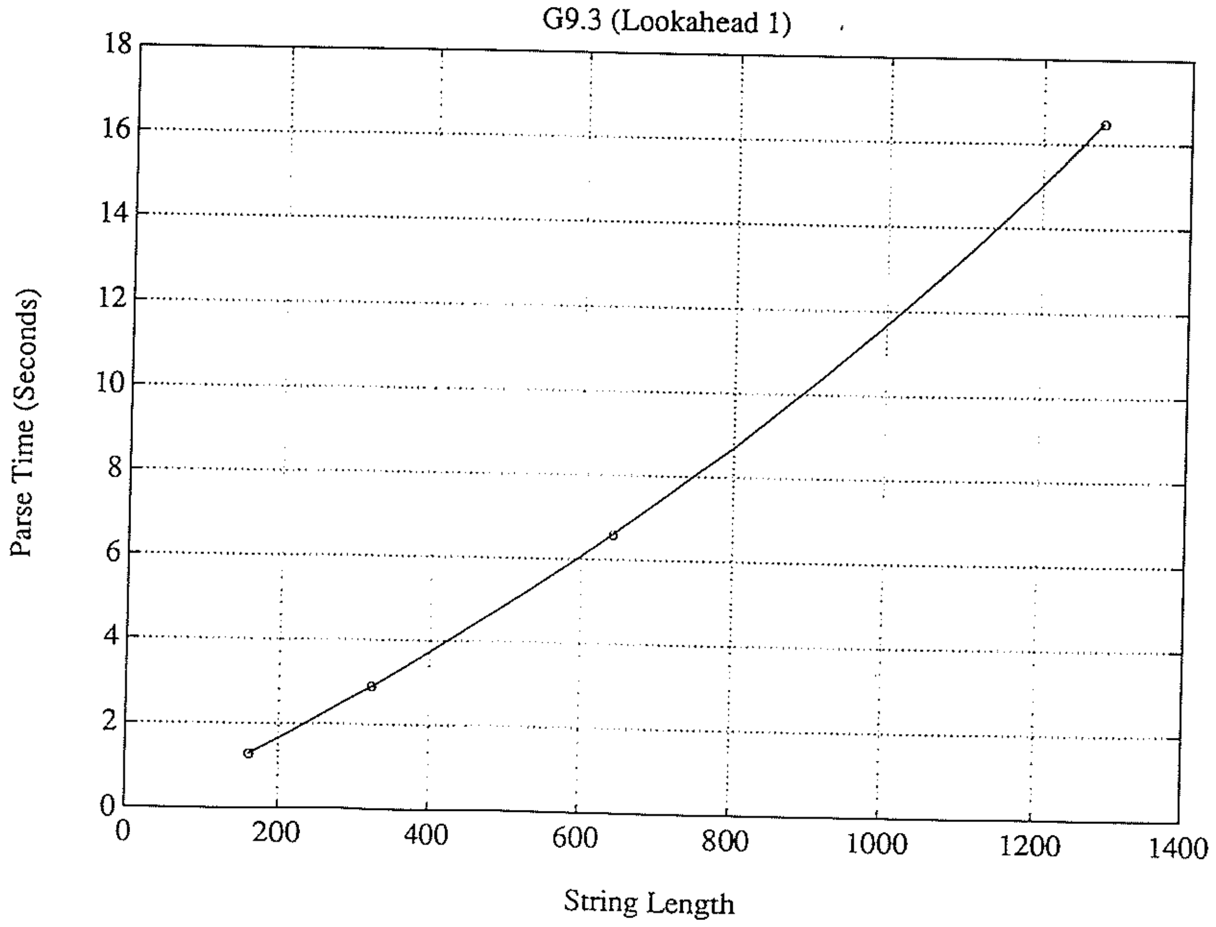


Figure 9.3: Performance of Grammar G9.3



