INCREMENTAL SYNTACTIC AND SEMANTIC ANALYSIS
OF MODIFIED PROGRAMS

by
DASHING YEH

A thesis submitted in partial fulfillment
of the requirements for the degree of

DOCTOR OF ENGINEERING
(Computer Science)

at
Division of Computer Science
The Norwegian Institute of Technology
The University of Trondheim

BIBLIOTEK
December 1983
The thesis with the title \textit{INCREMENTAL SYNTACTIC AND SEMANTIC ANALYSIS OF MODIFIED PROGRAMS} by \textit{Dashing Yeh} for the degree of \textit{Doctor Engineering} has been approved by the censor committee consisting of

\textit{Professor Martti Tienari (sign.)} \hspace{1cm} ,

\textit{Professor Kristen Rekdal (sign.)} \hspace{1cm} , and

\textit{Assoc. Professor Arne Wang (sign.)} \hspace{1cm} .

Date: \textit{December 1983} .
FORORD

Emnet for denne avhandlingen er inkrementell syntaktisk og semantisk analyse av modifiserte datamaskinprogrammer. Hensikten er å utvikle nye metoder for å kunne lage bedre verktøy for å utføre slik analyse. Selv om dette er primært en teoretisk undersøkelse, forventes ingen vesentlige problemer med å anvende resultatene i praksis.


Det er mange mennesker som har hjulpet meg under mitt 4-årige opphold i Norge. Først og fremst er jeg stor takk skyldig min veileder Professor Kristen Rekdal, for alt han har gjort for meg. RUNIT, Regnesentret ved Universitetet i Trondheim, skaffet meg kontorplass, adgang til datamaskiner, og tillot meg å bruke biblioteket de siste 4 år. En spesiell takk går til forskere og sekretærer i Gruppe for språk og oversettere for deres vennlighet og assistanse av forskjellig slag. Stipendiet fra NORAD ble høyt verdsatt.

Sist, men ikke minst, vil jeg takke min kone som bidro til denne avhandlingen ett tusen og en dag og natt under mitt fravær i et fjernliggende land som en gang var hjemsted til vikingene og i tillegg er fødestedet til Simula!
ABSTRACT

It is highly desirable for an interactive programming environment to include tools capable of performing syntactic and semantic analysis of a modified program in an incremental way. This thesis is a theoretical study devoted to developing new methods for automatic generation of such tools.

We start by presenting a method of augmenting a conventional shiftreduce parser into an incremental one; to parse a modified input string, the resulting incremental parser requires about $3(m+n)$ space, and runs in time roughly proportional to $m$, where $n$ is the length of the original input and $m$ is the size of the modification made. We then extend the method to the construction of incremental LR(1) parsers which allow multiple modifications to be made to an already parsed input. To parse a modified input, such parsers require less than $(3m+6n)$ space and run in time bounded by $O(m + L \cdot \log H)$, where $m$ is the total size of the modifications, $L$ is the number of original input symbols which can not be skipped during incremental parsing, and $n$, $H$ are the length and parse tree height of the original input, respectively.

As an approach to incremental semantic analysis, we further propose a method of constructing incremental evaluators for ordered attributed grammars. Such evaluators are statically deterministic in the sense that the evaluation order of attributes is determined at construction time, and evaluate a modified semantic tree in time proportional to the number of attribute instances affected by the modification.
ACKNOWLEDGEMENTS

There are many people who have extended their help to me during my four years in Norway. It is unnecessary to list all their names here, because we all clearly know who they are and how much I owe them. However, there are two exceptions. One is Professor Kristen Rekdal who is the supervisor of this thesis. The other is my wife Yan-Qin Jiang who contributed to the thesis a thousand and one days and nights during my absence while I was studying in a very remote country, once homeland of the Vikings and the birthplace of Simula.

RUNIT, the Computing Centre at the University of Trondheim, kindly gave me office space and use of their library and computers throughout the past four years. The NORAD scholarship granted by the Norwegian Agency of International Development is gratefully appreciated.
# TABLE OF CONTENTS

Preface (in Norwegian) ........................................... i
Abstract ......................................................... ii
Acknowledgements ............................................... iii
List of Illustrations ........................................... vi

CHAPTER 1

INTRODUCTION .................................................. 1
  1.1 Motivation .............................................. 1
  1.2 Incremental Parsing and Attribute Evaluation ........... 2
  1.3 Purpose of the Thesis ................................. 4
  1.4 Outline of the Thesis ................................... 5

CHAPTER 2

INCREMENTAL SHIFT-REDUCE PARSING .......................... 6
  2.1 Introduction .......................................... 6
  2.2 Definitions and Notations ............................. 7
  2.3 A General Approach to Incremental Parsing ............ 10
  2.4 The Start of Incremental Parsing ...................... 12
  2.5 The Termination of Incremental Parsing ................. 17
  2.6 An Example .......................................... 20
  2.7 Previous Work ....................................... 22
  2.8 Summary and Concluding Remarks ...................... 24

CHAPTER 3

ON CONSTRUCTION OF INCREMENTAL LR(1) PARSERS ........... 26
  3.1 Introduction .......................................... 26
<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Parse tree for a shift-reduce parser</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Parse tree for a modified program</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Representation of input string</td>
<td>14</td>
</tr>
<tr>
<td>2.4</td>
<td>Input representation</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>Snapshot of input representation</td>
<td>15</td>
</tr>
<tr>
<td>2.6</td>
<td>Resulting input representation</td>
<td>16</td>
</tr>
<tr>
<td>2.7</td>
<td>Matching conditions during parsing</td>
<td>18</td>
</tr>
<tr>
<td>2.8</td>
<td>Parsed input representation of w</td>
<td>21</td>
</tr>
<tr>
<td>2.9</td>
<td>Parsed input representation of w'</td>
<td>22</td>
</tr>
<tr>
<td>2.10</td>
<td>Parsing with empty productions prese</td>
<td>25</td>
</tr>
<tr>
<td>3.1</td>
<td>Input representation</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>Matching reduction</td>
<td>34</td>
</tr>
<tr>
<td>3.3</td>
<td>Illustration for proof of Theorem 3</td>
<td>37</td>
</tr>
<tr>
<td>4.1</td>
<td>The OAG driver</td>
<td>50</td>
</tr>
<tr>
<td>4.2</td>
<td>The semantic tree for (NEW x:=1; x:=)</td>
<td>54</td>
</tr>
<tr>
<td>4.3</td>
<td>Incremental parsing</td>
<td>55</td>
</tr>
<tr>
<td>4.4</td>
<td>The incremental OAG driver</td>
<td>58</td>
</tr>
<tr>
<td>4.5</td>
<td>An example of incremental attribute evaluation</td>
<td>66</td>
</tr>
</tbody>
</table>
CHAPTER 1  INTRODUCTION

1.1 Motivation

Excessive cost and questionable reliability are two severe problems of software systems. As the size and complexity of systems increase, such problems are exacerbated at an ever greater rate. In response to this, more and more researchers devote themselves to the study of software engineering. Today, a number of methodologies have been proposed to aid in the effective production of high quality software. Among others, one approach is called the programming environment.

Generally speaking, a programming environment consists of a high-level programming language, a program database, and an integrated set of software tools such as editors, compilers, linkers, debuggers, etc. Traditionally, the purpose of a programming environment is to support the programmer in transforming system specifications into working programs. As a result, most existing programming environments are in their designs oriented more towards the initial software development than towards the software maintenance and evolution. Program modification is one of the major activities which are involved in the software maintenance and evolution (see, e.g., [4]). Whenever a program is modified in accordance with a particular maintenance requirement, the modified program must be completely recompiled to accommodate the modification made. Such a complete recompilation could be very time-
consuming or even intolerable, if program modification is a frequent exercise and modified programs are very large. In order to support the development as well as maintenance and evolution of large longer-lived programs, the concept of incremental programming environment was further raised (see, e.g., [25]). An incremental programming environment normally includes some tools especially suited for program modification so that a modified program can be recompiled in an incremental way.

1.2 Incremental Parsing and Attribute Evaluation

The need to incrementally compile programs was actually recognized many years ago (see, e.g. [22]), and various attempts have been made to achieve this goal ever since. However, it has recently become popular to realize incremental compilation by establishing an incremental programming environment, because the information necessary for performing incremental compilation is rather easily available in an environment where all the software tools are well integrated into one system (see, e.g., [9]).

A common feature of incremental programming environments is making use of a language-based or syntax-directed editor, which in essence is a combination of a regular text editor and a syntax analyzer or parser (see, e.g., [17]). Using a syntax-directed editor, the programmer can create and modify a program in terms of language constructs, rather than by editing characters and lines. After each modification the parser embedded in the editor automatically performs syntax analysis
over the modified program to assure its syntactic correctness. Naturally, in order to reduce the delay between two modification sessions, the parser must work in an incremental way. In other words, rather than completely reparsing the modified program, it will re-use the result of a preceding parse upon the condition that all those parts of the program which are affected by the modification are guaranteed to be re-analyzed. Parsers like that are called incremental parsers. In this thesis, we shall present a method of augmenting ordinary shift-reduce parsers into incremental ones. The method will subsequently be extended to the construction of incremental LR(1) parsers, which allow any number and form of modifications to be made to an already parsed input.

Besides incremental parsers, an incremental programming environment also needs to include incremental semantic analyzers to check the semantic consistency of a modified program. Incremental semantic analysis is no doubt more difficult than incremental parsing. A close study of the success with incremental parsing shows that context-free grammars as a syntax specification formalism has played a fundamental role in the automatic construction of parsers, both conventional and incremental. Therefore, if the generation of semantic analyzers is to be automated, then a semantic specification formalism is the first necessity. From the point of view of language implementors, attributed grammars appear to be very attractive for this purpose (see, e.g., [28]). Generally speaking, attributed grammars allow to specify the static semantics of programming languages in a declarative form, while leaving the semantic analysis phase of compilation to be accomplished in a process of attribute evaluation. From this property of attributed
It naturally follows that incremental semantic analysis can be achieved by constructing incremental attribute evaluators. In this thesis, we shall propose a method of constructing incremental evaluators for ordered attributed grammars. Such grammars have been shown in a series of experiments [19] to be quite suited for specifying the static semantics of practical programming languages.

Finally it must be pointed out that in addition to the application in building syntax-directed editors, incremental parsers and attribute evaluators may be also used in many other fields, such as incremental data flow analysis (e.g., cf. [8]), incremental code generation (e.g., cf. [10], separate compilation, high-level debugging, etc.

1.3 Purpose of the Thesis

This thesis is dedicated to developing new methods for making tools for incremental syntactic and semantic analysis of modified programs. Emphasis is placed on the enhancement and automatic generation of such tools. The thesis is primarily a theoretical study. However, a few implementation issues are also discussed since they are explicitly of theoretical interest as well.
1.4 Outline of the Thesis

By topics, this thesis can be divided into two parts, one dealing with incremental syntactic analysis and the other with incremental semantic analysis. Specifically, the former consists of Chapters 2 and 3, and the latter of Chapter 4.

In Chapter 2 we shall propose a method of augmenting a conventional shift-reduce parser into an incremental one. In Chapter 3, we shall extend the method to the construction of incremental LR(1) parsers allowing multiple modifications. In Chapter 4, we shall propose a method of constructing incremental evaluators for ordered attributed grammars. Finally, Chapter 5 summarizes this research and indicates some directions for further work. At the end of the thesis, there are two appendices.

Chapters 2 and 4 have been published in an abridged form ([32,33]). Chapter 3 has been submitted for publication ([34]).

The reader is assumed to be familiar with the basic theory of context-free grammars and attributed grammars. However, for his convenience, Chapters 2, 3 and 4 each begin with a brief account of the prerequisites for the topic concerned.

Unless otherwise stated, the terminology and notation of [1] are used throughout the thesis.
CHAPTER 2 INCREMENTAL SHIFT-REDUCE PARSING

2.1 Introduction

Most present-day compilers work in the following way: input a source program and compile it into object code; if the source is modified (e.g., some symbols or lines are deleted, inserted, or substituted), then the modified program has to be recompiled as a whole even if the modification only involves a few symbols. Likely, much time could be saved if the new code can be obtained by recompiling a limited portion of the source which contains the modification and by using the remaining code as well. This is the way in which so-called incremental compilers work. To construct an incremental compiler brings forward the problems of an incremental approach to the main phases, i.e., lexical analysis, syntactic parsing, semantic analysis, and code generation.

In this chapter, we are only concerned with the problem of incremental parsing. Specifically, we will present a method to augment an ordinary shift-reduce parser into an incremental one.

The problem of incremental shift-reduce parsing was attacked in [3, 12, and 13], respectively. In [12] and [13], the nodes of a parse tree are in a special way threaded so that the stack contents at any point during parsing can be acquired if the concerned input symbol is given. Because parse trees are used, the incremental parsers given in
[12] and [13] require an amount of storage which is nonlinear with respect to the length of an input to be parsed. From the work mentioned above, Celentano [3] derived an incremental parser which used a so-called stack tree to represent the input. In his view, however, the incremental parser is impractical due to the need for a large amount of storage and is expensive in time during the search for a match between the old and new structures. To improve this, he suggested that the stack tree should represent only shift moves. This idea inspired us to design a new data structure to represent an input and its parsing. In terms of this data structure, an incremental parser is devised which uses about \(3(m + n)\) space and \(O(m)\) time to parse a modified input, where \(n\) is the length of the original input and \(m\) is the size of the modification made.

