A NEW APPROACH FOR ATTRIBUTE EVALUATION AND ERROR CORRECTION IN COMPILERS

Vimal Singh Begwani

Computer Sciences Technical Report #483

August 1982

A New Approach for Attribute Evaluation and Error Correction in Compilers

By

Vimal Singh Begwani

A thesis submitted in partial fulfillment of the

requirement of the degree of

Doctor of Philosophy

(Computer Sciences)

at the

University of Wisconsin - Madison

August 1982

A New Approach for Attribute Evaluation and Error Correction in Compilers

By

Vimal Singh Begwani

Under the supervision of Professor Charles N. Fischer

ABSTRACT

Attributed grammars can be used to specify the context-sensitive syntax of programming languages. We consider a subclass of attributed grammars known as absolutely non-circular attributed grammars. These have been shown to be equivalent to the class of non-circular attributed grammars. A very general on-the-fly attribute evaluation algorithm is presented. The algorithm can be used to parse any context free grammar augmented with an absolutely non-circular grammar.

We then present a context free and context sensitive error correction strategy. We use context past the point of error in choosing a correction. We present a two level error correction algorithm. The algorithm is automatic and table driven, and can deal with any error situation. We discuss the generality of the algorithm and its practicality using our implementation results.

ACKNOWLEDGEMENT

I wish to thank Professor Charles N. Fischer for his constant encouragement and advice throughout my entire research work. His Compiler Construction and Parsing Theory courses were the initial source of inspiration in programming languages. I also want to thank him for being so understanding to my personal problems and helping me with them.

I would like to thank Professors Marvin Solomon and Robert Cook for their dedicated reading of the manuscript. I am also thankful to Professors Donald Dietmeyer and Ed Desautels for serving on the examination committee.

I want to thank my wife Raju and my daughter Nisha for all their help and for being so understanding.

Finally, I want to thank my friends Shailesh Shah, Anil Chaudhary, Ashok Desai, Narendra Desai, Balmukund and his wife Rupal Parikh, Avinash and his wife Nutan Vaidya for making my stay in Madison very pleasant and most enjoyable.

I am grateful to the Computer Sciences Department at the University of Wisconsin - Madison for their financial support.

Table Of Contents

Chapter 1

INTRODUCTION

1.1. Review of the Previous Work

Error correction and recovery have always been an important aspect of compiler design, and accordingly there have been a large number of papers on this subject in recent years [GHJ79, PK80, PD78, FMQ80, MA82, JR82]. Yet researchers are not fully satisfied by these methods. Most of these methods have rather serious drawbacks. Some [GHJ79, PK80, PD78, TA78] of the proposed methods are completely ad hoc, and work well in some situations, but not so well in others. Others, when faced with certain syntax errors are forced to skip ahead, completely ignoring a portion of the input before restarting the parsing. The problem with such methods is that if there are more errors in the skipped input, then those errors will not be detected. Hence several compilations are required to detect and correct all the errors.

Recently, automatic table-driven algorithms have been suggested [DI78, FMD79, FMQ80, MA82] which are guaranteed to work on all the inputs. Yet these methods too seem to have drawbacks. Some of these methods [FMD79, FMQ80, MA82] can deal with context free (or parsing) errors only. Hence they cannot correct any context sensitive error (e.g. type mismatch or wrong qualifier with an identifier). Further, the corrections made by these algorithms can contain context-sensitive errors. Secondly, most of these error correctors work on one symbol lookahead at a time. Hence a correction made by these algorithms is only locally optimal and may not be the best correction. These drawbacks could be rather serious, especially in strongly-typed modern programming languages. Lately, an increasing amount of interest has been expressed in techniques for context-sensitive error corrections [DI78, GHJ79, JR82].

For example, the use of context-sensitive information was investigated by Dion [DI78], but the algorithm proposed is rather expensive (although it is linear in time and space complexity). Graham *et. al.* [GHJ79] have proposed a context-sensitive corrector, which works only in some limited situations (*parsing* errors which can be corrected by a single insertion, deletion, or replacement. That is, the author's algorithm can detect only syntax error and then tries one symbol correction which is semantically correct. Therefore, a construct which is syntactically correct but semantically wrong will not be correct by this algorithm).

One of the most powerful syntactic error correction methods is Aho and Peterson's [AP72], which tries to transform a given input into a legal program using a least number of repairs. It uses error productions to make insertions, deletions, and replacements, and uses Earley's [EA70] parser to select the minimum number of repairs. We will show that we can achieve the same results without including the error productions (which can drastically increase the grammar size). Our technique uses the parser suggested by Graham *et. al.* [GHR80]. Even with these proposed improvements, the Aho and Peterson algorithm will be very expensive to use in practice; but we will demonstrate that this idea of transforming the given input into a legal string using minimum repairs is very useful and we can apply it in some limited fashion (correcting a small region around the error token or tokens), to make fairly good corrections. Mauney [MA82] has proposed one such algorithm, but his algorithm can deal with syntax errors only.

We propose the following error correction strategy. When an error is detected, we first try a single insertion, deletion, or replacement and validate the correction over the next k input symbols (we term it as one-symbol correction with k symbols validation). If we cannot pick a suitable single insertion, deletion, or replacement, then we try to transform the next k input symbols (the value of k is determined using criteria explained later) into a legal string that can follow the parsed left context, using a minimum correction cost. While selecting a correction we use context-sensitive information also. We will use an attribute grammar for representing context-sensitive information. For parsing a few symbols past the error token (for both first level and second level corrector) we will use the parsing method proposed by Graham *et. al.*

In the next chapter we will present a technique for attribute evaluation in the parsing method proposed by Graham *et. al.* (we shall refer to this algorithm as the GHR parser in the rest of this thesis). We will present a fairly general attribute evaluation technique applicable for any non-circular attributed grammar [KN68]. Then in the following chapter we will present the error correction algorithm outlined above. Finally we will show that we can obtain the same result as the Aho and Peterson algorithm if we parse the whole program using our error corrector.

Chapter 2

Attribute Evaluation

2.1. INTRODUCTION

Context-free grammars have been throughly studied by compiler writers [AU72, BC79]. Efficient linear-time algorithms for recognizing many subclasses of context-free grammars are well known [AU72], but these methods are too restrictive in certain applications. There are several general context-free recognizers that can recognize any context-free grammar. It is well known that many semantic aspects of the programming languages cannot be expressed with any context-free grammar. A variety of techniques have been used to introduce the idea of semantics into context-free grammars, such as two-level grammars, affix grammars, and attributed grammars [PA80].

Efficient attribute evaluation algorithms are now known for evaluating any non-circular attributed grammar [JW75, KW76, KR79, LRS74]. Most of these algorithms are off-line, that is, to find the meaning of a string, they first find its parse tree and then determine the values of all the attributes of the symbols in the tree. This approach is not very attractive in those applications in which attributes of symbols can control the shape of the parse tree.