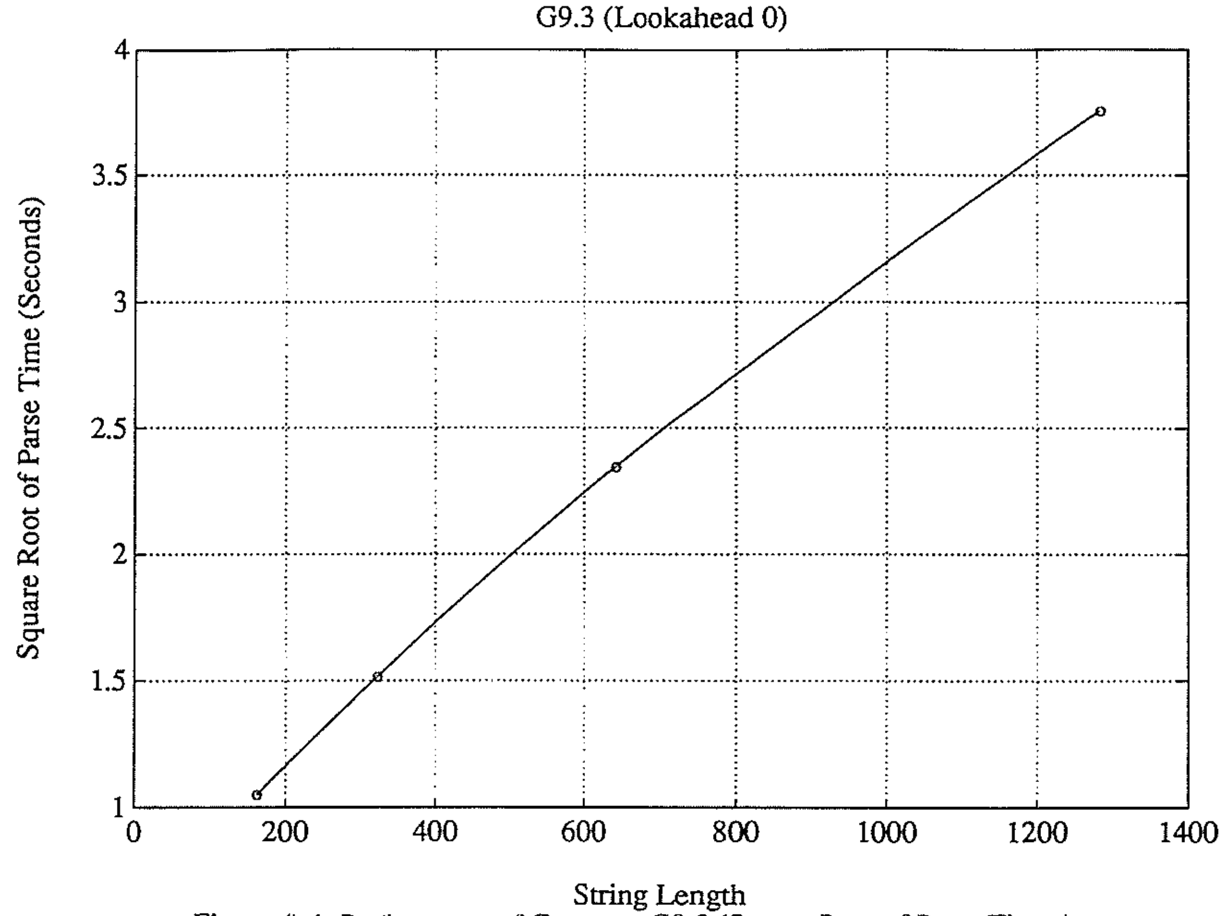
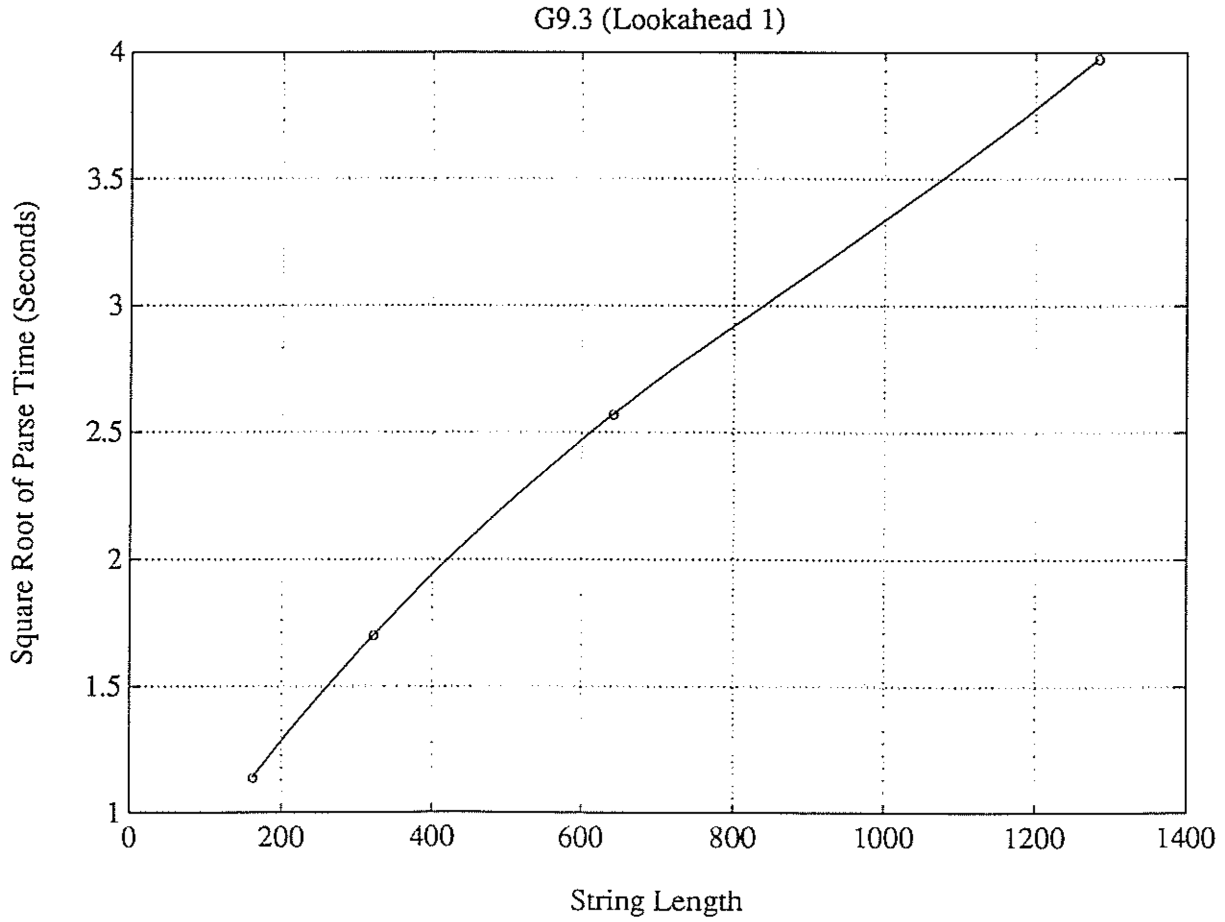