The rest of this chapter is organized as follows: section 2.2 contains the definition of shift-reduce grammars; section 2.3 reduces the problem of incremental parsing into three subproblems; then they are in turn solved in sections 2.4 and 2.5; a simple example is given in section 2.6; section 2.7 reviews the previous work; and finally a summary and several remarks are given in section 2.7.

2.2 Definitions and Notations

The reader is assumed to be familiar with the basic theory of context-free grammars and their parsing. For his convenience, however, the definition of shift-reduce grammars is duplicated below (see [1], pp. 400-401).
Let \( G = (N, \Sigma, S, P) \) be a context-free grammar, where \( N \) is the set of nonterminal symbols, \( \Sigma \) is the set of terminal symbols, \( S \) is the start symbol, and \( P \) is the set of productions. It is assumed that the productions are numbered from 1 to \( p \).

A shift-reduce parser for \( G \) is a pair of functions \((f, g)\), where \( f \) is called the shift-reduce function and \( g \) the reduce function, respectively. These two functions are defined as follows:

1. \( f : V^*X(\Sigma \cup \{\$\})^* \rightarrow \{\text{shift, reduce, error, accept}\} \), where \( V = N \cup \{\$\} \), and \( \$ \) is a new symbol, i.e., endmark.

2. \( g : V^*X(\Sigma \cup \{\$\})^* \rightarrow \{1, 2, \ldots, p, \text{error}\} \), under the constraint that if \( g(\alpha, u) = i \), then the right hand side of production \( i \) is a suffix of \( \alpha \).

To parse an input string, the shift-reduce parser uses a stack and scans the input from left to right. At each step, the function \( f \) is used to decide whether to shift the current input symbol on the stack or call for reduction, depending on what is on the stack and what remains on the input. If a reduction is called, then the function \( g \) is invoked to decide what reduction to make.

We shall describe the action of the shift-reduce parser in terms of configurations which are of the following form:

\[
(\$, X_1, X_2, \ldots, X_m, a_1, a_2, \ldots, a_n, \$, p_1, p_2, \ldots, p_r),
\]

where

1. \( X_1, X_2, \ldots, X_m \) represents the stack, with \( X_m \) on top. Each \( X_i \) is in \( V \) and \( \$ \) acts as the bottom of the stack.

2. \( a_1, a_2, \ldots, a_n, \$ \) is the remaining portion of the input, \( a_i \) is the symbol under scan, i.e., the current symbol, and \( \$ \) acts as the right endmark for the input.
(3) \( p_1, p_2, \ldots, p_r \) is the sequence of production numbers used to reduce the original input into \( X_1, X_2, \ldots, X_m, a_1, a_2, \ldots, a_n \).

Below is the action of the shift-reduce parser, described in terms of two relations \( \vdash_r \) and \( \vdash_s \) on configurations:

(1) If \( f(\alpha, au) = \text{shift}, \) then

\[
(\alpha, au, \pi) \vdash_s (\alpha \ a, u, \pi)
\]

for all \( \alpha \) in \( V^* \), \( u \) in \( (L \cup \{\$\})^* \), and \( \pi \) in \( \{1, 2, \ldots, p\}^* \).

(2) If \( f(\alpha \beta, au, \pi) = \text{reduce} \), \( g(\alpha \beta, au) = i \), and production \( i \) is \( B \rightarrow \beta \), then

\[
(\alpha \beta, au, \pi) \vdash_r (\alpha \ B, au \ \pi).
\]

(3) If \( f(\alpha, u) = \text{accept} \), then

\[
(\alpha, u, \pi) \vdash_s \text{accept}.
\]

(4) Otherwise, \( (\alpha, u, \pi) \vdash_s \text{error} \).

Furthermore, we define relation \( \vdash^* \) as the union of \( \vdash_r \) and \( \vdash_s \). Relation \( \vdash^* \) is derived from \( \vdash \) as usual.

For any \( w \in L^* \), if there is sequence of production numbers \( \pi \) such that

\[
(\$, w \$, e) \vdash^* (\$, S, \$, \pi) \vdash^* \text{accept},
\]

where \( e \) is the empty string, then \( \pi \) is called the parse sequence of \( w \), otherwise the parse sequence of \( w \) is said to be \( \text{error} \). A shift-reduce parser is valid for \( G \) if

\[
L(G) = \{ w \mid \text{the parse sequence of } w \neq \text{error} \}.
\]

On the other hand, if there is a valid shift-reduce parser for \( G \), then \( G \) is said to be shift-reduce parsable. It must be pointed out that not every context-free grammar is shift-reduce parsable, but the class of shift-reduce parsable grammars is a large subclass of context-free grammars and it includes, e.g., all LR(k) grammars.
In this chapter, since we exclusively consider shift-reduce parsable grammars and their parsers, the modifier "shift-reduce" will be omitted whenever possible.

2.3 A General Approach to Incremental Parsing

In this section we will discuss how to achieve incremental parsing in general. Let $G = (N, I, S, P)$ be a shift-reduce parsable grammar and assume that $G$ contains no productions with the empty right-hand side.

Suppose that $w = xyz$ and $w' = xz'y$ are in $L(G)$, and that $w$ has been parsed by a shift-reduce parser, yielding the parse tree shown in Figure 2.1.

![Parse tree for a shift-reduce parser](image)

Figure 2.1 Parse tree for a shift-reduce parser

As $w'$ can be obtained from $w$ by substituting $z'$ for $z$, it is natural to ask what similarities there are between the parse trees, or equivalently, the parse sequences of $w$ and $w'$. According to the literal definition of parsers, almost nothing can be said for certain
because the functions $f$ and $g$ may in the most general case depend effectively on the entire stack contents and the whole remaining input, and consequently, in the extreme case, it is possible that the parsing of $w'$ is completely different from that of $w$ although only one single symbol is changed. Therefore, there is no possibility to achieve incrementality for those most general shift-reduce parsers.

Fortunately, parsers encountered in practice are not so "bad". In fact, practical functions $f$ and $g$ usually depend only on the top few symbols on the stack and the next few input symbols. For instance, an $LR(k)$ parser depends only on the topmost symbol on the stack and the next $k$ input symbols. Therefore, the substitution of $z'$ for $z$ in $w$ can only give rise to a local effect on the parsing of $w'$. Specifically, the moves made in parsing $w'$ are the same as in parsing $w$, except when the current symbol falls in $z'$ or in the neighbourhood of $z'$. In terms of the parse tree of $w'$, this can be illustrated in Figure 2.2, where the shaded area is the same as in the parse tree of $w$ (see footnote 1 below), $x = x_0 x_1$ and $y = y_1 y_0$. Note that $x_0$, $x_1$, $y_0$ and $y_1$ may all be empty. Particularly, if both $x_0$ and $y_0$ are empty, then the parse tree of $w'$ is totally different from that of $w$.

Therefore, if we can identify substrings $x_1$ and $y_1$ shown in Figure 2.2, then we will be able to obtain the parse tree of $w'$ by only parsing a part of $w'$, i.e., $x_1 z' y_1$, and by using the remaining parse tree of $w$ as well. This is the incremental parsing of $w'$ that we are looking for. It is justified only when the time and space used for identifying $x_1$ and $y_1$ are a profitable trade-off in comparison with the complete parsing of $w'$. 
Thus, the problem of incrementally parsing \( w' \) has been reduced to the following subproblems:

1. Where is the first symbol of \( x_1 \)? It is equivalent to the question where to start the incremental parsing of \( w' \).

2. How does the incremental parser get started?

3. Where is the final symbol of \( y_1 \)? It is equivalent to the question where to terminate the incremental parsing of \( w' \).

We will in turn solve these subproblems in the next two sections.

### 2.4 The Start of Incremental Parsing

We begin with solving the first subproblem.

As said in section 2.3, practical shift-reduce functions usually

---

1) Comparing Figure 2.2 with Fig. 4 in [13], we can see the difference between the approaches here and there. In [13] the parsing of \( x_1 \) is assumed intact by the substitution of \( z' \) for \( z \), while no such assumption is made here.
depend only on the next few input symbols and likewise for practical reduce functions. The number of symbols to look ahead for may vary from step to step during a parsing process, but their maximum must be reasonable for the sake of practicability. Let us denote it by $K$; then $K$ is $k$ for LR($k$) parsers for instance. It is clear that the moves made in parsing $w'$ are the same as in parsing $w$ until the current symbol is $K$ symbols before $z'$. Only from there on the parsing of $w'$ might differ from that of $w$, although they actually differ later in some cases. Let us denote the symbol which is $K$ symbols before $z'$ by SYMB($K$, $z'$). It is easy to see that SYMB($K$, $z'$) can be taken as an approximation to the first symbol of $x'$, denoted by FIRST($x'$). They may be the same in some cases. Since $K$ in practice is rather small, we see that SYMB($K$, $z'$) is a fairly good approximation and it has saved us the effort to exactly identify FIRST($x'$).

By the mechanism of the shift-reduce parser, it is clear that the only request for the incremental parser to be able to start parsing $w'$ at SYMB($K$, $z'$) is the stack contents when SYMB($K$, $z'$) is the current symbol in parsing $w$. Therefore, the solution to the second subproblem consists in obtaining those stack contents. Of course, they can be acquired from the process of parsing $w$. But, as ordinary data structures for storing an input and its parse are not well suited for our purpose, we will design a new data structure and augment the parser accordingly.

In order to make the main idea more clear, we will resort to illustrations instead of a formal treatment.

The input is represented by a list of circle nodes where each node contains an input symbol. For example, the input $w = \$ b_1 b_2 \ldots b_n \$ can be represented as shown in Figure 2.3.
Figure 2.3  Representation of input string.

Associated with each circle node there is a rectangle node which is composed of two fields called LABEL and LINK, respectively. For each rectangle node, LABEL is initially set to the symbol contained in the associated circle node, and LINK is set to point to the LINK field of the preceding input symbol, except that the LINK field of the left endmark \$ is set to \( A \), a specially defined value. We call the whole data structure an input representation (see footnote 2). For example, the input representation of \( w \) is shown in Figure 2.4.

Figure 2.4  Input representation

In an input representation, the circle nodes are used to store the input symbols. Once a circle node is assigned a symbol, it will not be re-assigned. Therefore, we will refer to a circle node by its assigned

---

2) An input representation should also have a part to store the production numbers invoked so far (see, e.g., [1]). Here it has been deliberately omitted for simplifying the discussion.
symbol. On the other hand, the use of the rectangle nodes is twofold. They are used as working storage during a parsing process. When the process is ended, they contain the result of the parsing as well as the information about the stack evolution through the whole process. Thus, by means of the input representation, we can get the stack contents at any point during the parsing process.

Our intention is to get the stack contents at the moment when a given input symbol is the current one during parsing w by collecting the LABEL fields along the LINK chain starting at the symbol preceding the given one and ending up with the left endmark $. To this goal, the ordinary parser will be augmented as described below.

As in section 2.2, the action of the augmented parser will be described in terms of configurations. Suppose that the current configuration is of the form

$$\left( \$ X_1 X_2 \ldots X_m b_{i+1} \ldots b_n \$, p_1 p_2 \ldots p_r \right),$$

where $X_1 X_2 \ldots X_m$ is a string in $\{N U I\}^*$ to which $b_1 b_2 \ldots b_i$ has been reduced. Figure 2.5 is a snapshot of the input representation at that moment.

![Figure 2.5 Snapshot of input representation.](image)
Figure 2.6 Resulting input representation.

The action of the augmented parser is as follows:

1. If the value of the function \( f \) is shift or accept or error, then perform the same action as the ordinary parser and do nothing with the input representation.

2. If the value of the function \( f \) is reduce, that of the function \( g \) is \( j \) and production \( j \) is \( B \rightarrow \alpha \) where \( \alpha \) is of the length \( s \), then in addition to the action taken by the ordinary parser,
   
   2.1 set LABEL of \( b_i \) to \( B \), and
   
   2.2 traverse \( s \) LINK's along the chain starting at \( b_i \). Let the symbol reached be \( b_k \). Change LINK of \( b_i \) to that of \( b_k \). The resulting input representation is shown in Figure 2.6.

It is easy to see that, when the parsing of \( w \) is finished, the input representation exactly becomes what we want it to be, i.e., for any input symbol, we can get the stack contents when the symbol is the current one from the input representation by collecting LABEL's along the LINK chain starting at the symbol preceding the given one and ending up with the left endmark \( \$ \). Thus, we have also solved the second subproblem.
2.5 The Termination of Incremental Parsing

Comparing Figure 2.1 with Figure 2.2, we can see that, when the incremental parsing of \( w' \) terminates at the final symbol of \( y_1 \), there must be such a nonterminal \( A \) that \( A \) is the root of the subtree having the frontier \( x_1 z y_1 \) in the parse tree of \( w \) as well as the root of the subtree having the frontier \( x_1 z'y_1 \) in that of \( w' \), and vice versa. (Note that \( A \) may be the start symbol \( S \)). In terms of derivations, this can be expressed as the following matching conditions:

If

\[
\begin{align*}
(1) & \quad S \Rightarrow^{\pi_0}_{\text{LM}} \alpha A y_0 \Rightarrow^{\pi_1}_{\text{LM}} \alpha x_1 z y_1 y_0 \Rightarrow^{\pi_2}_{\text{LM}} x_0 x_1 z y_1 y_0 = x z y \\
(2) & \quad A \Rightarrow^{\pi'_1}_{\text{LM}} x_1 z'y_1
\end{align*}
\]

and

hold for some sequence of production numbers \( \pi_0, \pi_1, \pi_2 \) and \( \pi'_1 \), then

\[
S \Rightarrow^{\pi'}_{\text{LM}} x z'y \text{ with } \pi' = \pi_0 \pi'_1 \pi_2, \text{ where } \Rightarrow^{\pi}_{\text{LM}} \text{ means a rightmost derivation.}
\]

The matching conditions should be tested every time a reduction is made after the final symbol of \( z' \) is scanned. Looking into the input representations of \( w \) and \( w' \), we can see that the matching conditions hold when and only when a following reduction is made:

1. The current symbol, \( b_{i+1}' \), is situated to the right of \( z' \), and LABEL of \( b_i \) in the input representation of \( w \) is the same as in that of \( w' \).