In contrast, on-line attribute evaluators run in parallel with the parser and try to evaluate as many attributes as possible at any given point in the parse. On-line attribute evaluators are well known for certain restricted classes of attribute grammars (L-attributed grammars with LL(k) parsers and S-attributed grammars with LR(k) parsers [LRS74]). Rowland [RO77] extended this work to combine attribute evaluation for a general non-circular attribute grammar with GLC(k) [RO77] parsers. We further extend this work and develop an on-line attribute evaluator for a general context-free grammar.

We will develop an on-line attribute evaluator for the parsing method proposed by Graham *et. al.* [GRH80], which can be used to parse any context-free grammar. Since this method is very similar to other parsing methods for general context-free grammars (e.g. Early's [EA70] and Cocke, Kasami, and Younger's (also known as the CYK parser) [AU72] parsing methods), it should be easy to adapt this evaluator to other related parsing methods.

2.2. Attributed Grammars

In this section we review some basic definitions related to formal grammars, attribute grammars, and parsers.

An *alphabet* or *vocabulary* is a finite set of symbols. A *sentence* over an alphabet is a string of finite length composed of symbols of the alphabet. The empty sentence, λ, is the sentence consisting of no symbol. If Σ is the alphabet, Σ^* denotes the set of all sentences composed of symbols of Σ, including the empty sentence.

An *attributed grammar* is an ordinary context-free grammar augmented with *attributes* and *semantic functions* as described below.

Definition : A *context-free grammar* G is a four tuple G = (V_n, V_t, P, S). Where V_n is a set of non-terminal symbols, V_t is a set of terminal symbols, disjoint from V_n. We write V for $V_n \cup V_t$, called the vocabulary. S is a distinguished symbol of the grammar called the start symbol. P is a set of production rules. A production $p \in P$ is written as

$$p\colon X_0 \Longrightarrow X_1 \dots X_{n_p}, \text{ where } n_p \geq 0,\ X_0 \in V_n, \text{ and}$$
$$X_k \in V \text{ for } k>0$$

X_0 is the *left-hand side* and each X_i for $1 \leq i \leq n$ is a member of the *right-hand side* of the production.

Any production may have an empty right-hand side. Such productions are written as $A \Longrightarrow \lambda$. If $\alpha, \beta, \gamma \in V^*$ and $A \Longrightarrow \beta \in P$, we say that $\alpha A \gamma$ *directly derives* $\alpha\beta\gamma$, and write $\alpha A \gamma \Longrightarrow \alpha\beta\gamma$. The reflexive and transitive closure of $\Longrightarrow$ is denoted by $\overset{*}{\Longrightarrow}$.

We designate by SF(G) the set of *sentential-froms* derivable from S, that is

$SF(G) = \{ \alpha \in V^* \mid S \overset{*}{\Longrightarrow} \alpha \}$

The language generated by $G = (V_n, V_t, P, S)$ is $L(G) = SF(G) \cap V_t^*$.

To every sentential form there corresponds a *derivation tree.* The root of the tree is S. If $A \in V_n$ is rewritten using the production $A \Longrightarrow X_1 \dots X_m$, then A has $X_1 \dots X_m$ as the direct descendants.

A cfg is said to be *reduced* if $\forall A \in V_n$

(1) $S \overset{*}{\Longrightarrow} \dots A \dots$ and

(2) $\exists\ w \in V_t^*$ such that $A \overset{*}{\Longrightarrow} w$.

Definition : An *attributed grammar* AG is a 6-tuple $AG = (V_n, V_t, Q, P, S, F_Q)$.

Where V_n is a set of nonterminal symbols, V_t is a set of terminal symbols, disjoint from V_n, and Q is a set of primitive predicate symbols, disjoint from $V_n \cup V_t$.

For each grammar symbol $X \in V_n \cup V_t \cup Q$, there are two finite disjoint sets I(X) and S(X) of *inherited* and *synthesized* attributes respectively. For $X \in V_t$ we require that $I(x) = \Phi$. We write $A(X)$ for $I(X) \cup S(X)$. We assume that inherited attributes of the start symbol S are supplied before we start evaluating the attributes and synthetic attributes of a symbol from V_t are supplied with the terminal symbol.

P is a finite set of productions of the form

$$X_0 \downarrow a^0_1 \ \dots \downarrow a^0_{N_0} \uparrow b^0_1 \ \dots \uparrow b^0_{M_0} \Rightarrow$$

$$X_1 \downarrow a^1_1 \ \dots \downarrow a^1_{N_1} \uparrow b^1_1 \ \dots \uparrow b^1_{M_1} \dots X_n \downarrow a^n_1 \ \dots \downarrow a^n_{N_n} \uparrow b^n_1 \ \dots \uparrow b^n_{M_n}$$

Where $n \geq 0$, $X_0 \in V_n$, and $X_k \in V$ for $k > 0$.

The inherited attribute positions of the symbols are prefixed by "$\downarrow$" and synthesized attribute positions by "$\uparrow$". Each a^j_k or b^j_k is either an attribute variable or a constant attribute value.

Each production $p \in P$ has a set of attribute evaluation rules associated with it. A rule must be supplied for each *inherited* attribute appearing on the right-hand side of a production as well as for each *synthetic* attribute of the left-hand side. Attribute rules may only use attributes associated with the symbols of the corresponding production to compute values. These attribute evaluation rules are also called semantic functions. We will denote the semantic function for evaluating attribute v in production p by $f_{p,v}$. The value of attribute v in production p depends on some other attribute occurrences in p. We denote the set of these attributes occurrences by $D_{p,v}$. $D_{p,v}$ is called the *dependency set* of $f_{p,v}$. If $D_{p,v} = \{v_1, \dots, v_n\}$ then $f_{p,v}$ is a mapping

$$\text{domain}(v_1) \times \dots \times \text{domain}(v_n) \rightarrow \text{domain}(v)$$

Semantic functions which copy attribute values from one attribute position into another attribute position will not be stated explicitly. Rather, the same attribute name occurring in two or more attribute positions will indicate that the attribute value must be copied from a *defining* attribute position into *applied* attribute positions.

Definition : A *defining attribute position* is an inherited attribute position of the left-hand side of a production or a synthetic attribute position of a symbol on the right-hand side of a production. An *applied attribute position* is a synthetic attribute position of the left-hand side of a production or an inherited attribute position of a symbol on the right-hand side of a production.

F_Q is a finite set of *predicate functions.* For each $x \in Q$ there exists $f_x \in F_Q$ such that

f_x: $I(x) \rightarrow S(x) \times \{true, false\}$

f_x is total recursive over $I(x)$.

The role of a primitive predicate is twofold. Given the values for its inherited attributes, it evaluates its synthetic attributes. It also performs checks on the validity of the application of a production. Whenever it returns *false*, the presence of a context-sensitive error is detected. This corresponds to an "illegal" application of a production under the rules of the AG.