Figure 9.4: Performance of Grammar G9.3 (Square Root of Parse Times)



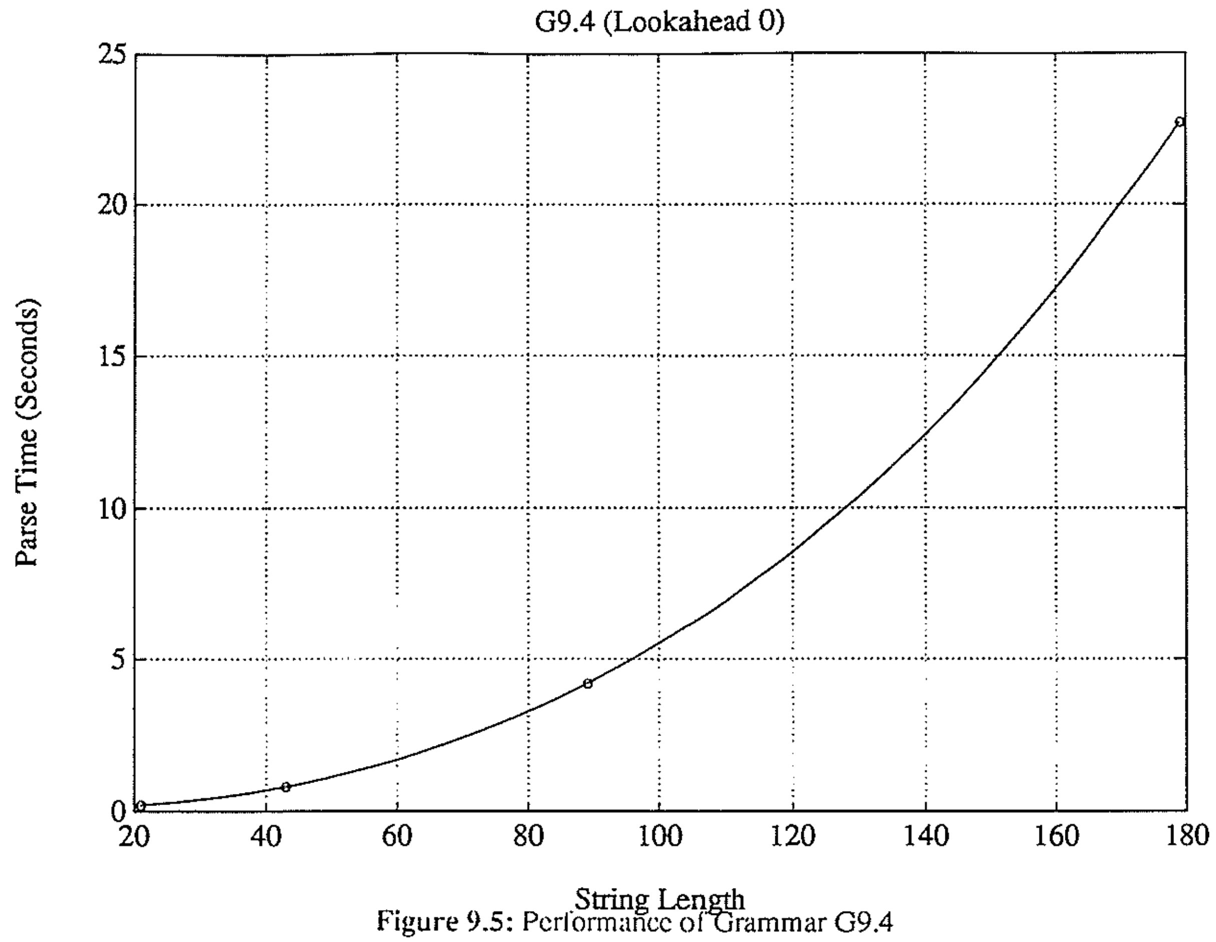
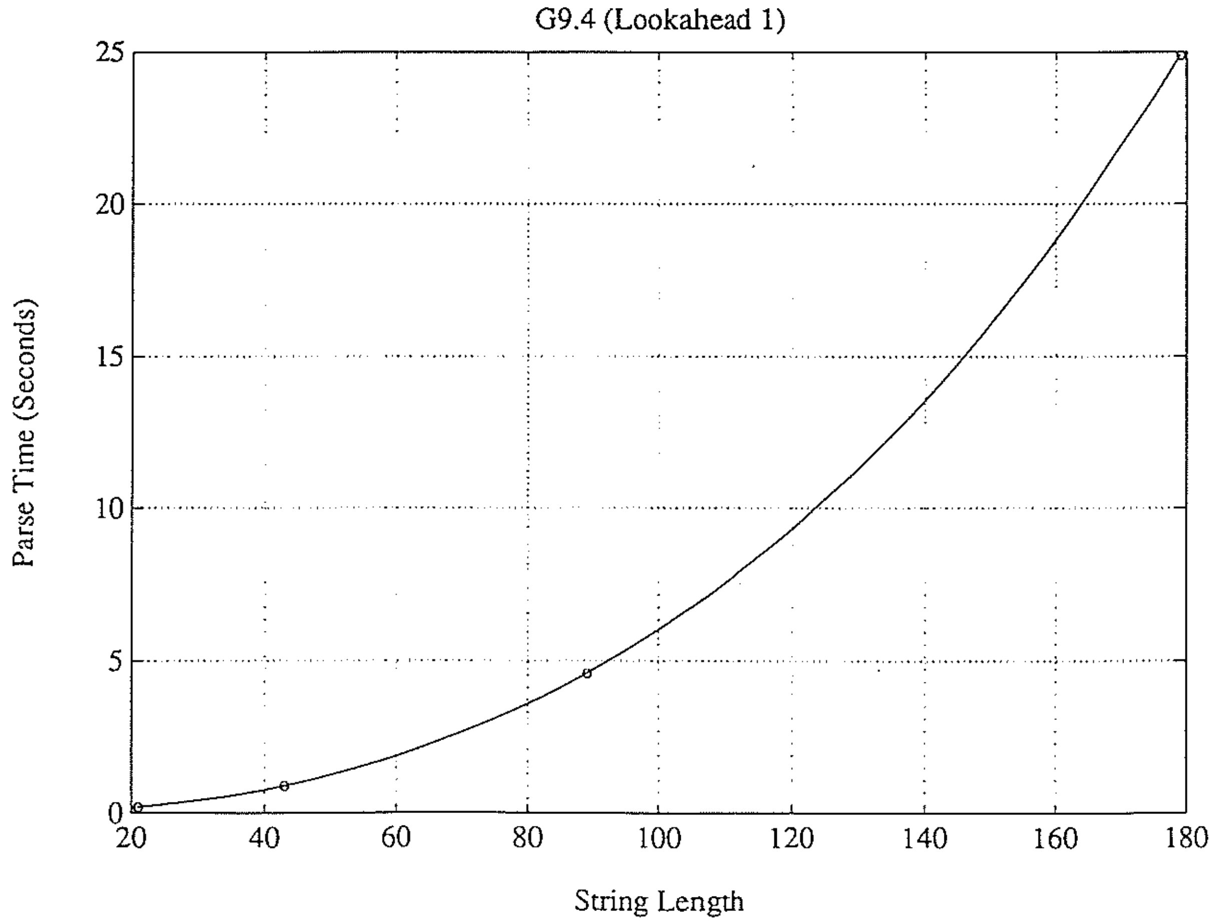