2. LINK of \( b_i \) in the input representation of \( w \) is also the same as in that of \( w' \), pointing to the LINK field of an input symbol which is situated to the left of \( z \) (or equivalently, to the left of \( z' \) in the input representation of \( w' \)).

These two conditions are illustrated in Figure 2.7, where

\[
z = c_1 c_2 \ldots c_h \text{ and } z' = d_1 d_2 \ldots d_q.
\]
Figure 2.7 Matching conditions during parsing.

Now we are able to present our incremental parser as a whole, which is stated as the following algorithm.

**ALGORITHM 1.1.** The incremental parser.

**INPUT.** The parsed input representation of \( w = xz'y \) and still unparsed one of \( w' = xz'y \).

**OUTPUT.** The parsed input representation of \( w' \).

**METHOD.** It consists of the following steps:

1. By using the input representation of \( w \), set the stack to have the same contents as when \( \text{SYMB}(K, z') \) is the current symbol in parsing \( w \).
2. Using the augmented parser, parse \( w' \) from \( \text{SYMB}(K, z') \) to the final symbol of \( z' \).
(3) Continue parsing $w'$ and test the matching conditions every time a reduction is made.

(4) If the matching conditions hold, then replace the remaining portion of the input representation of $w'$ by that of $w$. Stop.

In implementing this algorithm, it is more space-efficient to use a combined input representation for the common part of $w$ and $w'$ while keeping the separate references to the rest. Moreover, by forward linking the input symbols, the incremental parser can be improved so as to skip those parts of $y_1$ which have the same parsing in $w'$ as in $w$ (we will show this improvement for incremental LR parsers in chapter 3, although it is in essence applicable for all shift-reduce parsers).

We conclude this section by a brief discussion on the performance of the incremental parser as compared with that of an ordinary parser.

The incremental parser requires about $3(m+n)$ space to store a modified input representation and its original, on the assumption that an ordinary parser uses $n$ space for storing the original input of the length $n$ and the size of the modification is $m$.

Concerning the time complexity, two factors must be taken into account. On the one hand, the incremental parser needs additional time to process the rectangle nodes, as compared with an ordinary parser. On the other hand, the incremental parser only parses a part of $w'$, i.e., $x_1 z' y_1$. Among others, $z'$ must be parsed anyhow, and $x_1$ usually involves few symbols, as said in section 2.3. Therefore, the time performance of the incremental parser depends mainly on the length of $y_1$. Unlike $x_1$, the length of $y_1$ may vary a good deal, depending on both $G$ and $w'$. Nevertheless, since the modularity and
locality are two of the dominant considerations in designing programming languages and writing programs, it is normally not the case that a relatively small change made in a program syntactically spreads its influence so widely that the parsing of the modified program is drastically different from that of the original one. Therefore, in most ordinary cases, the length of $y_1$ is quite small in comparison with that of $w'$. Thus, by incorporating the improvement mentioned above, we can say in a sense that the incremental parser runs in time roughly proportional to the length of $z'$, or the size of the modification made. To see the net gain in time by using the incremental parser requires further work, both theoretical and practical.

2.6 An Example

As a simple example, we consider the following grammar $G_0 = (N, \Sigma, S, P)$, where $N = \{E, T, F\}$, $\Sigma = \{+, *, (, ), i\}$, and $P$ consists of the following productions:

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow i$

A valid parser for $G_0$ is devised with the following functions $f$ and $g$:

1. $f(SE, E) = \text{accept}$
2. $f(a_i, a) = \text{reduce}$ $g(a_i, a) = 6$ for $a$ in $\{+, *, (, )\}$
3. $f(\alpha(E), a) = \text{reduce}$ $g(\alpha(E), a) = 5$ for $a$ in $\{+, *, (, )\}$
Figure 2.8  Parsed input representation of w

4. $f(αE+T, a) = \text{reduce} \quad g(αE+T, a) = 1 \quad \text{for } a \text{ in } \{+, \), $\}$
5. $f(αT*F, a) = \text{reduce} \quad g(αT*F, a) = 3 \quad \text{for } a \text{ in } \{*, \}\
6. $f(α+F, a) = \text{reduce} \quad g(α+F, a) = 4 \quad \text{for } a \text{ in } \{+, \}\
7. $f(\$T, \$) = \text{reduce} \quad g(\$T, \$) = 2$
8. $f(\$F, \*) = \text{reduce} \quad g(\$F, \*) = 4$
9. $f(α(T, )) = \text{reduce} \quad g(α(T, )) = 2$
10. $f(αE, a) = \text{shift} \quad \text{for } a \text{ in } \{+, \}$
11. $f(αT*, a) = \text{shift} \quad \text{for } b \text{ in } \{i, \}$
12. $f(αa, b) = \text{shift} \quad \text{for } a \text{ in } \{, *, \} \text{ and } b \text{ in } \{i, \}$
13. $f(\$, b) = \text{shift} \quad \text{for } b \text{ in } \{i, \}$

Finally, $f$ takes the value \textbf{error} for all other values of the arguments.

Now let $w' = i+(i+i*_{i})*_{i}$ be a modification of $w = i+(i+i*_{i}$, obtained by inserting $i_{i}$ between the third $i$ and the right parenthesis $)$, i.e., $z' = *_{i}$ and $z = e$, the empty string. Using the augmented parser, we obtain the parsed input representation of $w$ shown in Figure 2.8.

Using the incremental parser, we obtain the parsed input representation of $w'$ shown in Figure 2.9, where the dashed line encircles the part which is actually reparsed.
2.7 Previous Work

To the author's knowledge, C. Ghezzi and D. Mandrioli seem to be the first persons who attacked the problem of incremental parsing. In 1976, they proposed a method of augmenting LR and LL parsers into incremental ones. The method was described in two interim reports which were later on published in [12] and [13], respectively. From this work, A. Celentano derived another incremental parser which differs from those exposed in [13] in the kind of information selected to represent a parse, i.e., the former uses a sequence of parsers' moves while the latter uses the syntactic tree. Both works have been briefly commented in section 2.1.

As part of a syntax-directed editor, J. Morris and M. Schwartz [27] constructed an incremental parser which is an extension of the
traditional LL(1) parsing algorithm. The LL(1) parse tables are augmented in order to treat entire subtrees of a parse tree as terminal in some cases. A series of parse trees are maintained with links to the text buffer. The text may be modified provided it obeys the "language constraint", viz., the text up to the token immediately preceding the cursor must be the legal beginning of a program. The text between two such discontinuities is covered by a separate parse tree. Whenever a command attempts to move beyond a discontinuity, the parse trees are patched back together using an extension of traditional LL(1) parsing. However, full reparsing may be necessary in many cases.

Based on operator precedence grammars, G. Kaiser and E. Kant [18] proposed an incremental algorithm for parsing expressions involving only binary operators. Except for syntax trees, no other data structures are required. After each modification is made, the incremental parser directly performs tree transformations on a syntax tree to enforce the correctness of the tree in terms of the operator precedence rules. Their method is quite attractive in its novelty. One of its disadvantages is that they have not yet worked it out to cover a more general class of languages than those handled by operator precedence parsers.

Noticeably, F. Jalili and J.H. Gallier [16] proposed a method of constructing incremental LR(1) parsers which allow any number and form of modifications to be made to an already parsed input. To parse a modified input, such parsers use a series of specially defined subtree operations, manipulating a parse tree where each node is added a field to record the state information. In their words, the condition for reusing a subtree of the parse tree is that the state in the preceding
node matches the state on top of the stack after modifications. As we shall see in Chapter 3, this is a sufficient condition but not a necessary one for reusing the subtree. Consequently, such incremental parsers may fail to catch many good chances of reusing the result of a preceding parse.

2.8 Summary and Concluding Remarks

In this chapter, a general approach is taken to the problem of incremental parsing. A method to achieve the incrementality is presented for the class of shift-reduce parsers.

Structurally, an ordinary shift-reduce parser is a table-driven algorithm, i.e., a driver runs under direction of tables. The tables are the functions $f$ and $g$, individually constructed for a given grammar. The driver is the same for all shift-reduce parsers. Speaking precisely, our method is indeed to augment the driver to support the incrementality under direction of the original tables.

The philosophy of the incremental parser is to save those moves in parsing a modified input which are guaranteed to be identical to the corresponding moves in parsing the original input. Normally, a modification made in an input only gives rise to a local effect on the syntactic aspect of the input. Basically, it is this locality that makes incremental parsing possible.

As argued at the end of section 2.5, the resulting incremental parser requires about $3(m+n)$ space and $O(m)$ time to parse a modified input, where $n$ is the length of the original input and $m$ is the size of the modification made.
The restriction that $G$ contains no productions with the empty right-hand side can be dropped if we allow the input representation to be amended as described below. Suppose the current configuration is

\[
(\$ X_1 \ X_2 \ \ldots \ X_m, \ b_{i+1}, \ \ldots \ b_n \ \$, \ p_1 \ p_2 \ \ldots \ p_r),
\]

the value of the function $f$ is reduce, that of $g$ is $j$, and production $j$ is $B \rightarrow \epsilon$. Then we insert a circle node and an associated rectangle node between $b_i$ and $b_{i+1}$ as shown in Figure 2.10.

Furthermore, as described in [12], our incremental parser can also be extended to handle the case that $T'$ is syntactically incorrect.

Finally, besides the application in syntax-directed editors, we indicate another potential application of our method in solving the problem of incremental code generation. In [14], Glanville and Graham proposed a method of constructing table-driven code generators whose core algorithm is an LR(1) parser. Hence, it is possible to apply our method to augment such code generators into incremental ones. An investigation in this direction is under way, and the result will be reported later on.
CHAPTER 3
ON CONSTRUCTION OF INCREMENTAL LR(1) PARSERS

3.1 Introduction

Along with the development of syntax-directed editors, the problem of incremental parsing has received much attention recently. A number of methods have been proposed for constructing incremental parsers (see [3, 12, 13, 16, 18, 27, 32, 34]). Among others, however, only the incremental parser given in [16] allows multiple modifications to be made to an already parsed input.

In [32], we described a method of augmenting ordinary shift-reduce parsers into incremental ones. In this chapter, we begin by extending the method to the construction of incremental LR(1) parsers allowing multiple modifications (insertions / deletions / substitutions). We then improve such parsers by using a skipping heuristic which was originally proposed by Wegman [33]. To parse a modified input, the resulting incremental LR(1) parsers require less than \((3m + 6n)\) space and run in time bounded by \(O(m + L^2 \log H)\), where \(m\) is the total size of the modifications, \(L\) is the number of original input symbols which can not be skipped during the incremental parsing, and \(n, H\) are the length and parse tree height of the original input, respectively.

The rest of this chapter is organized as follows: Section 3.2 reviews the definition of LR(1) parsers; section 3.3 describes the
data structure that we use for representing an input and its parsing; section 3.4 presents the method of constructing incremental LR(1) parsers allowing multiple modifications; section 3.5 is subsequently devoted to improving such parsers; section 3.6 analyzes the time and space complexity of the resulting incremental LR(1) parsers; and finally a summary and several remarks are given in section 3.7.

3.2 Review of LR(1) Parsers

Unless otherwise stated, the definitions, conventions and notation of [1] are used throughout the thesis. Particularly, the definition of LR(1) parsers is restated below.

Let \( G = (N, \Sigma, P, S) \) be an augmented LR(1) grammar with \( N \) the set of nonterminals, \( \Sigma \) the set of terminals, \( P \) the set of productions, and \( S \) the start symbol. As usual, the productions in \( P \) are assumed to be numbered from 0 to \( p \), with the augmented starting production being numbered with 0.

Structurally, an LR(1) parser for \( G \) is a table-driven algorithm, i.e., a driver runs under control of a table. The driver is the same for all LR(1) parsers, while one table is constructed for each particular LR(1) grammar. The LR(1) table can be represented as a pair of functions \((f, g)\), where \( f \) is called the action function and \( g \) the goto function. These two functions are defined as follows:

\[
(1) \quad f : \Delta \times (\Sigma \cup \{\$\}) \rightarrow \{\text{error, shift, accept}\} \cup \{\text{reduce-} i | i \text{ is the number of a production in } P \text{ and } 1 \leq i \leq p\}.
\]

\[
(2) \quad g : \Delta \times (\Sigma \cup N) \rightarrow \Delta \cup \{\text{error}\}.
\]
Where $\$ is a new symbol, i.e. endmark, and $\Delta = \{ s_0, s_1, s_2, \ldots, s_r \}$ is a finite set of symbols called states, with $s_0$ particularly distinguished as the initial state.

To parse an input string, the LR(1) parser uses a stack and scans the input from left to right, behaving like a shift-reduce parsing algorithm. During parsing, the stack only contains state symbols with $s_0$ at bottom, and at any moment only one input symbol is under scan, which is called the current symbol. The symbol following the current one is called the 1-lookahead. On the basis of what is the state on top of the stack and what is the 1-lookahead, the function $f$ decides what action to take, either to move the scanner head to the next input symbol or to call a reduction on the scanned input, while the function $g$ decides what state to push onto the stack next. More precisely, the parsing is accomplished by performing the following algorithm.