A parser for an attributed grammar AG is a context-free parser augmented by an evaluator which evaluates attributes as parsing progresses. The parser is constructed from the *head grammar* of AG defined as follows

Definition : The head grammar, HG, of an attributed grammar AG = (V_n, V_t, Q, P, S, F_Q) is a context-free grammar (V_n', V_t', P', S') obtained as follows :

1. The terminals of HG are terminals of AG ($V_t' = V_t$)
2. The set of nonterminals of HG includes nonterminals of AG and primitive predicates of AG ($V_n' = V_n \cup Q$)
3. The set of productions P' of H is obtained from the productions of AG by removing attributes. A production of the form q $\Longrightarrow \lambda$ is added to P' for each q $\in$ Q (primitive predicates)
4. The start symbol S' of H is the start symbol S of AG.

We illustrate the above definitions using a small example. This grammar defines a skeleton language in which identifier may be "declared" and "used", in which no identifier may be declared more than once and no identifier can be used without being declared first.

AG = (V_n, V_t, Q, <program>, P, F_Q) where

V_n = {<program>, <dec list>, <stmt list>, <var decl>, <var use>}

V_t = {ident, dcl, use, end, \$}

Q = { <declare>, <check> }

$F_Q = \{ f_{<declare>}, f_{<check>} \}$

P includes the following productions

p1: <program> $\Longrightarrow$ \$ BEGIN <dec list>$\downarrow\Phi\uparrow$symtab
    <stmt list>$\downarrow$symtab END \$

p2: <dec list>$\downarrow$symtab$_1$$\uparrow$symtab$_3$ $\Longrightarrow$ DCL
    <var decl>$\uparrow$id <declare>$\downarrow$symtab$_1$$\downarrow$id
    <dec list>$\downarrow$symtab$_2$$\uparrow$symtab$_3$

$\langle$dec list$\rangle_2$.symtab$_2$:= $\langle$dec list$\rangle_1$.symtab$_1$ $\cup$ {$\langle$var decl$\rangle$.id}

p3: $\langle$dec list$\rangle\downarrow$symtab$_1\uparrow$symtab$_2 \Longrightarrow \lambda$

$\langle$dec list$\rangle.\downarrow$symtab$_2$:= $\langle$dec list$\rangle.\uparrow$symtab$_1$

p4: $\langle$var decl$\rangle\uparrow$id $\Longrightarrow$ ident$\uparrow$id

p5: $\langle$stmt list$\rangle\downarrow$symtab $\Longrightarrow \lambda$

p6: $\langle$stmt list$\rangle\downarrow$symtab $\Longrightarrow$ USE $\langle$var use$\rangle\downarrow$symtab $\langle$stmt list$\rangle\downarrow$symtab

p7: $\langle$var use$\rangle\downarrow$symtab $\Longrightarrow$ ident$\uparrow$id $\langle$check$\rangle\downarrow$id$\downarrow$symtab

$f_{\langle check\rangle}$ = id $\in$ symtab

$f_{\langle declare\rangle}$ = id $\not\subset$ symtab

The head grammar, HG, (obtained automatically from the above) is given below.

HG = (V_n, V_t, $\langle$program$\rangle$, P) where

V_n = {$\langle$program$\rangle$, $\langle$dec list$\rangle$, $\langle$stmt list$\rangle$, $\langle$var del$\rangle$, $\langle$var use$\rangle$, $\langle$declare$\rangle$, $\langle$check$\rangle$}

V_t = {ident, dcl, use, end, \$}

P include the following productions

p1: $\langle$program$\rangle$ $\Longrightarrow$ \$ BEGIN $\langle$dec list$\rangle$ $\langle$stmt list$\rangle$ END \$

p2: $\langle$dec list$\rangle$ $\Longrightarrow$ DCL $\langle$var decl$\rangle$ $\langle$declare$\rangle$ $\langle$dec list$\rangle$

p3: $\langle$dec list$\rangle$ $\Longrightarrow$ λ

p4: $\langle$var decl$\rangle$ $\Longrightarrow$ ident p5: <stmt list> $\Rightarrow \lambda$

p6: <stmt list> $\Rightarrow$ USE <var use> <stmt list>

p7: <var use> $\Rightarrow$ ident <check>

p8: <check> $\Rightarrow \lambda$

p9: <declare> $\Rightarrow \lambda$

2.3. Problems With Attributed Grammars

There are several problems inherent in evaluating the attributes associated with the symbols of a syntax tree. First, the resulting definitions must be non-circular. A *circular* attributed grammar is one whose language contains a syntax-tree with an attribute that depends functionally upon its own value. Knuth [KN68] has given a circularity test for attributed grammars. Circular grammars are not of interest and are eliminated from all evaluation schemes.

Another problem in attribute evaluation is finding an order in which to evaluate the attributes of a sentence. One group of methods, known as *tree-walk* evaluators, relies on the prior completion of the syntactic analysis of a sentence. Notable among these methods are those proposed by Fang [FA72], Jazayeri [JW75], Bochman [BO76], and Kennedy and Warren [KW78].

A second class of attribute evaluators determines attribute values during syntactic analysis. We will refer to these methods as on-line attribute evaluators. Two such parse-time evaluators are described by Lewis et al [LRS74]. A left-to-right bottom-up parsing method, LR(k) for instance, finds all offspring and their subtrees before any parent. As a recognized production's right-hand side is reduced to its left-hand side symbol, synthetic attributes can be evaluated for the left-hand side if all attributes in the offspring are evaluated, and no attribute of the root depends on any inherited attribute of the root. Synthetic or S-attributed grammars meet this restriction.

Top-down parsing methods like LL(k) recognize nodes of a syntax tree in a different manner. All parents are recognized before their offsprings, and each child is found before its siblings to the right. The LL(k) parser performs a depth-first left-to-right walk through a syntax tree. Any attributed grammar that is LL(k) and that can be evaluated by a single depth first left-to-right walk for any sentence in the language can be evaluated during an LL(k) parse. Details of the stack machine that performs the evaluation are presented in the Lewis *et. al.* [LRS74]. The attributed grammars allowed by this method are termed *L-attributed Grammars.*

The definitions for L-attributed and S-attributed grammars are adopted from Lewis et al [LRS74].

Definition : An attributed grammar is *S-attributed* if all attributes are synthetic.

Definition : An attributed grammar is *L-attributed* if for each production $p \in P$ of the form

$$X_0 \Rightarrow X_1 \dots X_n$$

1. The synthetic attributes of X_0 are only dependent upon the inherited attributes of X_0 and arbitrary attributes of the symbols X_1 through X_n.
2. The inherited attributes of X_k are only dependent upon the inherited attributes of X_0 and arbitrary attributes of $X_1 \dots X_{k-1}$.