Figure 9.5: Performance of Grammar G9.4



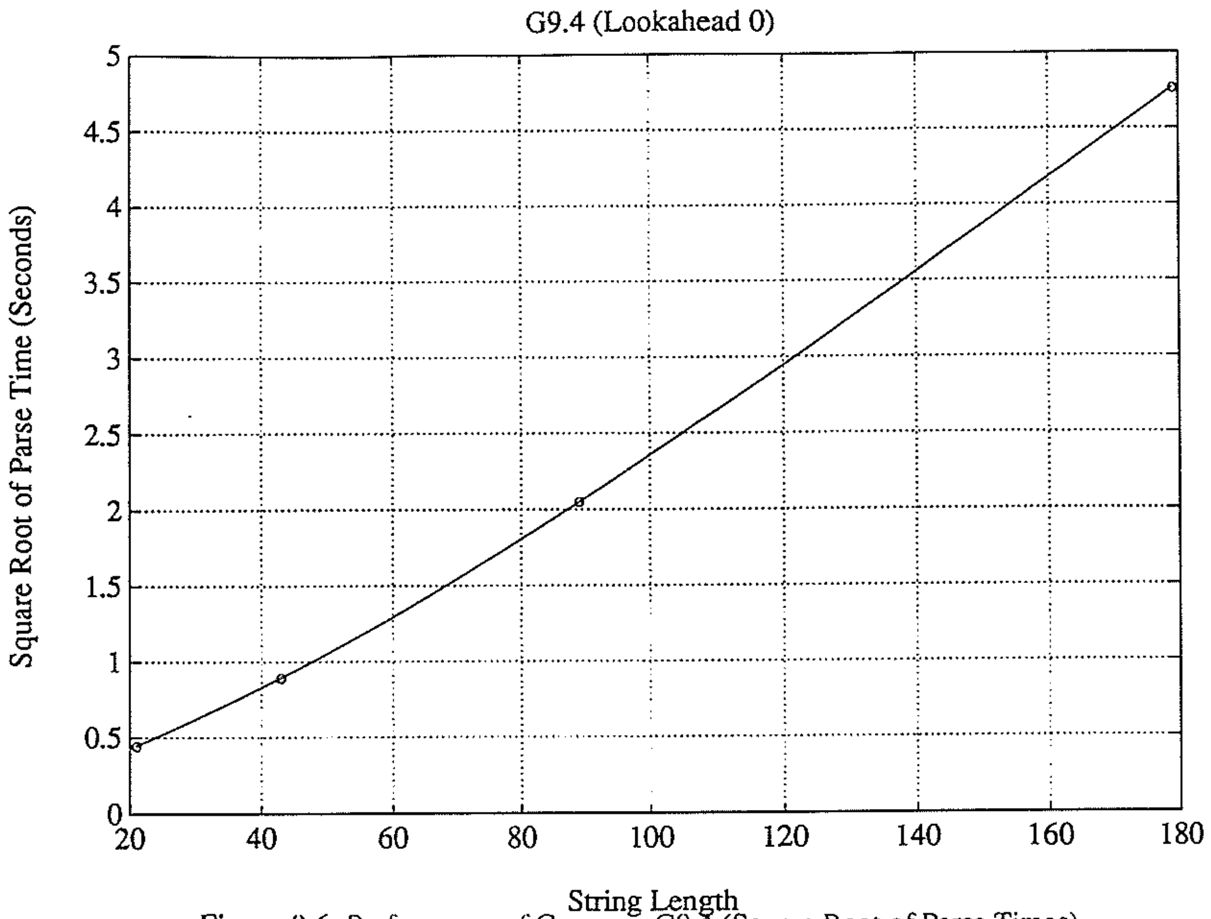
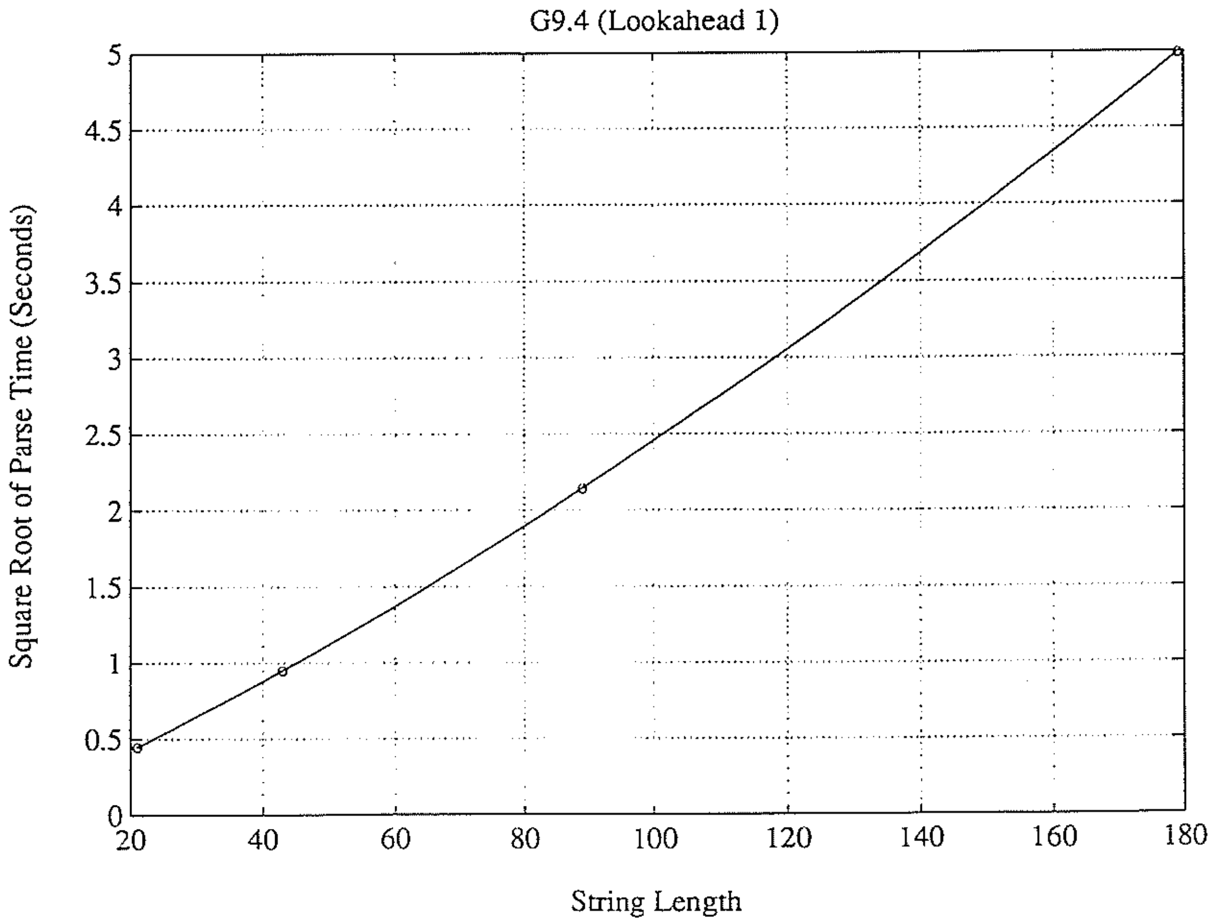


Figure 9.6: Performance of Grammar G9.4 (Square Root of Parse Times)