**ALGORITHM 3.1.** The LR(1) parser.

**INPUT.** The LR(1) table $(f, g)$ for $G$ and an input $\$ w \$, where $w$ is a string to be parsed and $w \in \Gamma^*$. 

**OUTPUT.** If $w \in L(G)$, then the right parse of $w$, which is a sequence of production numbers. Otherwise, an error indication.

**METHOD.** It consists of the following steps:

1. Initially, set the stack only to contain $s_0$ and the current symbol to be the leftmost endmark $\$. 

2. Perform steps (3)-(4) until acceptance or an error turns up. If acceptance occurs, then the string in the output buffer is the right parse of $w$. 

(3) Let \( s \) be the topmost state and \( c \) the current symbol. Apply \( f \) to 
\((s, c')\), where \( c' \) is the 1-lookahead.

(4) Depending on the value of \( f \), execute one of the following steps:

(a) \( f(s, c') = \text{shift} \): Move the scanner head to \( c' \), and apply \( g \) 
to \((s, c')\). Push the resulting state, say \( s' \), onto the stack 
and go to step (3).

(b) \( f(s, c') = \text{reduce-j} \): Let the production \( j \) be \( A \rightarrow \alpha \). Remove 
\(|\alpha| \) states from the top of the stack, and send production 
number \( j \) into the output buffer. Let the state exposed on 
the top of the stack be \( s' \). Apply \( g \) to \((s', A)\). Push the 
resulting state, say \( s'' \), onto the stack and go to step (3).

(c) \( f(s, c') = \text{error} \): Stop the parsing, signaling error.

(d) If \( f(s, c') = \text{accept} \): Terminate the parsing, signaling 
acceptance. The string in the output buffer is declared to 
be the right parse of \( w \).

When an error occurs, the LR(1) parser above simply stops, giving 
an error indication. For LR(1) parsers capable of error recovery, see 
[15, 26].

In the remainder of the thesis, \( G \) always denotes an augmented 
LR(1) grammar, \( s \) the state on top of the stack, \( c \) the current symbol, 
and \( c' \) the 1-lookahead. Further, it is assumed that \( G \) contains no 
productions with the empty right-hand side and the LR(1) parser of \( G \) 
has been optimized so as to eliminate reductions by single 
productions.
3.3 Input Representations

Clearly, in order to perform incremental parsing, it is necessary to retain not only the result of a preceding parse, but also some other useful information generated during parsing. This could require a large amount of storage. Therefore, great care must be taken in finding an informative as well as space-efficient data structure for representing an input and its parse. In what follows we describe the data structure we use for this purpose.

The input is represented by a list of nodes where each node is composed of three fields called SYMB, STATE and LINK, respectively. Each input symbol is assigned a node in which SYMB is set to that symbol. Once a node is assigned to an input symbol, it will never be reassigned. Therefore, we shall refer to a node by the symbol to which it is assigned. In each node, the STATE field contains a state symbol, and the LINK field contains a pointer to the LINK field of another node. Initially, all STATES are set to $\Lambda$, a specially defined value, and all LINKs are set to point to the LINK fields of their preceding nodes, except for the first node in which STATE is set to $s_0$ and LINK is set to $\Lambda$. We call the whole data structure an input representation. For instance, the input $w = S b_1 b_2 \ldots b_n S$ can be represented as shown in Figure 3.1.

In an input representation, the SYMB fields are used to store the input symbols and their contents do not vary during a parsing process. On the other hand, the use of the STATE and LINK fields is twofold. They are used as working storage during a parsing process, and at the end of the process, they contain the result of the parsing as well as
other useful information from which we can acquire, e.g., the stack configuration at any point in the parsing process. However, as we will see later, it will suffice for incremental parsing to have the stack configuration at \textsc{Moment}(b) for any given symbol b, where \textsc{Moment}(b) denotes the moment at which b turns into current.

Our intention is to get the stack configuration at \textsc{Moment}(b) by collecting the STATE fields along the LINK chain starting at the symbol preceding b and ending up with the leftmost endmark $. To this goal, the LR(1) parser given in section 3.2 will be augmented as described below.

To put it briefly, the augmented LR(1) parser is exactly as same as the one described in Algorithm 3.1, except that the following operations need to be added in steps (4.a) and (4.b), respectively:

(4.a) \(f(s, c') = \text{shift}: \) Set STATE of \(c'\) to \(s' = g(s, c')\).

(4.b) \(f(s, c') = \text{reduce-}j: \) Let production \(j\) be \(A \rightarrow \alpha\). In the input representation of \(w\), traverse \(|\alpha|\) STATES along the LINK chain starting at \(c\). Let the node reached be \(b\) and STATE of \(b\) be \(s'\). Set LINK of \(c\) to point to the LINK field of \(b\) and STATE of \(c\) to \(s'' = g(s', A)\).
It is easy to show that when the parsing of \( w \) is finished, the resulting input representation becomes what we want it to be, i.e., for any input symbol \( b \), we can get the stack configuration at \( \text{MOMENT}(b) \) by collecting \( \text{STATES} \) along the \( \text{LINK} \) chain starting at the symbol preceding \( b \) and ending up with the leftmost endmark $.$

3.4 An Incremental LR(1) Parser

In Chapter 1, we presented a method of augmenting ordinary shift-reduce parsers into incremental ones. The resulting parsers only allow one single modification to be made to an already parsed input. In this section, we shall extend the method to the construction of incremental LR(1) parsers allowing multiple modifications such as insertions, deletions, and substitutions.

Let \( w = x_0 y_1 x_1 y_2 x_2 \ldots y_m x_m \) be a sentence generated in \( L(\mathcal{G}) \), and let \( w' = x_0 y'_1 x y'_2 x_2 \ldots y'_m x_m \) be a string obtained from \( w \) by substituting \( y'_i \) for \( y_i \), where \( i = 1, 2, \ldots, m \). Note that \( x_i, y_i, \) and \( y'_i \) all may be empty. In particular, if \( y_i \) is empty but \( y'_i \) not, then it means that \( w \) is modified by the insertion of \( y'_i \). On contrary, if \( y'_i \) is empty but \( y_i \) not, then it means that \( w \) is modified by the deletion of \( y_i \).

Suppose that \( w \) has been parsed by the augmented LR(1) parser described in section 3.3 and we are going to parse \( w' \). Of course, \( w' \) can also be parsed by the same parser. However, as \( w' \) is generated from \( w \) by the substitutions of \( y'_i \) for \( y_i \), it is likely that there are some similarities between the parses of \( w' \) and \( w \). Specifically,
\( x_i, 1 \leq i \leq m, \) may partially or entirely be parsed the same way in \( w' \) and \( w. \) Thus, if we can identify such parts of \( x_i, \) then we will be able to parse \( w' \) in an incremental way. However this is not trivial in general. To solve this problem, we will resort to heuristic methods.

As seen from section 3.2, the action of the LR(1) parser is decided by the functions \( f \) and \( g \) which only depend on the state on top of the stack and 1-lookahead. From this property we can derive the following skipping heuristic(SH-1, for short): Let \( c \) belong to \( x_i, 0 \leq i \leq m; \) if the the stack configuration at \( \text{MOMENT}(c) \) in parsing \( w' \) is the same as in parsing \( w, \) which we will refer to as condition \((*)\) below, then the incremental LR(1) parser that we are looking for can skip from \( c \) to the last symbol of \( x_i, \) denoted by \( \text{LAST}(x_i), \) and continue to parse \( w' \) from there on. By the mechanism of the LR(1) parser, it is clear that the only request for the LR(1) parser to be able to resume parsing from \( \text{LAST}(x_i) \) is the stack configuration at \( \text{MOMENT}(\text{LAST}(x_i)). \) This causes no problems because, as indicated at the end of section 3.3, we can get this stack configuration from the input representation of \( w \) which, as assumed, has already been parsed.

The obstacle in using SH-1 is that testing condition \((*)\) directly could be very laborious. Therefore, we need to find a condition which is equivalent to condition \((*)\), but is more easily testable. To this goal, we introduce a variable \( \text{LEFTBORDER} \) and call a reduction made in parsing \( w' \) matching if after this reduction,

(i) \( \text{STATE} \) of \( c \) in the input representation of \( w' \) is the same as in that of \( w, \) and

(ii) \( \text{LINK} \) of \( c \) in the input representation of \( w' \) is also the same as in that of \( w, \) pointing to the \( \text{LINK} \) field of a node situated to the left of \( \text{LEFTBORDER}. \)
Figure 3.2 Matching reduction

The conditions (i)-(ii) are illustrated in Figure 3.2.

Now we are in a position to present the incremental LR(1) parser constructed by using SH-1.

**Algorithm 3.2.** The incremental LR(1) parser allowing multiple modifications

**Input.** The LR(1) table \((f, g)\) of \(G\) and the input resenations of \(w\) and \(w'\), where \(w = x_0 y_1 x_1 y_2 x_2 \ldots y_m x_m \in L(G)\),

\[ w' = x_0 y'_1 x_1 y'_2 x_2 \ldots y'_m x_m \in \Sigma^* \],

\(w\) has been parsed, and \(w'\) is to be parsed.
OUTPUT. The parsed input representation of \( w' \) if \( w' \) belongs to \( L(G) \).
Otherwise an error indication.

METHOD.

(1) From the input representation of \( w \) get the stack configuration at \( \text{MOMENT}(\text{LAST}(x_0)) \). Set the stack to this configuration.

(2) Set \( \text{LEFTBORDER} \) to the first symbol of \( y'_1 \), denoted by \( \text{FIRST}(y'_1) \), and set \( i = 1 \).

(3) Repeat steps (4) and (6) until acceptance or an error turns up. Go to step (7).

(4) Using the augmented LR(1) parser, parse \( w' \) from \( \text{LAST}(x_{i-1}) \) to \( \text{LAST}(y'_i) \).

(5) Continue to parse \( x_i \). Every time a reduction is made, check the conditions (i)-(ii) above to see if it is a matching reduction.

(6) If it is, then from the next input symbol to \( \text{LAST}(x_i) \) replace the input representation of \( w' \) by that of \( w \). Set \( i = i + 1 \). If \( i = m \), then go to step (7); otherwise set \( \text{LEFTBORDER} \) to \( \text{FIRST}(y'_i) \).

(7) Stop.

The correctness of Algorithm 3.2 is guaranteed by the following theorem.

**Theorem 3.1.** If \( w' \) is in \( L(G) \), then the right parse of \( w' \) can be obtained by performing Algorithm 3.2.

**Proof.** First, since \( G \) is an LR(1) grammar and \( w' \) is obtained from \( w \) by substituting \( y'_1 \) for \( y_1 \), the moves made in parsing \( w' \) must be the same as in parsing \( w \) until \( \text{LAST}(x_0) \). In other words, the incremental parsing of \( w' \) can start at \( \text{LAST}(x_0) \). Hence, step (1) of Algorithm 3.2
is justified. Moreover, it also follows that the input representation of \( w' \) is the same as that of \( w \) from beginning up to \( \text{LAST}(x_0) \).

Next, as seen from steps (3)-(6), from \( \text{LAST}(x_0) \) on, Algorithm 3.2 goes on to parse \( w' \) just by the same mechanism as the augmented LR(1) parser, except that after a matching reduction, it immediately skips to \( \text{LAST}(x_i) \) where \( x_i \) contains the current symbol. Therefore, the point now is to justify such skips. However, as SH-1 says, such skips are legitimate if condition (*) holds. Thus, what remains to be shown is that after a matching reduction, condition (*) always holds true for the skipped symbols of \( x_i \). We shall prove this assertion by induction on the subscript \( i \) of \( x_i \).

**Basis.** It is an immediate consequence of the facts that LEFTBORDER is initially set to \( \text{FIRST}(y_1) \) and the input representation of \( w' \) is as same as that of \( w \) from beginning up to \( \text{LAST}(x_0) \).

**Induction.** Suppose that the induction hypothesis is true for all \( i \)'s < \( k \). Assume that during parsing \( x_k \) a matching reduction is found to be made at \( c \). Then, this must be as shown in Figure 3.3. The stack configuration at that moment consists of \( s \) and all STATES along the LINK chain starting at \( b \) and ending up with the leftmost endmark \( $ \). By the induction hypothesis and the assumption that the reduction is a matching one, this configuration must be the same as the corresponding one encountered in parsing \( w \). That is, condition (*) holds. This completes the induction as well as the proof of Theorem 3.1.

In implementing Algorithm 3.2, it is more space-efficient to use a combined input representation for the common part of \( w \) and \( w' \) while keeping the separate references to the rest.
3.5 An Improvement

We now proceed to improve the incremental LR(1) parser above by using two other skipping heuristics. However, for the reason we shall see below, the first one of them will not be fully pursued. The main topic of this section is the second one which is in essence a generalization of the skipping heuristic proposed by Wegman [9].

Since condition (*) involves the whole stack, the applicability of SH-1 is restricted considerably. As a consequence, the incremental LR(1) parser above may fail to catch some good skipping chances.