These two classes of attributed grammars, S-attributed and L-attributed, do not include many desirable properties of full attributed grammars. Forward references, for example, cannot be handled in either class without undue complication.

Rowland [RO77] extended this work of combining parsing and attribute evaluation for a general attributed grammar. The parsing method he used was GLC(k) [RO77]. A subtree is retained until all the attributes of its root are evaluated. Once all the attributes of a root are known, the subtree below it can be discarded, as it will not be required any further for attribute evaluation.

We would like to extend this work and develop an on-line attribute evaluator for any context-free grammar with a general non-circular attributed grammar. Our method is based on attribute evaluator proposed be Kennedy and Warren [KW78], but it runs in parallel with the parser. The parsing method we have selected is the one proposed by Graham et al [GHR80].

The rest of this chapter is organized as follows. In the next section we will give a brief review of the GHR parsing algorithm. Then we will show how to combine attribute evaluation with parsing (separate sections for the S-attributed, the L-attributed and the absolutely non-circular (defined later) attributed grammars) assuming that there are no predicates in the grammar. Finally we will show how to include primitive predicates.

2.4. Brief Review of the GHR Parsing Method

The parsing method proposed by Graham et al can be used to parse any context-free grammar. It is a "dynamic programming" method in which derivations matching longer portions of the input are build up by pasting together previously computed derivations matching shorter portions. In order to handle arbitrary grammars, it uses the idea of matching only part of the right-hand side of a production rule to the input, rather than the whole production rule. As a notation for dealing with this situation it uses dotted rules as defined below.

Definition : Let $G = (V_n, V_t, S, P)$ be a context-free grammar and let "." be a symbol not in $V_n \cup V_t$. If $A \Longrightarrow \alpha\beta$ is in P, then $A \Longrightarrow \alpha.\beta$ is a dotted rule of G.

The idea is that a dotted rule $A \Longrightarrow \alpha.\beta$ indicates that α has been matched to the input, but it is not yet known whether β matches (dotted rules are similar to items in LR parse states).

To parse an input string $w = a_1 \ldots a_n$, the GHR parser constructs an $(n+1)X(n+1)$ upper triangular matrix $t = (t_{i,j})$ $0 \le i \le j \le n$, whose entries are sets matching substrings of the input. Elements of these sets are dotted rules.

If $A \Longrightarrow \alpha.\beta$ is a dotted rule, we say that $A \Longrightarrow \alpha.\beta$ matches x if $\alpha \overset{*}{\Longrightarrow} x$. $B \in V_n$ matches x if $B \overset{*}{\Longrightarrow} x$. If $A \Longrightarrow \alpha.B\beta$ matches x and B matches y, then we can "paste" the derivations together to conclude that $A \Longrightarrow \alpha B.\beta$ matches xy. The operations needed to paste two derivations are defined next.

Definition : Let $G = (V_n, V_t, S, P)$ be a context-free grammar. Let Q be a set of dotted rules and R be a set of symbols (a subset of $V_n \cup V_t$). Define

$$QxR = \{A \Longrightarrow \alpha B\beta.\gamma \mid A \Longrightarrow \alpha.B\beta\gamma \in Q,\ \beta \overset{*}{\Longrightarrow} \lambda, \text{ and } B \in R\}$$

$$Q^*R = \{A \Longrightarrow \alpha B\beta.\gamma \mid A \Longrightarrow \alpha.B\beta\gamma \in Q,\ \beta \overset{*}{\Longrightarrow} \lambda, \text{ and } B \overset{*}{\Longrightarrow} C \text{ for some } C \in R\}$$

The possibility of both arguments for x and * product being dotted rules is included in the following definitions.

Definition : Let $G = (V_n, V_t, S, P)$ be a context-free grammar. Let Q,R be sets of dotted rules, then

$$Q \times R = \{A \Longrightarrow \alpha B\beta.\gamma \mid A \Longrightarrow \alpha.B\beta\gamma \in Q,\ \beta \overset{*}{\Longrightarrow} \lambda, \text{ and } B \Longrightarrow \rho. \in R\}$$

$$Q{*}R = \{A \Longrightarrow \alpha B\beta.\gamma \mid A \Longrightarrow \alpha.B\beta\gamma \in Q,\ \beta \overset{*}{\Longrightarrow} \lambda, \text{ and } B \overset{*}{\Longrightarrow} C \text{ for some } C \Longrightarrow \rho. \in R\}$$

Informally, the parsing algorithm works as follows. Let $w = a_1 \ldots a_n$ be the input string we want to parse. Denote $a_{i+1} \ldots a_j$ by $w_{i,j}$ and $a_1 \ldots a_i$ by w_i. In $t_{i,j}$ we include the dotted rule $A \Longrightarrow \alpha.\beta$ iff $\alpha \overset{*}{\Longrightarrow} w_{i,j}$. Further if we have $A \Longrightarrow \alpha.B\beta$ in $t_{i,j}$ and we have $B \Longrightarrow \gamma.$ in $t_{j+1,k}$ then we can include $A \Longrightarrow \alpha B.\beta$ in $t_{i,k}$ (we have left out few details for handling chain rules and λ productions). Instead of finding all the dotted rules $A \Longrightarrow \alpha.\beta$ which match some substring $w_{i,j}$ of the input, we need find only those dotted rules $A \Longrightarrow \alpha.\beta$ which match $w_{i,j}$ *and* may legally follow the parsed left context w_i. The GHR algorithm does this by using a PREDICT function, which is defined as follows.

Definition : Let $G = (V_n, V_t, S, P)$ be a context-free grammar and let R be a subset of $V_n \cup V_t$. Then define

$$\text{PREDICT}(R) = \{C \Longrightarrow \alpha.\beta \mid C \Longrightarrow \alpha\beta \in P,\ \alpha \overset{*}{\Longrightarrow} \lambda,\ B \overset{*}{\Longrightarrow} C\gamma \text{ for some } B \in R \text{ and } \gamma \in (V_n \cup V_t)^*\}$$

If R is a set of dotted rules then

$$\mathrm{PREDICT}(R) = \mathrm{PREDICT}(\{B \mid A \Rightarrow \alpha.B\beta \in R\})$$

The complete GHR parsing algorithm is given below in a Pascal-like notation (adopted from [GHR80]).