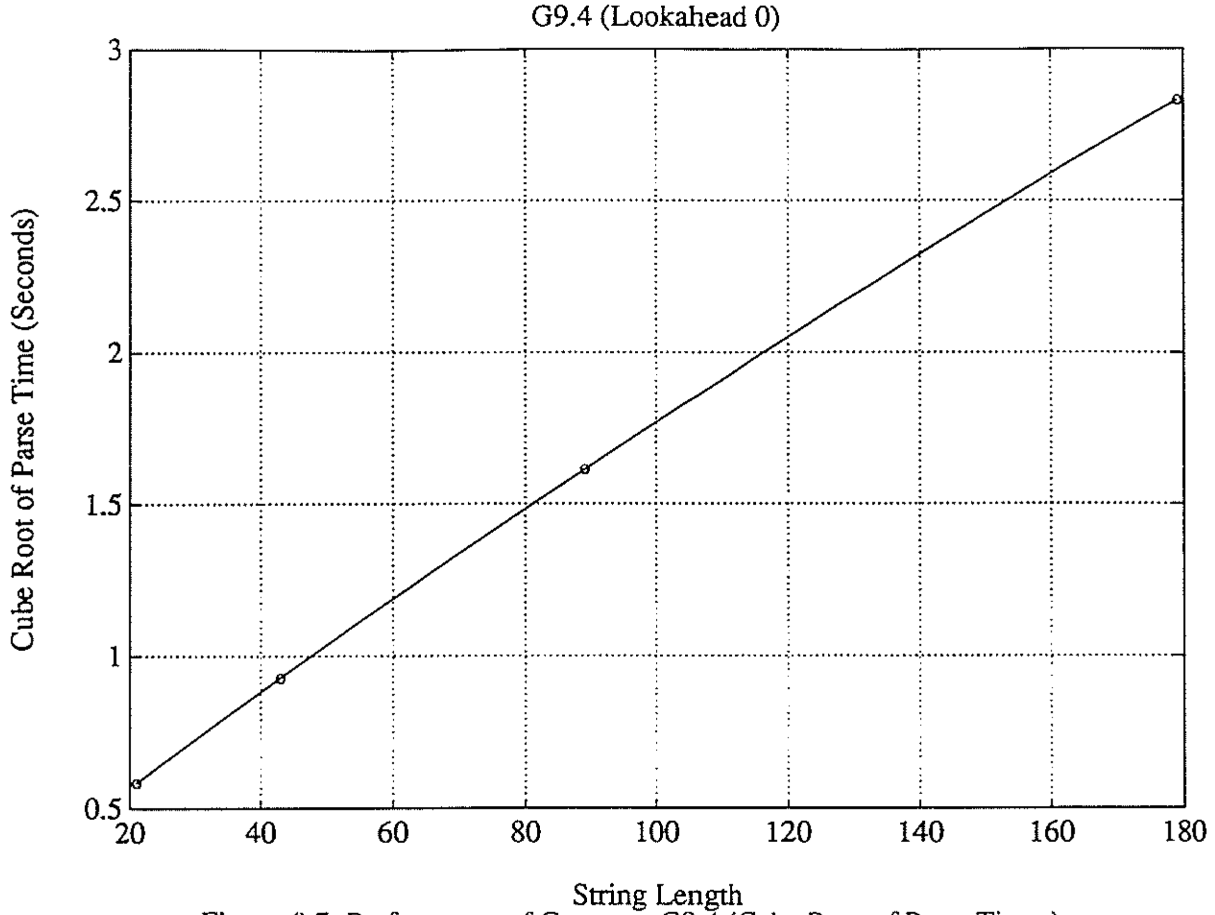
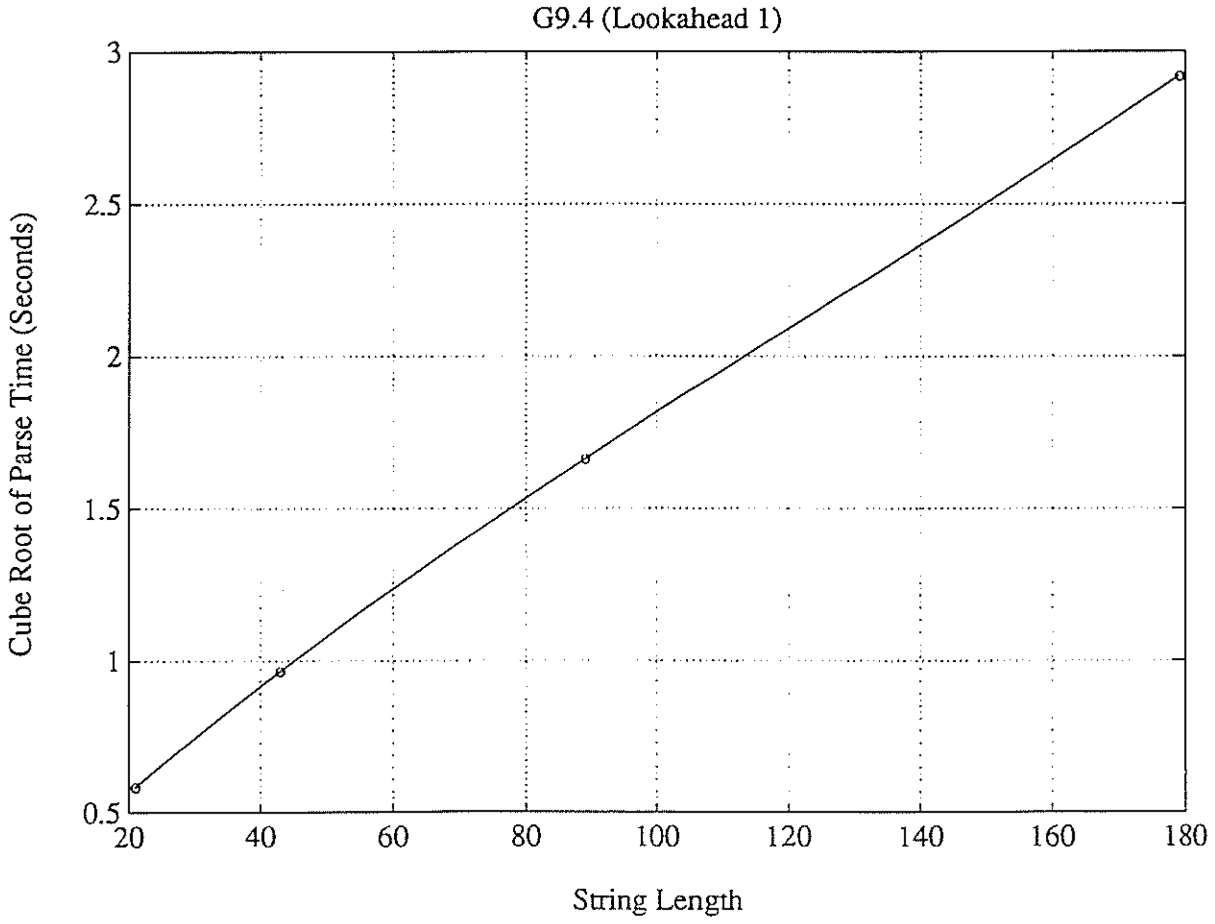