Again from the mechanism of the LR(1) parser, we can derive the following second skipping heuristic (SH-2, for short): If the state at top of the stack at MOMENT(c) in parsing $w'$ is the same as that in parsing $w$, which we shall refer to as condition (**) below, then $w'$ should be parsed the same as $w$ from $c$ to $d$, where $d$ is such an input
symbol that (1) d belongs to the same substring \( x_i \) as c; (2) in the
parse tree of w, c and d are the leftmost and rightmost leaves of a
subtree, respectively; and (3) d is the rightmost one among the
symbols satisfying (1) and (2). In other words, the incremental LR(1)
parser above can skip from c to d if condition (*) holds.

As condition (**) only involves the state on top of the stack, the
applicability of SH-2 is certainly less restrictive than that of SH-1.
However, we should note that condition (**) is not on any account a
necessary condition for the skips described above. To see this, e.g.,
let us consider the following program:

\[
\ldots \ldots ; S_1 ; \ S_2 ; \ \ldots \ldots
\]

where \( S_1 \) and \( S_2 \) are two statements. Suppose we modify it as follows:

\[
\ldots \ldots ; \text{if } i > 0 \text{ then } S_1 \ \text{else } S_2 ; \ \ldots \ldots
\]

Then, in parsing the modified program, \( S_1 \) and \( S_2 \) should be parsed the
same as before the modifications, while condition (**) obviously does
not hold for their first symbols. Just for this reason, we leave SH-2
here. For further discussion of using SH-2 to construct incremental
LR(1) parsers, the reader is referred to [16].

It is natural to ask what is a sufficient and necessary condition
for the skip described above. Lemma 3.1 below provides an answer to
this question. Before presenting it, we introduce a definition and a
few notations. Let \( t \) be a subtree in the parse tree of w. We use
\( \text{ROOT}(t) \), \( \text{FRONTIER}(t) \), and \( \text{FOLLOW}(t) \) to denote the nonterminal labelling
the root of \( t \), the frontier of \( t \), and the input symbol following
\( \text{FRONTIER}(t) \), respectively. Furthermore, let \( b \) be the leftmost leaf of
\( t \), we call \( t \) an incident tree of \( b \). Note that an input symbol may have
no incident tree or more than one incident trees.
Lemma 3.1 Assume that $t$ is an incident tree of $c$ and $\text{FRONTIER}(t)$ is completely contained in $x_1, 1 \leq i \leq m$. Then $\text{FRONTIER}(t)$ is retained to be parsed as $t$ in the parse of $w'$ if and only if the following condition (*** holds):

$$f(g(s, \text{ROOT}(t)), \text{FOLLOW}(t)) \neq \text{error}.$$ 

Proof. The argument is similar to that of Lemma 1 in [9].

ONLY-IF. If $\text{FRONTIER}(t)$ is parsed as $t$ in the parse of $w'$, then after $\text{FRONTIER}(t)$ is reduced to $t$, the state on top is $g(s, \text{ROOT}(t))$. Hence, the claim follows from the construction of the LR(1) table $(f, g)$.

IF. If condition (*** holds, then from beginning up to $\text{FOLLOW}(t)$ the portion of $w'$ constitutes a viable prefix of $G$. In other words, this substring, followed by some string which begins with $\text{FOLLOW}(t)$, forms a sentence of $L(G)$. Therefore, owing to the unambiguity of $G$, in the parse of $w'$ $\text{FRONTIER}(t)$ must be parsed as $t$ if $w' \in L(G)$.

By Lemma 3.1, we can see that the substring consisting of the symbols from $c$ to $d$ described above is indeed $\text{FRONTIER}(t)$ of such a $t$ that (1) $t$ is an incident tree of $c$, (2) $\text{FRONTIER}(t)$ does not stretch beyond $\text{FIRST}(y_{i+1})$, and (3) $t$ satisfies condition (***). We call this $t$ the biggest incident tree of $c$, denoted by $\text{BINTREE}(c)$. From Lemma 3.1, we derive the following third skipping heuristic (SH-3, for short): if condition (*** holds, then $\text{FRONTIER}(\text{BINTREE}(c))$ can be skipped.

In order to use SH-3, we need a table called $\text{INCIDENTTREES}$ as a means of finding $\text{BINTREE}(b)$ for any input symbol $b$. In detail, for input symbol $b$, the table contains an entry $\text{INCIDENTTREES}(b)$ which is a pointer to a list. The list has as many nodes as the incident trees of $b$. Each node is composed of two fields, containing the $\text{ROOT}$ and
FOLLOW values of an incident tree of b, respectively. INCIDENTTREES can be constructed along with the parsing process of w. For any given symbol b we can find BINTREE(b) by a binary search in INCIDENTTREES(b) within time less than $\log_2 H$, where H is the parse tree height of w.

We now improve the incremental parser described in section 3.4 by incorporating SH-3 as follows. The only change in Algorithm 3.2 is to replace step (5) by the following step (5'):

(5') Continue to parse $x_i$ by using the augmented LR(1) parser. When executing step (3) of Algorithm 3.2, instead of applying function f to $(s, c')$, first search for BINTREE(c). Skip FRONTIER(BINTREE(c)) if BINTREE(c) is found. Otherwise perform step (3) as usual. Every time a reduction is made, check conditions (i) and (ii) to see if it is a matching reduction.

Skipping FRONTIER(BINTREE(c)) for each symbol $c$ of $x_i$, $1 \leq i \leq m$, remarkably enables the improved incremental LR(1) parser to catch all potential skipping chances in parsing $w'$. In other words, the improved parser can indeed utilize the parse of w to a maximum.

Finally, we must indicate that despite FRONTIER(BINTREE(c)) can be skipped in parsing $w'$, the STATE fields in this portion of the input representation of $w'$ may contain values different from those contained in the corresponding STATE fields in that of w. Therefore, they must be updated accordingly so as to enforce the data consistency. The only reason for such updating is that later on modifications may be made to $w'$. However, it can be shown (see Appendix 1) that such updating can be saved if G is a simple LR(1) grammar.
3.6 Complexity

In this section, we analyze the space and time complexity of the improved incremental LR(1) parser. Noticing the remark at the end of section 3.4, we can see that \(3(m' + n)\) space are required for storing the input representations of \(w\) and \(w'\), on the assumption that ordinary LR(1) parsers use \(n\) space to store \(w\), where \(n = |w|\) and \(m' = \sum_{i=1}^{m} |y_i'|\).

To estimate the space required by the table INCIDENTTREE, we first prove the following lemma:

**Lemma 3.2** If a tree has no nodes of degree 1, then the number of its terminal nodes is greater than that of nonterminal ones.

**Proof.** As seen from [8], it holds that

\[n_0 = 1 + n_2 + 2n_3 + \ldots + qn_q,\]

where \(n_i\) is the number of nodes of degree \(i\), \(0 < i < q\), and \(q\) is the highest degree of the nodes. Since the tree has no nodes of degree 1, \(n_1 = 0\). Hence, the claim follows from the inequality below:

\[n_0 = 1 + n_2 + 2n_3 + \ldots + qn_q > n_1 + n_2 + n_3 + \ldots + n_q.\]

We now give an upper bound on the amount of space required by the table INCIDENTTREES.

**Lemma 3.3** The space required by INCIDENTTREES is less than \(3n\).

**Proof.** First, for each input symbol \(b\), INCIDENTTREES contains a pointer to the list INCIDENTTREES\((b)\). Next, if INCIDENTTREES\((b)\) is not an empty list, then it has as many nodes as the incident trees of \(b\). Recall that the LR(1) parser of \(G\) is assumed to be optimized so as no reduction is made using a single production. Consequently, there is an
one-to-one correspondence between the set of all incident trees and
the set of nonterminals in the parse tree of \( w \). Moreover, the parse
tree of \( w \) must have no nodes of degree 1. Therefore, by Lemma 3.2, the
total number of incident trees is less than \( n \), the length of \( w \). Hence,
the claim follows from fact that each node of INCIDENTTREES(b) has two
fields.

Combining Lemma 3.3 with the remark at the beginning of this
section, we obtain the following theorem:

**Theorem 3.2** The space required by our incremental LR(1) parser
is less than \( 3m' + 6n \).

As to the time complexity of our incremental LR(1) parser, there
are two factors to be taken into account. First, as seen from
Algorithm 3.2, all \( y'_i \) must be parsed if they are not empty. This can
be accomplished in time proportional to \( m' \), the total size of \( y'_i \),
\( 1 \leq i \leq m \). Second, during parsing \( x_i \), for each symbol \( b \) the time
required by the search for BINTREE(b) is certainly bounded by
\( O(\log_2 H) \), where \( H \) is the parse tree height of \( w \). Therefore, the total
time spent in parsing \( x_i \) is bounded by \( O(L \cdot \log_2 H) \), where \( L \) is the
number of symbols which can not skipped during incremental parsing.

Put them together, these two items add up to an upper bound on
the time complexity of our incremental parser, as stated in the the
following theorem:

**Theorem 3.3** The time complexity of our incremental LR(1) parser
is bounded by \( O(m' + L \cdot \log_2 H) \).
3.7 Conclusion

In this chapter, we have discussed three skipping heuristics and used them in the construction of incremental LR(1) parsers which allow any number and form of modifications to be made to an already parsed input. To parse a modified input, such incremental parsers require less than \((3m + 6n)\) space and run in time bounded by \(O(m + L\times\log_2 H)\), where \(m\) is the total size of the modifications, \(L\) is the number of original input symbols which can not be skipped during incremental parsing, and \(n, H\) are the length and parse tree height of the original input, respectively.

To be precise, our incremental parsers are actually generated by only augmenting the ordinary LR(1) driver. Consequently, our approach can work together with any LR(1) parser generator. Moreover, as seen from the description above, there should be no difficulties on our part to apply this approach to the construction of incremental LR(k) parsers with \(k > 1\).

By the same adaption as described at the end of [32], our method can be extended to admit LR(1) grammars which contain productions with the empty right-hand side.

So far the study of incremental parsers is mostly connected with the development of syntax-directed editors for which it is nearly a common practice to store the input in a tree structure, e.g., in an attributed parse tree. Note that because of the LINK fields, our input representation is not a linear list like it looks at the first glance. In fact, our input representation can be transformed into a tree-like
structure while keeping all links. Needless to say, the principle of our incremental LR(1) parsers remains fully workable for this new representation of the input.

From the time complexity formula above, we can see that as far as the running time is concerned our incremental parsers behave favorably in comparison with ordinary parsers, especially when scattered minor modifications are made to an original input, such as, e.g., the insertion of begin end, the addition of an else clause to an if then statement, etc. However, on the other hand, the space complexity formula above shows that our incremental parsers require at least five times more storage than as required by ordinary parsers. This is a substantial increase. Therefore, the use of our incremental parsers is justified only when such trade-off between time and space is considered profitable. Given the present decreasing cost of memory and the emphasis on interactive program development, the trend is definitely towards to reducing time at the expense of space.

Finally, we would like to point out that in the time complexity formula, $H$ is a quite loose upper bound and $L$ is a variable dependent on $G$ and $w$. Since modularity and locality are two of the dominant considerations in designing programming languages and writing programs, it is unlikely that a minor modification to a program syntactically spreads its influence so widely that there is inherently no bound on how far the influence can go. In other words, there might exist a language-dependent and stylistic constant $C$ which bounds such influence. If so, then $L$ would be the product of $C$ and the number of modifications made. About this conjecture, further work remains to be done, both theoretically and practically.
CHAPTER 4

INCREMENTAL EVALUATION FOR ORDERED ATTRIBUTED GRAMMARS

4.1. Introduction

Recently, the notion of incremental attribute evaluators was raised and their construction was studied as an approach to incremental semantic analysis of modified programs (see [6, 29]). There have been a few incremental attribute evaluators running or coming, which are common in that the evaluation order of attributes is determined at run time.

In this chapter, we first present a method of augmenting a conventional evaluator for an ordered attributed grammar (OAG, for short) into an incremental one, then suggest three ways to improve it. The resulting incremental attribute evaluator is statically deterministic, viz., the evaluation order is determined at construction time, and evaluates a modified semantic tree in time proportional to the amount of attribute instances affected by the modification.

Although the method is described here only for OAG evaluators, it can be readily extended to tree-walker evaluators constructed by the algorithm given in [5] for any non-circular attributed grammars (AG's, for short).

The rest of this chapter is organized as follows: Section 4.2 contains an informal introduction to OAGs and their evaluators; section 4.3 describes an example of OAG, borrowed from [19]; section 4.4
formulates the problem of incremental attribute evaluation in its form; section 4.5 presents our incremental evaluator generated by augmenting a conventional OAG evaluator; section 4.6 proves the validity of the incremental evaluator; in section 4.7 three ways are suggested to improve the evaluator; section 4.8 briefly reviews previous work; and finally, some remarks are given in section 4.9.

4.2 Preliminaries

The reader is assumed to be familiar with AGs, e.g., such as described in [22, or 28]. Below is a brief introduction to OAGs and their evaluators; see [19] and others for a complete description.

An AG is a context-free grammar extended by attaching to each symbol of the grammar a finite set of attributes. Associated with each production of the grammar is a finite set of semantic functions defining values of the attributes occurring in the production. There are two kinds of attributes, viz. inherited and synthesized. The start and terminal symbols are assumed to have only synthesized attributes.