Algorithm 1

Let $G = (V_n, V_t, S, P)$ be any context-free grammar. Let $w = a_1 \ldots a_n$, where $n \geq 0$ and $a_k \in V_t$ for $1 \leq k \leq n$, be the string to be recognized. Form an $(n+1)X(n+1)$ matrix $t = (t_{i,j})$ as follows

```
begin
 $t_{0,0}$ := PREDICT({S}); (* Match $\lambda$ *)
for j := 1 to n do
begin (* Build column j, given columns 0 ... j-1 *)
   $t_{j-1,j} := t_{j-1,j-1} * \{a_j\}$ (* paste input symbol to $\lambda$
                                derivations that precede it *)
  for i := j-2 downto 0 do
    begin
      r := ( $\bigcup_{i<k<j-1} (t_{i,k} \times t_{k,j})) \cup$
           $(t_{i,j-1}) \times (t_{j-1,j} \cup \{a_j\})$;
         (* Paste non-$\lambda$ derivations *)
      $t_{i,j}$ := r $\cup$ $t_{i,i}$ * r; (* Paste matched suffix to
                  $\lambda$ derivations that precede it and
                  extend match to reflect chain rules*)
    end;
  $t_{i,j}$ := PREDICT( $\bigcup_{0\leq i<j} t_{i,j}$);
  end;
```

```
    if some $S \Rightarrow \alpha. \in t_{0,n}$ then accept
                                  else reject
    end.
```

For complete details and number of possible variations to this algorithm see [GHR80].

2.5. S-attributed Grammars

If the attributed grammar corresponding to the head grammar we are parsing is S-attributed (as defined above), then attribute evaluation while parsing is very easy. As a recognized production's right-hand side is reduced to its left-hand side symbol, synthetic attributes of the left-hand side symbol can be evaluated, as all the attributes of the right-hand side symbols (which have been recognized) are known (this follows directly from the definition of S-attributed grammar). Therefore, before we "paste" a newly recognized symbol into another dotted rule, we compute its synthetic attributes and include them with the symbol. Therefore in any dotted rule $A \Rightarrow \alpha.\beta$, all the symbols of α will have their attributes evaluated.

Now we shall redefine the $\times$ and * product so that the attributes of a symbol are evaluated as it is recognized.

Note that for all the sets used in the following definitions we will treat a grammar symbol with different attribute values as distinct set elements. Therefore, a set $S = \{A\uparrow a_0, A\uparrow a_1\}$ has two elements if a_0 and a_1 are two attribute constants with different values. Similarly a set containing two dotted rules with the same grammar symbols and the dot in the same place are two distinct set elements if there exists at least one attribute position where two dotted rules have different attribute values.

Further, we require that for each symbol D such that $D \overset{*}{\Longrightarrow} \lambda$, we evaluate the attributes of D when we move the dot over D. These attributes can be precomputed. But we can have a problem if the grammar is ambiguous. For example consider the following attributed grammar.

$B{\uparrow}a_B \Longrightarrow A{\uparrow}a_A$

$\quad B.a_B := A.a_A;$

$A{\uparrow}a_A \Longrightarrow A{\uparrow}a_A$

$\quad A[1].a_A := A[2].a_A + 1;$

$A{\uparrow}a_A \Longrightarrow \lambda$

$\quad A.a_A := 1;$

Now $B{\uparrow}a_B \overset{*}{\Longrightarrow} \lambda$, but we cannot determine a unique value for attribute a_B. If we include B a number of times with different attribute values, the number of dotted rules can grow indefinitely. There are two alternatives to deal with situation. We can either take the smallest derivation of $B \overset{*}{\Longrightarrow} \lambda$ and precompute the attributes of B (this can be done for S-attributed grammars; in arbitrary attributed grammars, the synthetic attributes of a symbol may depend on the inherited attributes of that symbol). Our choice of smallest derivation is not arbitrary: If we use the GHR parsing method and get a parse tree before evaluating the attributes, the parse tree will have the smallest subtree for each $B \overset{*}{\Longrightarrow} \lambda$. Hence what we are doing is what would be done by a tree-walk evaluator.

The other alternative to deal with the above mentioned problem is to have a user defined semantic functions (or a predicates) which will compute the synthetic attributes of B of each B $\Rightarrow^* \lambda$. These semantic functions will use the attributes of the lookahead symbol (and the inherited attributes of B when we have arbitrary attributed grammars) to decided the synthetic attributes of B. These semantic functions will decide how many times we should loop in the derivation B $\Rightarrow^* \lambda$ for computing the synthetic attributes of B. We feel that using predicates or semantic functions to decide how many times we should loop is very attractive as this will require no modification in the GHR parsing algorithm and we can deal with this rather uncommon situation.

We can get into the similar problem with chain rules. For example consider the following attributed grammar.

$B\uparrow a_B \Rightarrow A\uparrow a_A$
  $B.a_B := A.a_A;$
$A\uparrow a_A \Rightarrow A\uparrow a_A$
  $A[1].a_A := A[2].a_A + 1;$
$A\uparrow a_A \Rightarrow C\uparrow a_C$
  $A.a_A := C.a_C;$

Now $B\uparrow a_B \Rightarrow^* C\uparrow a_C$, but we cannot determine a unique value for attribute a_B from the value of a_C. Here again we have two alternatives to deal with this situation. We can either choose the smallest possible derivation $B\uparrow S_B \Rightarrow^* C\uparrow S_C$ in determining the attributes of B from the attributes of C. Or alternatively we can have a user defined semantic functions (or a predicates) which will compute the synthetic attributes of B of each B $\Rightarrow^*$ C. These semantic functions will use the attributes of the symbol C and the attributes of the lookahead symbol (and the inherited attributes of B when we have arbitrary attributed grammars) to decided the synthetic attributes of B. These semantic functions will decide how many times we should loop in the derivation $B \stackrel{*}{\Longrightarrow} C$ for computing the synthetic attributes of B.

In the definitions given below only some of the symbols in a dotted rule will have their attributes fully evaluated (symbols appearing left of the dot). Attributes unavailable for a symbol will not be shown. Further, note that $\overline{\beta}$ is obtained from the β after evaluating the attributes of all the symbols in it.

Definition : Let AG = (V_n, V_t, Q, S, P, F_Q) be a S-attributed grammar. Let Q be a set of dotted rules and let R be a set of attributed symbols with *all* the attributes evaluated. Define

$Q{\times}R = \{A \Longrightarrow \alpha B{\uparrow}S_A\overline{\beta}.\gamma \mid A \Longrightarrow \alpha.B\beta\gamma \in Q,\ \beta \stackrel{*}{\Longrightarrow} \lambda$, and $B{\uparrow}S_A \in R$. Attributes of B are also included with the symbol. For each grammar symbol C such that $\beta = \rho C\omega$ the attributes of C are evaluated according to the production rule $C \stackrel{*}{\Longrightarrow} \lambda\ \}$

Note that attributes of B are included with the symbol (as we are taking the x product of a set of dotted rules with a set of attributed symbols). Further, if we move the dot over a symbol because it derives λ, then we compute its synthetic attributes (as explained before) and include them with the symbol.