Figure 9.7: Performance of Grammar G9.4 (Cube Root of Parse Times)



For Grammars G9.1 and G9.2, the parsers are linear. This result is not surprising because these two grammars are unambiguous, and Earley's Algorithm is O(n) for most unambiguous grammars. Figures 9.1, and 9.2 give the graphs for these two parsers.

The parser for Grammar G9.3 is also linear. The graphs of this parser are displayed in Figure 9.3. Note that Figure 9.4 shows that the parser has a better performance than $O(n^2)$. This result is surprising because Grammar G9.3 is highly ambiguous. However, this result is the same as the one obtained by Earley in his thesis for this grammar.

The parser for Grammar G9.4 is $O(n^2)$. The graphs for this parser are given in Figure 9.5. The graphs of the string length versus the square root of the parse times are given in Figure 9.6, confirming the conjecture that this parser is $O(n^2)$. Note that the graphs given in Figure 9.7 show that the parser definitely has a better performance than $O(n^3)$. Again, this result is exactly the same as obtained by Earley for this grammar.

As the above results show, using a lookahead of 1 actually improves the order of the timing behavior of the parsers. In fact, there are grammars for which using a lookahead is necessary to get O(n) timings. One type of grammar that deserves special mention is a right-recursive grammar. Consider Grammar 9.5. If a PEG parser is created based on Grammar 9.5 using a lookahead of 1, then the parser has timing O(n). However, if no lookahead is used then the parser for Grammar 9.5 has timing between $O(n^2)$ and $O(n^3)$.

$$S \rightarrow a \mid a\, S$$

Grammar 9.5

10. Grammar for PEG

This section gives a pseudo YACC specification for the structure of the grammar specification file for PEG. The regular expressions that are used to identify the tokens, by the LEX generated lexical analyzer, are also given (as comments). In the actual YACC specification of PEG, there are semantic actions attached to the productions; these semantic actions are omitted from the listing given here.

```
%token OR /* "| " */
%token LB /* "<" */
%token RB /* ">" */
%token ZERO /* "0" */
%token SEMI /* ";" */
%token COLON /* ":" */
%token PERCENT /* "%" */
%token DELIMIT /* "%%" */
%token TOKEN /* "%token" */
%token TYPES /* "%type" */
%token START /* "%start" */
%token UNION /* "%union" */
%token TSIZE /* "%termtable" */
%token NSIZE /* "%nonttable" */
%token TOKEN_VAL /* [1-9][0-9]+ */
%token LOOKAHEAD /* "%lookahead" */
%token C_CODE /* "{C language code}" */
%token IDENT /* [a-zA-z](_[a-zA-Z0-9]|[a-zA-Z0-9])* */

%%
gram_start:
      tables start_sym look_union tokens nont_types
      header_code DELIMIT production_set user_routines
      ;

tables: table_defs
;

table_defs: table_def
      |
      ;

table_def: TSIZE TOKEN_VAL NSIZE TOKEN_VAL
      | NSIZE TOKEN_VAL TSIZE TOKEN_VAL
      | TSIZE TOKEN_VAL
      | NSIZE TOKEN_VAL
      ;
```

```
tokens: tokens TOKEN types IDENT TOKEN_VAL
      | TOKEN types IDENT TOKEN_VAL
      | tokens TOKEN type_tokens
      | TOKEN type_tokens
      ;

type_tokens: types name_list
      ;

nont_types: nont_type
      |
      ;

nont_type: nont_type TYPES LB IDENT RB name_list
      | TYPES LB IDENT RB name_list
      ;

name_list: name_list IDENT
      | IDENT
      ;

types: LB IDENT RB
      |
      ;

look_union: lookahead union_type
      | union_type lookahead
      | lookahead
      | union_type
      |
      ;

start_sym: START IDENT
      |
      ;

lookahead: LOOKAHEAD TOKEN_VAL
      | LOOKAHEAD ZERO
      ;

union_type: UNION C_CODE
      ;

header_code: PERCENT C_CODE
      |
      ;
```

```
production_set: production_set production_item
      | production_item
      ;

production_item: lhs COLON rhs SEMI
      ;

lhs: IDENT
      ;

rhs: alternate OR rhs
      | alternate
      ;

alternate: nonnull_alt sem_action
      | null_alt sem_action
      ;

nonnull_alt: nonnull_alt nonnull_item
      | nonnull_item
      ;

nonnull_item: IDENT
      ;

null_alt:
      ;

sem_action: C_CODE
      |
      ;

user_routines: DELIMIT C_CODE
      | DELIMIT
      ;
%%
```