Within a production $p$, for each occurrence of a symbol and each attribute of the symbol, there is correspondingly an attribute occurrence. For instance, let $X$ be a symbol occurring in $p$, and $a$ an attribute of $X$, then we use $X.a$ to denote the attribute occurrence. If $X$ occurs more than once in $p$, then we use $X_1.a$ to refer to the attribute occurrence corresponding to the first occurrence of $X$, and $X_2.a$ to that to the second, and so on. For each synthesized attribute
occurrence of the left-hand side (LHS, for short) symbol, there is exactly one semantic function defining its value, and the arguments, if any, are either inherited from the LHS symbol, or synthesized from some of the right-hand side (RHS, for short) symbols; likewise for each inherited occurrence of an RHS symbol. If \( X.a \) is used in defining \( Y.b \), then \( Y.b \) is said to depend on \( X.a \), denoted by \( X.a \rightarrow Y.b \). \( Y.b \) is also called a dependant of \( X.a \), while \( X.a \) is called a donator of \( Y.b \). The set of all dependants of \( X.a \) in \( p \) is denoted by \( \text{DEPENDANTS}(X.a,p) \).

Let \( G \) be an AG, \( w \) a sentence in \( L(G) \). Suppose \( w \) has been parsed yielding a parse tree \( T \). A semantic tree of \( w \) is generated by attaching all attributes to their associated symbols that label the nodes of \( T \). Each attached attribute is said to have an instance on the semantic tree. For simplicity, we shall use the same notations to denote an attribute occurrence and its corresponding instance as well as a parse tree and its corresponding semantic tree; there would be no confusion if we read notations in the contexts in which they occur.

From \( T \) we can construct such a directed graph that the vertices are all attribute instances on \( T \) and the edges are all pairs \((X.a, Y.b)\) where \( X.a \rightarrow Y.b \). The graph is called the dependency graph of \( T \). If there is a circuit in the graph, then \( T \) is said to be circular. If for every sentence in \( L(G) \) there is a non-circular semantic tree, then \( G \) is said to be non-circular or well-defined.

The term "attribute evaluation" refers to a process of assigning values to all attribute instances on a semantic tree in accordance with their defining semantic functions. Algorithms accomplishing attribute evaluation are called attribute evaluators, or simply evaluators. An attribute instance is ready for evaluation when and
only when its defining function is a constant or its donators all have been evaluated. Therefore, for an evaluator to evaluate a semantic tree it is essential to find a proper evaluation order.

Informally, an AG is said to be orderly arrangeable if for each symbol X of the grammar a partial order among the attributes of X can be set up in such a way that the attribute instances attached to a node labelled with X on a semantic tree are always evaluated according to such an order, no matter where on the tree, nor on which semantic tree the node is situated. An OAG is such an AG that its orderly arrangeability can be decided in time polynomial to the size of the grammar. OAGs have been shown to be quite well-suited for specifying the static semantics of practical programming languages in a series of experiments (see [19]).

For each OAG, a tree-walker evaluator can be mechanically constructed by the algorithm given in [19]. Let us call it the OAG evaluator. With each evaluation process the OAG evaluator starts at the root, then walks around on the tree; finally, it returns to the root and terminates there. When it comes to a node, which is also called visiting the node, it evaluates all attribute instances that occur in the production applied at the node as well as are ready for evaluation, and then leaves for the father or a son of the node. A node may be visited several times and the number of visits may vary from node to node. When the OAG evaluator eventually terminates at the root, the semantic tree is completely evaluated and the evaluation process is finished.

Structurally, an OAG evaluator is a table-driven algorithm, i.e., a driver runs under control of a table. The driver is the same for all
OAG evaluators, while one table is constructed for each OAG in particular. In detail, let G be an OAG, then for each production p of G, a so-called visit sequence VS(p) is constructed. VS(p) consists of two kinds of items: X.a and v(k,i), where X.a is an attribute occurrence appearing in p. Suppose p is applied at a node N on a semantics tree, in which case p is also called the production indicator of N. Then, X.a in VS(p) means evaluating the attribute instance X.a, while v(k, i) paying the k-th visit to the father or to the i-th son of N, depending on i = 0 or i > 0.

If there is a total of m items of the form v(k, 0) in VS(p), where k = 1, 2, ..., m, then VS(p) is further divided into m segments, each ended with a v(k, 0). We will use VS(p, k) to denote the k-th segment of VS(p). For all productions of G, all segments are collected into one table. A function called MAPDOWN is devised which maps the visit number k of a son visit v(k, i) and the production indicator p of the son to be visited into the index d of the first table entry of VS(p, k), i.e.,

\[ d = \text{MAPDOWN}(k, p). \]

The action of the OAG evaluator is directed by the table above. At any moment in an evaluation process there is exactly one table entry, which is called the current entry, directing the OAG evaluator. The visit sequence to which the current entry belongs is called the current one and associated with the production applied at the node being visited. If the current entry is an item of the form X.a, then the driver calls the semantic function defining X.a, and the next item in the current visit sequence will be current next; if the current entry is of the form v(k, i), then the driver leaves for the father or
the ith son, depending on whether \( i = 0 \) or \( i > 0 \), and the current visit sequence will be changed accordingly. When a node is revisited, its associated visit sequence resumes control from the item next to where it was left during the last visit. In order to keep track of such control flow, two parameters must be saved. One is the reference to the node being visited, and the other is the index of the table entry that will be current next. A stack is used to maintain such parameter pairs throughout an evaluation process. Figure 4.1 is the main loop of the OAG driver, written in a PASCAL-like language.

\[
\begin{align*}
\text{begin} & \\
\text{push} & \{\text{root, MAPDOWN (1, root.prod_indicator)}\}; \\
\text{repeat} & \\
\text{case} & \text{stack_top.table_entry of} \\
X.a : & \text{call semantic function defining } X.a; \\
& \text{increment( stack_top.table_entry );} \\
v(k,i) : & /* i > 0 */ \\
& \text{increment( stack_top.table_entry );} \\
& \text{push( stack_top.node_ref, MAPDOWN(k,} \\
& \text{stack_top.node_ref.prod_indicator));} \\
v(k, 0) : & \text{pop;} \\
\text{esac} & \\
\text{until stack_is_empty ;} \\
\text{end}
\end{align*}
\]

Figure 4.1 The OAG driver

Finally, if an evaluation process is coordinated on the time axis, then for each node \( A \) there is a unique moment, viz. that of first visiting \( A \), denoted by \( \text{MOMENT}(1, A) \). The stack configuration at \( \text{MOMENT}(1, A) \) is computable and the computation procedure will be described in Appendix 2.
4.3 An Example of OAG

We will use the example given in [19] to illustrate the notion of OAGs and their evaluators.

The example is a simple language which covers four of the most important context sensitive properties of languages: scope rules, mode checking, coercion, and operator identification.

The underlying context-free grammar is $G_0 = (N, I, <program>, P)$, where $N$ is the set of nonterminal symbols, $I$ the set of terminal symbols, and $P$ the set of productions. Specifically,

$$N = \{ <program>, <primary>, <declaration>, <assignment>, <expression> \},$$

$$I = \{ <identifier>, <intconstant>, <realconstant>, +, =, (, ), ;, \text{NEW} \},$$

and $P$ consists of the following productions:

1. $<program> \rightarrow <primary>$
2. $<primary> \rightarrow ( <declaration> ; <assignment> )$
3. $<primary> \rightarrow <identifier>$
4. $<primary> \rightarrow <intconstant>$
5. $<primary> \rightarrow <realconstant>$
6. $<assignment> \rightarrow <identifier> := <expression>$
7. $<expression> \rightarrow <expression> + <primary>$
8. $<expression> \rightarrow <primary>$
9. $<declaration> \rightarrow \text{NEW} <identifier> := <expression>$

Below are the attributes used in $G_0$ (see [19] for their meanings):

{description, access, primode, postmode, evaluable, value, identifier}
Among them, access and postmode are inherited, and the rest are synthesized.

For simplicity, all these attributes will be abbreviated to their first two letters. Association between symbols and attributes is as follows:

\[
\begin{align*}
X.ac & \text{ for } X \text{ in } \{\langle\text{expression}\rangle, \langle\text{primary}\rangle, \langle\text{assignment}\rangle, \langle\text{declaration}\rangle\} \\
X.po & \text{ for } X \text{ in } \{\langle\text{expression}\rangle, \langle\text{primary}\rangle, \langle\text{assignment}\rangle\}, \\
X.pr & \text{ for } X \text{ in } \{\langle\text{expression}\rangle, \langle\text{primary}\rangle, \langle\text{assignment}\rangle\}, \\
X.ev & \text{ for } X \text{ in } \{\langle\text{expression}\rangle, \langle\text{primary}\rangle\}, \\
X.va & \text{ for } X \text{ in } \{\langle\text{expression}\rangle, \langle\text{primary}\rangle, \langle\text{intconstant}\rangle, \\
& \quad \langle\text{realconstant}\rangle\}, \\
& \langle\text{declaration}\rangle.de, \\
\text{and} \\
& \langle\text{identifier}\rangle.id.
\end{align*}
\]

The list of semantic functions associated with each production is omitted here; the interested reader is referred to [19].

Using the algorithm given in [19], we can verify that \( G_0 \) is an OAG and obtain the following visit sequences:

\[
\begin{align*}
\text{VS}(1) &= \langle\text{primary}\rangle.ac, v(1, \langle\text{primary}\rangle), \langle\text{primary}\rangle.po, \\
& \hspace{1cm} v(2, \langle\text{primary}\rangle), v(1, 0). \\
\text{VS}(2) &= \langle\text{declaration}\rangle.ac, v(1, \langle\text{declaration}\rangle), \langle\text{assignment}\rangle.ac, \\
& \hspace{1cm} v(1, \langle\text{assignment}\rangle), \langle\text{primary}\rangle.pr, v(1, 0), \langle\text{primary}\rangle.ev, \\
& \hspace{2cm} \langle\text{primary}\rangle.va, \langle\text{assignment}\rangle.po, v(2, \langle\text{assignment}\rangle), v(2, 0). \\
\text{VS}(3) &= \langle\text{primary}\rangle.pr, v(1, 0), \langle\text{primary}\rangle.ev, \langle\text{primary}\rangle.va, \\
& \hspace{1cm} v(2, 0). \\
\text{VS}(4) &= \text{VS}(5) \\
& = \langle\text{primary}\rangle.pr, v(1, 0), \langle\text{primary}\rangle.ev, \langle\text{primary}\rangle.va, \\
& \hspace{1cm} v(2, 0).
\end{align*}
\]
\[ VS(6) = \langle expression \rangle . ac, v(1, \langle expression \rangle ), \langle assignment \rangle . pr, \\
 v(1, 0), \langle expression \rangle . po, v(2, \langle expression \rangle ), v(2, 0). \]

\[ VS(7) = \langle expression-2 \rangle . ac, v(1, \langle expression-2 \rangle ), \langle primary \rangle . ac, \\
 v(1, \langle primary \rangle ), \langle expression-1 \rangle . pr, \langle expression-2 \rangle . po, \\
 v(2, \langle expression-2 \rangle ), \langle primary \rangle . po, v(2, \langle primary \rangle ), v(1, 0), \\
 \langle expression-1 \rangle . ev, \langle expression-1 \rangle . va, v(2, 0). \]

\[ VS(8) = \langle primary \rangle . ac, v(1, \langle primary \rangle ), \langle expression \rangle . pr, v(1, 0), \\
 \langle primary \rangle . po, v(2, \langle primary \rangle ), \langle expression \rangle . ev, \\
 \langle expression \rangle . va, v(2, 0). \]

\[ VS(9) = \langle expression \rangle . ac, v(1, \langle expression \rangle ), \langle expression \rangle . po, \\
 v(2, \langle expression \rangle ), \langle declaration \rangle . de, v(1, 0). \]

Figure 4.2 shows the semantic tree for \( (\text{NEW \ } x := 1; \ x := x + 2) \), which is a sentence in \( L(G_0) \). In Figure 4.2, the solid lines represent the context-free derivations, while the dotted lines indicate the route taken by the OAG evaluator of \( G_0 \) for evaluating the tree. The reader is invited to verify this evaluation order by manually executing the visit sequences associated with the nodes. Later on, we shall return to this example to illustrate our algorithm for incremental attribute evaluation.

4.4 Formulation of the Problem

In this section, we formulate the problem of incremental attribute evaluation.

Let \( G \) be an OAG. We assume that the underlying context-free grammar of \( G \) has an incremental parser such as, e.g., the one described in the preceding chapters.
Figure 4.2 The semantic tree for \( \text{NEW } x := 1; x := x + 2 \)

Now let \( w = xzy \) and \( w' = xz'y \) be two sentences in \( L(G) \). Suppose \( w \) and \( w' \) have been parsed yielding the parse trees \( T \) and \( T' \), respectively. As said in [34], \( T \) and \( T' \) must have the structures shown in Figure 4.3, where \( x = x_0 x_1 \), \( y = y_1 y_0 \), and the shaded parts are the same in both \( T \) and \( T' \), viz. \( T' \) can be obtained from \( T \) by replacement of the subtree rooted at the node labelled with symbol \( A \).
Note that $x_0$, $x_1$, $y_0$, and $y_1$ all may be the empty string, and particularly, if both $x_0$ and $y_0$ are empty, then the node $A$ becomes the root $S$ in both $T$ and $T'$.

Attaching all attributes to their associated symbols on $T$ and $T'$ we obtain the semantic trees of $w$ and $w'$. For simplicity, we still use $T$ and $T'$ to denote these two semantics trees and refer to nodes by their labelling symbols. Besides, the set of all attribute instances on $T'$ is denoted by $\text{ATTR}$.