$Q{*}R = \{A \Longrightarrow \alpha B{\uparrow}S_A\overline{\beta}.\gamma \mid A \Longrightarrow \alpha.B\beta\gamma \in Q,\ \beta \stackrel{*}{\Longrightarrow} \lambda$, and $B \stackrel{*}{\Longrightarrow} C{\uparrow}S_C$ for some $C{\uparrow}S_C \in R$. Also evaluate the attributes of B before pasting it into the dotted rule and for each grammar symbol D such that $\beta = \rho D\omega$, evaluate the attributes of D according to the production $D \stackrel{*}{\Longrightarrow} \lambda\ \}$

Note that from the attributes of C we can compute the synthetic attributes of B if $B \overset{*}{\Rightarrow} C$ and we are parsing an S-attributed grammar.

Let Q,R be sets of dotted rules. Define x and * products as follows

QxR = $\{A \Rightarrow \alpha B{\uparrow}S_A\bar{\beta}.\gamma \mid A \Rightarrow \alpha.B\beta\gamma \in Q,\ \beta \overset{*}{\Rightarrow} \lambda$, and $B{\uparrow}S_B \Rightarrow \delta. \in R$; Evaluate attributes of B and include them with the symbol, and for each grammar symbol C such that $\beta = \rho C\omega$, evaluate the attributes of C according to the production rule $C \overset{*}{\Rightarrow} \lambda\ \}$

If $B \overset{*}{\Rightarrow} \delta. \in R$ then attributes of the symbols on the right-hand are known (they were evaluated as we recognized δ). Using these attributes we compute the synthetic attributes of B before we move the dot over B.

Q*R = $\{A \Rightarrow \alpha B{\uparrow}S_A\bar{\beta}.\gamma \mid A \Rightarrow \alpha.B\beta\gamma \in Q,\ \beta \overset{*}{\Rightarrow} \lambda$, and $B{\uparrow}S_B \overset{*}{\Rightarrow} C{\uparrow}S_C$ for some $C{\uparrow}S_C \Rightarrow \delta. \in R$. Also evaluate the attributes of B before pasting it into the dotted rule and for each grammar symbol D such that $\beta = \rho D\omega$, evaluate the attributes of D according to the production $D \overset{*}{\Rightarrow} \lambda$ $\}$

The PREDICT function remains same. The parsing algorithm is same as given above except that it will use the new definitions of x and * product. These modifications can be incorporated easily into the various implementations suggested by Graham et al [GHR80] for x and * products.

2.6. L-Attributed Grammars

If the attributed grammar corresponding to the head grammar we are parsing is L-attributed (as defined above) then on-line attribute evaluation is also very easy.

From the GHR algorithm it is clear that we predict a symbol when we have the dot before it. That means all the symbols left of the dot have been recognized. If the grammar is L-attributed, then all the inherited attributes of the symbol being predicted can be computed (this follows directly from the definition of the L-attributed grammar). Hence when we predict a symbol, we also include its inherited attributes. Here again grammar symbols with different attribute values will be considered distinct. Hence if we have two dotted rules predicting the same grammar symbol with different inherited attribute values, then we include that symbol twice.

Now we will redefine * and x products and PREDICT function for L-attributed grammars.

Note that when we paste an attributed symbol $B{\downarrow}I_B{\uparrow}S_B$ into a dotted rule, we require that the inherited attributes of B match the inherited attributes of B in the dotted rule. This is due to the fact that if B has different inherited attributes then it must have been predicted by some other dotted rule and therefore, cannot be pasted with this dotted rule.

In the definitions given below only some of the symbols in a dotted rule will have their attributes fully evaluated (symbols appearing left of the dot). Attributes unavailable for a symbol will not be shown. For example, if only inherited attributes are known for B, then we shall represent this fact by using $B{\downarrow}I_B$ $({\uparrow}S_B$

will not be shown). Further, we make following observations about attribute evaluation while taking x or * product.

If we move the dot over a symbol C because C $\Rightarrow^* \lambda$, then we get its inherited attributes from the symbols left of it (since those symbols will have all their attributes evaluated). Using these inherited attributes of C, we compute synthetic attributes of C (If C $\Rightarrow^* \lambda$ then the synthetic attributes of C are a function of only the inherited attributes of C). There are two alternatives for evaluating synthetic attributes of C from its inherited attributes if C $\Rightarrow^* \lambda$. We can do a pre-analysis and come up with the semantic functions for evaluating synthetic attributes of C from the inherited attributes of the C (these functions will be similar to plans defined in the next section); or we can implement the x and * products such that the parsing algorithm does not have to know in advance which symbol can derive λ. This information can be computed at run time while parsing. (For more detail on this see [GHR80]). From our implementation experience, we find the second alternative to be more attractive.

Furthermore, if we move the dot over a sequence of symbols, because they all derive λ, then we must move the dot over one symbol at a time, compute its attributes (inherited and synthetic), then move the dot over the next symbol, and so on. Here again we can implement the x and * products such that the parsing algorithm does not have to know in advance which symbol can derive λ, but computes it at run time. This technique can deal with predicates without any undue complication (we will discuss the predicates in more detail in the section 2.9).

If $B{\downarrow}I_B \Rightarrow^* C{\downarrow}I_C{\uparrow}I_C$, then using inherited attributes of B and attributes of C (both inherited and synthetic), we compute the synthetic attributes of B before we move the dot past B. Again semantic functions for evaluating these attributes can be established in advance, or we can implement the * product such that the parsing algorithm does not have to know in advance about the chain rules, but computes them at run time while parsing.

Finally, when we have $B{\downarrow}I_B \Rightarrow \delta$, all the symbols on the right-hand side will have their attributes evaluated (they were evaluated as we recognized δ). From these attributes and inherited attributes of B we compute the synthetic attributes B before we include B into a dotted rule.

Definition : Let AG = (V_n, V_t, Q, S, P, F_Q) be an L-attributed grammar. Let Q be a set of dotted rules and let R be a set of attributed symbols with all the attributes evaluated. Define

QxR = $\{A{\downarrow}I_A \Rightarrow \alpha B{\downarrow}I_B{\uparrow}S_B\overline{\beta}.\gamma \mid A{\downarrow}I_A \Rightarrow \alpha.B{\downarrow}I_B\beta\gamma \in Q$, $\beta \overset{*}{\Rightarrow} \lambda$, and $B{\downarrow}I_B{\uparrow}S_A$ $\in$ R. Attributes of B are also included with it. Let $\beta = C_1 \ldots C_n$. Go from left to right computing the inherited attributes of C_i and using these inherited attributes to compute the synthetic attributes of C_i before considering C_{i+1} $\}$

Q*R = $\{A{\downarrow}I_A \Rightarrow \alpha B{\downarrow}I_B{\uparrow}S_B\overline{\beta}.\gamma \mid A{\downarrow}I_A \Rightarrow \alpha.B{\downarrow}I_B\beta\gamma \in Q$, $\beta \overset{*}{\Rightarrow} \lambda$, and $B{\downarrow}I_B{\uparrow}S_B$ $\overset{*}{\Rightarrow} C{\downarrow}I_C{\uparrow}S_C$ for some $C{\downarrow}I_C{\uparrow}S_C \in$ R. Evaluate the attributes of B before pasting it into the dotted rule and let $\beta = D_1 \ldots D_n$. Go from left to right computing the inherited attributes of D_i and using these inherited attributes to compute the synthetic attributes of D_i before considering D_{i+1} $\}$