11. Conclusions

Most automatic parser generators generate parsers from CFG specifications provided by the user. Usually these parser generators are based on LL(k) or LR(k) parsing algorithms. These parsers generators exhibit the disadvantages that are inherent in the parsing algorithms on which they are based. For example, ambiguous and cyclic grammars cannot be parsed by these algorithms. Furthermore, a parser generator based on LL(k) or LR(k) algorithms cannot parse CFGs that are lookahead greater than k.

Earley's Algorithm is a general parsing algorithm capable of parsing any arbitrary CFG. A parser generator based on Earley's Algorithm is a very powerful tool. In particular, the implementation discussed in this paper (PEG) is powerful because of the following reasons:

- PEG can create parsers from any arbitrary CFG of any lookahead, including cyclic and ambiguous grammars. Therefore, the programmer can write a grammar that is most natural to him/her and not have to change it in any way.

- Deferred semantic actions enable PEG to create parsers from non-LR(k) grammars that have an unbounded lookahead.

- A derivation graph is created during the parse and made available to the user. This derivation graph is a factored representation of all the possible derivation trees of a string under a CFG.

- Syntactically extendible languages can be implemented, by adding and deleting productions at execution time.

PEG is a parser generator that allows semantic actions to be embedded within the CFG specification. In that respect it is similar to YACC. However, PEG generates parsers that are fundamentally different from the parsers that YACC produces. As with any new tool, the real validation for PEG will come from the users as they experiment with applications that are made possible with this new parser generator.

Acknowledgments

I would like to thank Professor Will D. Gillett for his guidance during the implementation of this project. He initially suggested the idea of a parser generator based on Earley's Algorithm. I have used many software tools developed by him, to implement this parser generator. Throughout the project he provided invaluable advise, and most of the conceptual content of the project was developed with his help. I value his help as an advisor and his inspiration as a teacher.

Claudia Schaab very patiently corrected the numerous drafts of this report and provided moral support during the project implementation. I benefited immensely from her non-technical perspective on parsing, and the long philosophical discussions about computer science. Without her encouragement and help I would not have been able to complete this project.

I would like to thank Jim Daues, Liz Hanks, and Judy Lewis for putting up with my constant presence at their DNA Mapping Group. Deepak Bhargava and Bala Swaminathan were an attentive audience during my frequent need to discuss the implementation. Finally, I would like to thank Professor Stan C. Kwasny and Professor Seymour V. Pollack for their interest in my project as members of my project committee.

BIBLIOGRAPHY

UNIX Programmer's Manual, 5, CBS College Publishing , June 1974.

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman, Compilers: Principles, Techniques, and Tools, Addison Wesley, 1988.

Earley, Jay Clark, "An Efficient Context-Free Parsing Algorithm," Ph.d. Thesis, UMI Dissertation Services, August 1968. Carnegie-Mellon University, Pittsburgh.

Earley, Jay Clark, "An Efficient Context-Free Parsing Algorithm," CACM, pp. 94-102, 1970.

Firebaugh, Morris W., Artificial Intelligence, A Knowledge-Based Approach, Boyd & Fraser, 1988 . Boston.

Gillett, Will D., "Manipulating Computer Languages," Unpublished , 1989. Washington University, St. Louis.

Gillett, Will D., "DNA Mapping Algorithms: Abstract Data Types (ADTs)- Concepts and Implementation," Technical Report Number WUCS-91-33, Department of Computer Science, Washington University, St. Louis., June 1991.

Kernighan, Brian A. and Dennis M. Ritchie, The C Programming Language, Prentice-Hall, 1978.

Knuth, Donald E., "On the translation of languages from left to right," Information and Control, vol. 8, pp. 607-639, 1965.

Lewis, P. M., D. J. Rosenkrantz, and R. E. Stearns, Compiler Design Theory, Addison Wesley, May 1976.