Suppose that (1) $C$ is a node which appears on $T$ as well as on $T'$, and (2) the production applied at $C$ on $T$ is the same as on $T'$, if $C$ is not a leaf. Then, each attribute instance of $C$ is said to be retained after the subtree replacement above was made. By this definition, we divide $\text{ATTR}$ into two subsets, one consisting of all those retained and the other of the rest. We denote them by $\text{RETAIN}$ and $\text{NEWBORN}$, respectively. Clearly, the attribute instances in $\text{NEWBORN}$ all belong to the subtree rooted at $A$ and are brought into existence exclusively due to this subtree replacement. Furthermore, in evaluating $T'$ only
attribute instances in RETAIN may have the same values as in evaluating T. Let us denote the subset of all such attribute instances in RETAIN by EQUAL, and the rest by NOTEQU. Needless to say, the subtree replacement above is also the sole source that causes the attribute instances in NOTEQU to have the different values in evaluating T and T'. Just for this reason we call the union of NEWBORN and NOTEQU the set of attribute instances affected by the subtree replacement, denoted by AFFECT. Summing up, we have the following relations:

\[
\text{ATTR} = \text{RETAIN} \cup \text{NEWBORN},
\]

\[
\text{RETAIN} = \text{EQUAL} \cup \text{NOTEQU},
\]

\[
\text{AFFECT} = \text{NEWBORN} \cup \text{NOTEQU}, \text{ and}
\]

\[
\text{ATTR} = \text{EQUAL} \cup \text{AFFECT}.
\]

Now suppose that T has been evaluated and consider evaluation of T'. Of course, T' can be evaluated by running the OAG evaluator of G over T' from beginning to end; however, to the same goal, an incremental approach appears to be more time-efficient because there are two potential advantages available: (1) T has been evaluated, and (2) the attribute instances in EQUAL can retain their values obtained in evaluating T and therefore need not be re-evaluated. In essence, for the sake of evaluation of T', it suffices to evaluate the attribute instances in AFFECT only, and the size of AFFECT is usually by far smaller than that of ATTR. However, owing to our ignorance of the membership in NOTEQU, evaluating more than the attribute instances in AFFECT is unavoidable. In fact, there is no way of knowing a priori the membership in NOTEQU, though we can learn that in RETAIN from parsing. Thus we have to resort to heuristic methods. Precisely, we
will search for the members of NOTEQU within RETAIN in a trial-and-error manner, which inevitably results in evaluating someones in EQUVAL.

Considering the discussion above we formulate the problem of incremental attribute evaluation as follows: Given two semantic trees T and T' such that (1) T has been evaluated and (2) T' can be obtained from T by replacement of the subtree rooted at a node A, without evaluating the whole set ATTR, how should we accomplish evaluation of AFFECT while re-evaluating members in EQUAL as few times as possible.

4.5 Description of the Algorithm

In this section, we present our method of constructing incremental attribute evaluators. The incremental evaluator generated by the following algorithm is a conventional OAG evaluator augmented with a marking procedure.

**ALGORITHM 4.1** The incremental evaluator.

**INPUT.** Two semantic trees T and T' such as described in (1) and (2) at the end of section 4.4.

**OUTPUT.** The evaluated T'.

**METHOD.** The conventional OAG driver is augmented as follows:

(1) Assign to each attribute instance in RETAIN its value obtained from evaluation of T, and to each in NEWBORN a special value $\Lambda$, meaning undefined;

(2) by using the procedure described in Appendix 2, compute the
stack configuration at MOMENT(1, A) in a conventional evaluation process of T', then set the stack to this configuration;

(3) mark every attribute instance in NEWBORN;

(4) re-code the first case of the case statement in Figure 4.1 as shown in Figure 4.4.

begin
...... /* initialisation as described in (1), (2), */
/* and (3) above */
repeat
  case
    stack_top.table_entry of
      X.a : increment(stack_top.table_entry);
        if X.a_is_marked then
          begin
            call the semantic function defining X.a;
            if not result_equal_old_value then
              mark all members of DEPENDANTS(X.a, stack_top.node_ref.prod_indicator);
          end;
      v(k, i) : /* i > 0 */
        increment(stack_top.table_entry);
        push(stack_top.node_ref, MAPDOWN(k, stack_top.node_ref.prod_indicator));
      v(k, 0) : pop;
  esac
  until stack_is_empty;
end

Figure 4.4. The incremental OAG driver

In implementing the algorithm above, the common parts of T and T' can be combined into one structure, while keeping their different subtrees separated. This reduces storage use as well as assignment operations a great deal. Moreover, for an attribute instance, the information on its marking can be stored within its corresponding table entry, and also the first case of the case statement in Figure 4.4 is accordingly split into two cases, one dealing with marked attribute instances and the other with unmarked ones. Thus, mark-checking can be saved.
4.6 Correctness of the Algorithm

In this section, we shall prove the validity of the algorithm above.

**Lemma 4.1** Every attribute instance in NOTEQU must directly or indirectly depend on some members of NEWBORN.

**Proof.** Let X.a be an attribute instance in NOTEQU. The function defining X.a must not be a constant. If there is a member of NEWBORN among the donators of X.a, then the claim is proved, otherwise there must be a donator of X.a, say, Y.b which is a member of NOTEQU. Apply the same reasoning with Y.b, and so on. As G is non-circular, such a deductive process must end up with an attribute instance, say, Z.c such that a donator of Z.c is a member of NEWBORN, because otherwise Z.c would be in EQUVAL, which contradicts our selection of Z.c.

**Lemma 4.2** Let BEFORE(1, A) denote the set of all attribute instances evaluated before MOMENT(1, A) in a conventional evaluation process of T'. Then,

BEFORE(1, A) ⊆ EQUVAL.

**Proof.** By definition, BEFORE(1, A) must be contained in RETAIN, because the attribute instances in NEWBORN all belong to the subtree rooted at A. Furthermore, recall that RETAIN = EQUVAL U NOTEQU, and by lemma 4.1 it must hold that BEFORE(1, A) ⊆ EQUVAL, because the attribute instances in BEFORE(1, A) by no means depend on anyone in NEWBORN.

Lemma 4.2 justifies the skip of evaluating the members of BEFORE(1, A) which is carried out by step (2) of the algorithm above.
Lemma 4.3. Let MARK denote the set of the marked attribute instances throughout an incremental evaluation process of $T'$. Then,

$$AFFECT \subseteq MARK.$$ 

Proof. First, by step (3), we have NEWBORN $\subseteq$ MARK.

Second, as seen from the first case of the case statement in Figure 4.4, an attribute instance will be marked if and only if it depends on a marked one which i not a member of EQUVAL. By lemma 4.1, we can conclude that a member of NOTEQU will be marked sooner or later, which completes the proof because AFFECT = NEWBORN U NOTEQU.

Lemma 4.3 implies that the attribute instances in AFFECT will be all evaluated when the incremental evaluation process terminates.

Lemma 4.4 For each attribute instance in ATTR, the value assigned by the incremental evaluator accords with its defining function.

Proof. By induction on the evaluation order number of an attribute instance.

Basis. Suppose X.a is the first evaluated attribute instance. As an attribute instance is ready for evaluation if and only if its defining function is a constant or its donators all have been evaluated, X.a must depend on no others in ATTR. Therefore, no matter if X.a is in RETAIN or in NEWBORN, the value assigned by the incremental evaluator to X.a always accords with its defining function.

Induction. Assume that the claim holds for all attribute instances with the evaluation order numbers < n. Let X.a be the nth evaluated attribute instance. There are two cases: X.a is either marked or
unmarked. If X.a is unmarked, then, by lemma 4.3, X.a must be in EQUVAL. Noticing that the incremental evaluator never re-evaluates an unmarked attribute instance, we get the claim proved for the unmarked X.a. Now if X.a is marked, then X.a must have been evaluated by the incremental evaluator, as indicated in the first case of the case statement in Fig. 4.4. There are also two subcases: viz. the defining function of X.a is either a constant or not. In the former subcase, the claim to be shown holds spontaneously; in the latter, by the induction hypothesis, the values assigned to all donators of X.a accord with their defining functions, and consequently the value assigned to X.a accords with its defining function too. Thus, we complete the induction as well as the proof.

Combining lemmas 4.2-4.4, we obtain the following

Theorem 4.1 The incremental evaluator generated by the algorithm in section 4.5 evaluates T' correctly.

The following theorem 4.2 is essential for later computing the time complexity of the incremental evaluator.

Theorem 4.2 The size of MARK is proportional to that of AFFECT.

Proof. By step (3), MARK is NEWBORN at the start of the incremental evaluation of T'. Afterwards, as seen from the first case of the case statement in Figure 4.4, MARK is enlarged only when a member of NOTEQU is found and each enlargement is always bound by the maximum number of dependants of a symbol in G. Noticing that AFFECT = NEWBORN U NOTEQU, the theorem follows immediately.
4.7 Three Improvements

In this section, three improvements are in turn suggested to terminate an incremental evaluation process of $T'$ sooner than by running the incremental evaluator described above.

4.7.1 Termination conditions

Generally, an incremental evaluation process of $T'$ terminates when the incremental evaluator finally returns to the root of $T'$. However, a close study indicates that it may terminate sooner than such a moment. In fact, by the definition of AG's, each semantic function is associated with one production and only applied to the attribute instances occurring in the production. Consequently, the only way in which a replacement of the subtree rooted at $A$ may propagate its influence is to expand successively upwards one subtree by another.

Now suppose $A_i$ is a node such that (a) $A_i$ is an ancestor of $A$, (b) the incremental evaluator is going to execute the last item of the visit sequence associated with $A_i$, and (c) all attribute instances of $A_i$ have been found to be in EQUVAL. Then, by the definition of $A_i$, we can see that the attribute instances that are evaluated thereafter must be all in EQUVAL because (1) they must all fall outside the subtree rooted at $A_i$, and (2) the replacement of the subtree rooted at $A$ has no more effect outside the subtree rooted at $A_i$. Therefore, the incremental evaluator need not run any longer. Moreover, as far as the semantics is concerned, only through its synthesized attributes does $A_i$ interface with other parts of $T'$ outside the subtree rooted at $A$, and
therefore condition (c) above can be relaxed as (c') all synthesized attribute instances of \( A_i \) have been found to be in EQUVAL. Thus, the incremental evaluator can terminate when we are going to execute the last item of the visit sequence associated with \( A_i \) satisfying the following two conditions:

1. \( A_i \) is A or an ancestor of A;
2. all synthesized attribute instances of \( A_i \) have been found to be in EQUVAL.

We call (1) and (2) the termination conditions. They should be tested with A and each ancestor of A. In order to detect such testing moments, a special symbol $ is inserted as the second last item in each visit sequence, and a new case dealing with $, as shown below, is accordingly added to the case statement in Figure 4.3:

\[
\begin{align*}
\$ & \: \text{increment(stack_top.table_entry);} \\
& \quad \text{if termination_conditions_hold then goto fin;}
\end{align*}
\]

where fin is a label placed just before the end of the outmost block.

4.7.2 Skip visiting the father

Suppose \( A_i \) is the first node satisfying the termination conditions and the visit sequence associated with \( A_i \) is VS(p). The incremental evaluator will terminate after executing $ in VS(p). However, among the attribute instances that are evaluated before executing that $, there may be some members of EQUVAL. More precisely, VS(p) is generally of the following form:

\[
\text{VS(p) = ........, A_i.a, ........, $, v(m, 0).}
\]
Where \( A_i.a \) is the last synthesized attribute occurrence. Among the items between \( A_i.a \) and $ there may be some visits to the father of \( A_i \) which may further lead to visits to brothers or more ancient ancestors of \( A_i \). Anyway, the attribute instances attached to such nodes definitely fall outside the subtree rooted at \( A_i \) and therefore must belong to EQUVAL. In other words, they need not be re-evaluated. This means that the incremental evaluator can skip all visits to the father of \( A_i \) between \( A_i.a \) and $. In order to achieve such a skip another special symbol & is inserted immediately after the last synthesized attribute occurrence in each visit sequence. When the incremental evaluator executes & the termination conditions are tested. If they hold true, then a flag skip_father is set up, meaning that from now on all visits to the father can be skipped. Naturally, when the incremental evaluator comes to $, the termination conditions need not be tested once again, because the result of testing can be learnt by checking whether the flag skip_father has been set up or not.

**Degression.** This improvement also brings in a new insight into the construction of visit sequences. Indeed, in order to skip visiting the father as described above, it should always be favorable to have synthesized attribute occurrences evaluated as early as possible. In other words, if in a visit sequence there is a father visit \( v(k, 0) \) and a synthesized attribute occurrence \( X.a \) such that \( v(k, 0) \) precedes \( X.a \) but does not contribute to evaluation of \( X.a \), then \( v(k, 0) \) should be swapped with \( X.a \) if possible.
4.7.3 *Skip visiting brothers*

The termination condition (2) says that all synthesized attribute instances of \( A_i \) must be in \( \text{EQUVAL} \). As a matter of fact, it can be a bit more relaxed as described below.

Suppose \( s \) is such a synthesized attribute that if \( s \) is associated with a right-hand side symbol, then \( s \) is only used in defining synthesized attributes of the left-hand symbol. Let us call \( s \) of type \( \text{UP} \). Now suppose \( N \) is an ancestor of \( A \) such that the incremental evaluator has executed & in the visit sequence associated with \( N \) and found that the synthesized attributes violating condition (2) are all of type \( \text{UP} \). Then, by definition, it is easy to see that the attribute instances belonging to the subtree rooted at brothers of \( N \) must all be in \( \text{EQUVAL} \). Therefore, it is superfluous to visit such brothers of \( N \) hereafter. In order to skip visiting them, a flag \( \text{skip\_brothers} \) is introduced. If the node being visited is an \( N \) as described above, then the flag \( \text{skip\_brothers} \) is set up. After resuming execution of the visit sequence associated with the father of \( N \), all visits to brothers of \( N \) are skipped if the flag \( \text{skip\_brothers} \) is set up.