Let Q,R be sets of dotted rules then define x and * products as follows

QxR = $\{A{\downarrow}I_A \Rightarrow \alpha B{\downarrow}I_B{\uparrow}S_B\overline{\beta}.\gamma \mid A{\downarrow}I_A \Rightarrow \alpha.B{\downarrow}I_B\beta\gamma \in Q$, $\beta \overset{*}{\Rightarrow} \lambda$, and $B{\downarrow}I_B{\uparrow}S_B$

$\Rightarrow \delta. \in R$. Attributes of B are evaluated before including it into the dotted rule. Let $\beta = C_1 \ldots C_n$. Go from left to right computing the inherited attributes of C_i and using these inherited attributes to compute the synthetic attributes of C_i before considering $C_{i+1}\}$

$Q^*R = \{A{\downarrow}I_A \Rightarrow \alpha B{\downarrow}I_B{\uparrow}S_B\overline{\beta}.\gamma \mid A{\downarrow}I_A \Rightarrow \alpha.B{\downarrow}I_B\beta\gamma \in Q$, $\beta \overset{*}{\Rightarrow} \lambda$, and $B{\downarrow}I_B{\uparrow}S_B \overset{*}{\Rightarrow} C{\downarrow}I_C{\uparrow}S_C$ for some $C{\downarrow}I_C{\uparrow}S_C \Rightarrow \delta. \in R$. Evaluate the attributes of B before pasting it into the dotted rule. Let $\beta = D_1 \ldots D_n$. Go from left to right computing the inherited attributes of D_i and using these inherited attributes to compute the synthetic attributes of D_i before considering D_{i+1} $\}$

Now we will define the PREDICT function for an L-attributed grammar.

Definition : Let $AG = (V_n, V_t, Q, S, P, F_Q)$ be an L-attributed grammar and let R be a set of attributed symbols with all the inherited attributes evaluated. Define

$\text{PREDICT}(R) = \{C{\downarrow}I_C \Rightarrow \overline{\alpha}.\beta \mid C \Rightarrow \alpha\beta \in P$, $\alpha \overset{*}{\Rightarrow} \lambda$, and $B \overset{*}{\Rightarrow} C\rho$ for some $B{\downarrow}I_B \in R$ and some $\rho \in V^*$. Evaluate the inherited attributes of C from the inherited attributes of B. Let $\alpha = D_1 \ldots D_n$. Go from left to right evaluating the inherited attributes of D_i and using these attributes to compute the synthetic attributes of D_i before considering D_{i+1} $\}$

If R is a set of dotted rules then define

$\text{PREDICT}(R) = \text{PREDICT}(\{B{\downarrow}I_B \mid A{\downarrow}I_A \Rightarrow \alpha.B\beta \in R$. Compute all the inherited attributes of B$\})$

The rest of the parsing algorithm remains same.

2.7. Non-Circular Attributed Grammar

When we have a general non-circular attributed grammar, attributes can be evaluated as follows.

In every attribute domain we can include a special value called 'unknown'. If we have an attribute position whose value is not available when we move the dot over a symbol (and cannot be computed at this moment) then we put the value 'unknown' in that position.

Eventually we will put some legal value from the attribute domains in all the attribute positions where we have put the 'unknown'. One approach for doing this would be to delay evaluations of those attribute positions where we had to put the 'unknown' until the end of parsing. At the end we have the complete parse tree. Therefore, we can evaluate all the attributes. This approach is not very attractive for the following two reasons.

First, because of lack of information (or attributes), some dotted rules which could be deleted by the predicate functions will not be deleted (the exact use of predicates will be explained later). Further, we could end up with a parse tree that is semantically wrong. Hence at the end we would have to modify the parse tree to create the one which is correct for the given input.

Second, as we parse further to the right, there will tend to be more and more attribute positions with the 'unknown' values (as we cannot evaluate an attribute unless *all* the attributes on which it depends have some legal value from their corresponding domains). Hence towards the end we may well be parsing without any attributes.

An alternative approach would be to evaluate as many attributes as possible at any given time as follows.

When we move the dot over a terminal or a non-terminal in a dotted rule, we examine all the symbols left of it whose attributes could be affected by the new symbol. Additional inherited attributes could become available for these symbols. We move these additional attributes to the symbols to the left of the dot. These additional inherited attributes could make it possible to compute more synthetic attributes. If so, we then reconsider the symbols which are used in forming this dotted rule and get these additional synthetic attributes. Using attribute dependency information we can avoid any unnecessary reconsideration of a symbol (see [KW78]). There is one problem that needs to be considered in the above method. Since a symbol could be predicted by more than one production, it is possible that more than one dotted rule is using the same symbol (because two or more productions could predict the same symbol with the same subset of inherited attributes). When we reconsider a symbol (because some additional inherited attributes are available) with additional inherited attributes, we cannot attach these attributes to the symbol, as other dotted rules using this symbol could have different inherited attribute values for it (because different productions can have different attribute evaluation rules). This problem can be solved as follows.

When we reconsider a symbol, we do not change its attributes. When we reconsider a symbol with more inherited attributes, it will return additional output (synthetic) attributes without changing any of its existing attributes. This approach could turnout to be bit expensive. We can make it more efficient by putting a count with each symbol, which says how many dotted rules are pointing to it. If there is only one dotted rule pointing to it then we can change its attributes from the 'unknown' to some specific value freely. But keeping a proper reference count could also be unacceptable due to additional links required. An alternative to reference count would be to have multiple copies of the symbols which are used by more than one dotted rule. We can easily incorporate this technique without changing the basic algorithm. When a symbol is predicted by more than one dotted rule, we currently include it a number of times with different inherited attributes. Now we include a symbol a number of times with the same subset of inherited attributes if it is predicted by different dotted rules. That is, since an attribute position which contains 'unknown' can be expanded in many different ways, we have a choice to delay the expansion (by keeping the attribute position 'unknown') or follow multiple different expansions (by making multiple copies of the symbols predicted by different dotted rules).

When we have a general attributed grammar (not S-attributed or L-attributed) then we cannot evaluate *all* the attributes of a symbol as it is recognized. Therefore, we will try to evaluate as many attributes as possible as we recognize a symbol. As it becomes possible to evaluate additional attributes, we revisit the symbol and evaluate these additional attributes. To see which attributes can be evaluated at any given time, we use attribute dependency information. This information can be extracted from the grammar definition. More specifically, suppose we have a production $p \in P$

$$p: X_0 \Longrightarrow X_1 \dots X_n$$

Then we must have a semantic function for each synthetic attribute of X_0 and for each inherited attributes of X_1 through X_n. The relationship among attributes occurrences of a given production can be represented by its dependency graph, denoted by D(p), defined as follows.