This improvement could be better appreciated if one notices that a good many attributes used in practice are of type \( \text{UP} \).

Before concluding this section, let us revisit the example OAG given in section 4.3. Figure 4.5 shows the semantic tree for the modified program \( (\text{NEW } x := 1; x := x + 0.5 + 2) \) evaluated by our incremental evaluator which has incorporated the improvements above. The attribute instances framed in boldface are those which are actually evaluated in the process of incremental evaluation.
Figure 4.5 An example of incremental attribute evaluation
4.8 Conclusion

In this chapter, we have presented a method of augmenting a conventional OAG evaluator into an incremental one and suggested three ways to improve the time requirement of the resulting incremental evaluator. Noticing the two remarks made at the end of section 4.5 as well as the improvements, it is easy to see that the time complexity of our incremental evaluator can be represented in terms of the size of the set MARK. Hence, by theorem 4.2, we conclude that our incremental evaluator accomplishes evaluation of a modified semantic tree in time proportional to the number of attribute instances affected by the modification.

Furthermore, as nothing more than the static determinacy of OAG evaluators has been utilized in the augmentation, the method can be readily extended to tree-walker evaluators constructed by the algorithm given in [5] for any non-circular AG’s.

The inherent drawback of all incremental evaluators, including ours, is the requirement of a large amount of storage for saving the evaluation result of an original semantic tree. Indeed, in an evaluation process a good many attribute instances are ephemeral and allocated only temporary storage by a conventional evaluator. Considering this fact, incremental evaluators appear to be rather space-expensive. Nevertheless, the cost in space is paid off by the gain in time when using an incremental evaluator in an interactive environment. Very promisingly, along with the increasing interactivity in programming and the decreasing cost of memory, incremental
evaluators will more and more go into use in the fields such as code
generation, data flow analysis, separate compilation, etc.
CHAPTER 5 SUMMARY AND EXTENSIONS

In this chapter we will summarize the results of this research and indicate some directions for further work.

5.1 Summary

This thesis is concerned with incremental syntactic and semantic analysis of modified programs. The body of the thesis consists of Chapters 2, 3 and 4.

In chapter 2, we presented a method of augmenting ordinary shift-reduce parsers into incremental ones. The obtained incremental parsers are more theoretical than practical because they allow only one single modification to be made to an already parsed input. In Chapter 3, we extended the method to the construction of incremental LR(1) parsers which allow multiple modifications such as insertions, deletions, and substitutions. Moreover, by using the generalized skipping heuristic, the improved incremental parsers are actually able to make the maximum use of a preceding parse within time bounded by $O(m + L \times \log_2 H)$, where $m$ is the total size of the modifications, $L$ is the number of original input symbols which can not skipped during incremental parsing, and $H$ is the parse tree height of the original input.

In Chapter 4, as an approach to incremental semantic analysis, we described a method of constructing incremental evaluators for ordered
attributed grammars. The obtained incremental attribute evaluators are statically deterministic and time optimal in the sense that the amount of re-evaluated attribute instances is bounded by $O(|AFFECT|)$, where AFFECT is the set of attribute instances affected by the modification made.

5.2 Suggestions for Further Work

We have suggested several very specific topics for future research in the preceding chapters. At this concluding point, we would like to consider some of them a bit more closely.

First, we take up the topic of incremental code generation. When the work reported in Chapter 2 was finished, an idea immediately came upon us: our method of constructing incremental shift-reduce parsers may be used in solving the problem of incremental code generation, because in the table-driven code generation algorithm [14] the driver is actually an LR(1) parser. However, as code generation mainly deals with the semantic aspect of a programming language, we soon realized that, rather than by tools of incremental syntactic analysis, the problem of incremental code generation should be handled by tools of incremental semantic analysis. This idea was confirmed when we read Ganapathi's thesis [10]. To improve the Glanville-Graham's code generators [14], Ganapathi used attribute grammars to specify the intermediate representation of a source code and left code generation to be accomplished in a process of attribute evaluation. In fact, Ganapathi merely used some very simple synthesized attributes which
can be evaluated in one pass from left to right. Therefore, despite
the fact that the method described in Chapter 4 is completely workable
for attribute grammars evaluable in one left-to-right pass (as pointed
out in [18], such grammars are a subclass of OAGs), we would like to
find a simpler way to construct incremental evaluators for such
attribute grammars which are more efficient in time and space
requirement.

Next, incremental data flow analysis is another field in which the
techniques of incremental attribute evaluation may be also applicable.
As a matter of fact, Farrow [8] has specified attribute grammar models
for various problems of global data flow analysis. Therefore, it seems
promising to work further in this direction.

Further, an extention of the incremental OAG evaluators described
in Chapter 4 is to allow more than one subtree to be replaced in an
already evaluated semantic tree. For this extention, the only change
that needs to be made to Algorithm 4.1 is to initialize NEWBORN by the
attribute instances of all replacing subtrees. Unfortunately, however,
the termination conditions given in section 4.7.1 are much complicated
by the multiple subtree replacement, and so are the other improvements
described in sections 4.7.2 and 4.7.3. Therefore, from the practical
point of view, it remains open if we should make this extention.

Last, we turn to the enhancement of incremental OAG evaluators. As
pointed out in [17], when used in an incremental update environment
found in a syntax-directed editor, attribute grammars have the
following shortcoming: objects that are closely related in a semantic
sense can be arbitrarily far from each other in the syntax tree
describing the program. For instance, in a Pascal program, a variable
can be used at any depth in the block containing its declaration. If
the variable has been semantically modified, then an incremental OAG
evaluator must traverse a long syntactic path in order to accommodate
this modification. This certainly slows down a process of incremental
attribute evaluation. The improvements described in section 4.7 could
alleviate the problem to some extent, but do not solve it completely.
As done in [17], a possible solution is to extend the definition of
attribute grammars in some ways. It is still to be seen how our
incremental OAG evaluators can be adapted to admitting such
extensions.


APPENDIX 1

In this appendix, we shall prove that if the given grammar is a SLR(1) grammar, then after skipping \textsc{frontier}(\textsc{bintree}(c)), the updating described in section 3.5 is superfluous because the \textit{STATE} fields in this portion of the input representation of \( w' \) contain the same values as those in that of \( w \).

For the reader's reference, first we give a brief account of what is SLR(1) grammars and how to construct their SLR(1) tables. Unless otherwise stated, we shall use the terminology and formulation of [2].

Let \( G = (N, \Sigma, P, S) \) be a context-free grammar. As usual, \( S \) is a new symbol, i.e., endmark, and \( V = N \cup \Sigma \). Suppose \( A \in N \) and \( \alpha \in V^* \), we define the following sets:

\[
\text{first}(\alpha) = \{X \in V \mid \alpha \Rightarrow Xw, \text{ where } w \in V^*\},
\]

and

\[
\text{follow}(A) = \{X \in V \cup \{\$\} \mid S \Rightarrow \betaAXY, \text{ where } \beta \in V^* \text{ and } X\gamma \in V^*\{\$\}\}.
\]

Let the \( p \)-th production be denoted by

\[
A_p \rightarrow X_{p1} X_{p2} \ldots X_{pn}, \quad \text{where } n \geq 1.
\]

For each \( p \), we define the set \( T_p \) as follows:

\[
T_p = \text{follow}(A_p) \cap (\Sigma \cup \{\$\})
\]

Furthermore, \( \Delta = \{s_0, s_1, \ldots, s_r\} \) is a set of states, where each state is a set of elements \([p, j]\), and particularly \( s_0 = \{[0, 0]\} \) is called the initial state. Informally, the occurrence of an element \([p, j]\) in a state \( s_k \) means that \( j \) symbols on the right-hand side of production \( p \) have been recognized from the scanned input.

SLR(1) grammars are a subclass of LR(1) grammars for which the
LR(1) table \((f, g)\) can be constructed with less effort than for non-SLR(1) grammars and the table size is also much smaller than that of non-SLR(1) ones. There are different ways to define SLR(1) grammars. We call \(G\) a SLR(1) grammar if the following algorithm successfully terminates with the SLR(1) table for \(G\).

**Algorithm.** The SLR(1) table constructor.

**Input.** A context-free grammar \(G = (N, \Sigma, P, S)\).

**Output.** The SLR(1) table \((f, g)\) for \(G\) if it is a SLR(1) grammar. Otherwise a message that \(G\) is not SLR(1).

**Method.**

1. Set \(s_0 = \{[0, 0]\}\) and \(m = 0\).

2. Compute \(s'_m\) from \(s_m\), where \(s'_m\) is defined recursively as the smallest set satisfying

\[
s'_m = s_m \cup \{[q, 0] \mid \exists \ [p, j] \in s'_m, j < n_p, X_p(j+1) = A_q\}.
\]

3. Compute

\[
Z' = \{X \in V \mid \exists \ [p, j] \in s'_m, j < n_p, X = X_p(j+1)\},
\]

and \(Z_p = T_p, \text{ if } [p, n_p] \in s'_m \)

\[= \emptyset, \text{ otherwise.}\]

4. If \(Z'_p \cap Z'_q = \emptyset\) for all \(p \neq q\), then for each symbol \(X \in V\), a state table entry for the state \(s_m\) is determined as follows:

   - If \(X\) is not in \(Z'_p\) for any \(p\), the entry is blank,
   - If \(X \in Z_p\), the entry is the production number \(p\),
   - If \(X \in Z'_p\), the entry is the state \(s\) to be entered next, where

\[
s = \{[p, j+1] \mid [p, j] \in s'_m, j < n_p, X = X_p(j+1)\}.
\]

5. Otherwise go to step (6), giving the message that \(G\) is not a SLR(1) grammar.
(5) If $s$ is not present in the list of all possible states, then add $s$ into it. If $m$ is not pointing to the last state in this list, then increment $m$ by one and go to step (2).

(6) Stop.

We now proceed to prove our assertion stated at the start of this appendix. Suppose that the current symbol is $c$ and condition (***) holds for $\text{BINTREE}(c)$. By Lemma 3.2 $\text{FRONTIER(BINTREE}(c))$ is retained to be parsed as $\text{BINTREE}(c)$ in the parse of $w'$. Thus, $\text{FRONTIER(BINTREE}(c))$ is reduced to $A = \text{ROOT(BINTREE}(c))$ in the parse of $w'$ as in that of $w$. Let $s'$ and $s$ be the states at the top of the stack at $\text{MOMENT}(c)$ in parsing $w'$ and $w$, respectively. By step (2) of the algorithm above, we can see that both $s'$ and $s$ must contain an element $[p, j]$ such that $A = X_{p}^{j+1}$. Further, by steps (2)-(4), we can deduce that $c$ must be in the set $Z'$ and $c = X_{q}^{1}$ for some $q$. Therefore, after $s'$ the next state to enter in parsing $w'$ is the same as after $s$ in parsing $w$. Moreover, since $G$ is an LR(1) grammar, this should remain true through the last symbol of $\text{FRONTIER(BINTREE}(c))$. Hence, the updating described in section 3.5 is superfluous.
APPENDIX 2

Below we describe for a given node A how to compute the stack configuration at \( \text{MOMENT}(1, A) \) in an evaluation process of \( T' \).

Let \( p_1 \) be the production applied at \( A \), \( \text{VS}(p_1) \) the visit sequence associated with \( p_1 \), and \( A \) the \( i \)th son of \( A_1 \). From the mechanism of the OAG driver it is easy to see that the topmost stack element must be \((A_1, d_1 + 1)\), where \( d_1 \) is the index of the table entry \( v(1, i) \) in \( \text{VS}(p_1) \). Further suppose that \( v(1, i) \) occurs in the \( k \)th segment of \( \text{VS}(p_1) \), \( p_2 \) is the production applied at \( A_2 \), \( \text{VS}(p_2) \) is the visit sequence associated with \( p_2 \), and \( A_2 \) is the \( j \)th son of \( A_2 \). Again from the mechanism of the OAG driver it is easy to see that the second topmost element must be \((A_2, d_2 + 1)\), where \( d_2 \) is the index of the entry \( v(j, k) \) in \( \text{VS}(p_2) \). Continuing this reconstruction until the root of \( T' \) is reached, we can recover all elements in reverse order compared with how they were pushed down into the stack, so long as we have a means of computing \( d_1, d_2, \ldots \).

Two auxiliary functions are devised for computing \( d_1, d_2, \ldots \). One is called \( \text{TRACEUP} \) and maps the visit number \( k \), the son number \( i \), and the production indication \( p \) into the index \( d \) of the entry \( v(k, i) \) in the visit sequence \( \text{VS}(p) \) associated with \( p \). Thus, for instance, we have

\[
d_1 = \text{TRACEUP}(1, i, p_1), \quad d_2 = \text{TRACEUP}(k, j, p_2), \ldots
\]

The other is called \( \text{SEGMENTNO} \) and maps the index \( d \) of an entry \( v \) and the production indicator \( p \) into the segment number \( s \) such that \( v \) occurs in the \( s \)th segment of the visit sequence associated with
production p. Thus, for instance, we have

\[ k = \text{SEGMENTNO}(d_1, p_1). \]

For any node N, we can learn from parsing the son number of N as well as the number of the production applied at N. Therefore, knowing TRACEUP and SEGMENTNO is enough for computing \( d_1, d_2, \ldots \).