$D(p) = (V_p, E_p)$

The node set V_p is the set of all attribute occurrences of p and the edge set is the set of dependency pairs for p. Formally

$V_p = \{b \mid b \text{ is an attribute occurrence in } p\}$

$E_p = \{(b_1, b_2) \mid \text{value of attribute } b_2 \text{ is directly dependent on the value of attribute } b_1\}$

This graph is always acyclic for non-circular attributed grammar and hence yields a partial order for attribute evaluation.

If we have a parse tree T of an attributed grammar, we can construct a dependency graph D(T) that represents all data flow paths in the tree. This graph is the result of "pasting together" copies of D(p)'s for productions occurring in the tree. Let T be a derivation tree; P: $X_0 \Longrightarrow X_1 \ldots X_n$ is the production rule applied at the root of T and T_k is the kth subtree of T, D(T) is recursively constructed from D(p), $D(T_1)$, ... ,$D(T_n)$ by identifying the nodes for attribute occurrences of X_k in D(p) with the corresponding nodes for the attribute occurrences of the root of T_k in $D(T_k)$, $1 \leq k \leq n$.

If a subtree is detached from the rest of the tree, some data flow paths through the root of the subtree will be interrupted. Data flows into the subtree through inherited attributes and out of the subtree through synthetic attributes. Therefore, we will refer to inherited attributes of a symbol as *input* attributes and synthetic attributes of a symbol as *output* attributes. The dependency graph of a subtree may imply that certain input attributes must be made available before an output attribute can be evaluated. These relationships are called *input-output* dependencies of the subtree.

Similarly, we can associate *input-output* dependencies among attributes of a nonterminal. With each nonterminal X of the grammar, we will associate an "I/O graph" IO_X, which will describe the input-output dependencies of subtrees with root X.

Different subtrees with the same root symbol may exhibit different input-output dependencies. So we cannot hope to characterize a nonterminal precisely by means of single graph. Instead, IO_X will present a worst-case picture of input-output dependencies.

In order to compute IO_X, we first show how to find input-output dependencies of X_0 with respect to a production P: $X_0 \Longrightarrow X_1 \ldots X_n$. To find input-output dependencies of X_0 with respect to a production, we will use I/O graphs of the symbols $X_1 \ldots X_n$ and D(p), the dependency graph for production p, as defined before. This gives us a recursive definition of the IO_X's.

Let D(p) be the dependency graph associated with the production P. Suppose that for k = 1, ... ,n we have a directed graph G_k whose nodes are $A(X_k)$. Then let $D_p[G_1, \ldots, G_n]$ be a directed graph obtained from D(p) by adding an arc from attribute occurrence *a* to attribute occurrence *b* (both *a* and *b* are attributes of the same symbol X_k, $0 \le k \le n$), whenever there is an arc from *a* to *b* in G_k.

Definition : The *augmented dependency graph* $D^*(p)$ for a production p: $X_0 \Longrightarrow X_1 \ldots X_n$ is a graph $D_p[IO_{X_1}, \ldots, IO_{X_n}]$, where IO_{X_i} is an empty graph if $X_i \in V_t$.

The graph $D^*(p)$ is the dependency graph for a production p extended to include potential dependency chains through subtrees. Any path in $D^*(p)$ from an input attribute of X to an output attribute of X represents an apparent input-output dependency of X, and must be represented by an arc in IO_X. We therefore, have the following recursive definition (adopted from [KW78]).

Definition : The *set of I/O graphs* is a set $\{IO_X\}$ of directed graphs indexed by V_n satisfying

(a) The nodes of the graph IO_X are the attributes A(X) and

(b) There is an arc from i to s in IO_X iff there is a path from i to s in the graph $D^*(p)$ for some production $p \in P$ whose left-hand side is X.

Hence we can use following iterative algorithm to compute the set of I/O graphs.

Initially let each IO_X have nodes A(X) and no arcs. Then repeat until no more arcs are added to any IO_X: If there is production $p \in P$ with left-hand side X such that $D^*(p)$ has a path from i to s but IO_X has no arc from i to s, add the missing arc to IO_X.

The algorithm must terminate since there are only a finite number of arcs possible.

The arcs in IO_X reflect all the actual input-output dependencies that could exist in a subtree with root X. However, not every arc necessarily represents a possible input-output dependency; the IO_X's may be excessively pessimistic, asserting I/O dependencies that no subtree could actually exhibit. This phenomenon occurs because we are merging all I/O relationship of a nonterminal into a single graph. This information loss may prevent evaluator construction by making it seem that an evaluator is deadlocked in certain situation when it is not. For a wide class of attributed grammar we can be sure that the evaluator will not be deadlocked. This subclass of attributed grammars, called "absolutely non-circular", is defined as follows.

Definition : An attributed grammar is *absolutely* non-circular if no $D^*(p)$ contains a directed cycle.

Most of the practical attributed grammars are absolutely non-circular. Even if an attributed grammar is not absolutely non-circular, it can be converted into an equivalent absolutely non-circular grammar. An algorithm for coverting an arbitrary non-circular attributed grammar into an absolutely non-circular attributed grammar has been suggested by Katayama [KA80]. Basically if we have a symbol X with cycles in IO_X, then we can replace X by a set of nonterminal symbols, one for each production $p \in P$ having X on the left-hand side. For further details see [KA80].

Since we can transform any non-circular attributed grammar into an equivalent absolutely non-circular grammar, we shall consider only absolutely non-circular grammars in the rest of this chapter.

Now we will first briefly review the *tree-walk* evaluator described by Kennedy and Warren [KW78]. Then we will show how to extend it so that it can run in parallel with the GHR parser. The purpose of performing dependency analysis in advance is to avoid run-time analysis and blind searches through a tree. From the IO_X graph we can predict which synthetic attributes of X can be evaluated from a given set of its inherited attributes. Therefore, if we visit a node labeled X with inherited attributes T_X (a subset of I(X)), on return from node X we can be sure to have all those synthetic attributes of X evaluated which are dependent only on the attributes from T_X. Hence for each possible input attribute set we could have on visiting a node X, we can predetermine which synthetic attributes can be evaluated. Further, if we know the production used at that node then we can write down a sequence of instructions for evaluating these attributes (the production used at the node is required to find out which semantic function to use in evaluating various attributes). We will refer to these sequence of instructions as "plans".

Definition : A plan is a sequence of instructions. There are two kinds of instructions in a plan.

1. Execute a semantic function of some production.
2. Visit a node.

The set of all plans is denoted by PLAN.

When we visit a node labeled X with certain input attributes I_X, we would have certain output attributes available on return from the node. On our next visit to the same node with more input attributes, we would be able to evaluate some additional output attributes. To make sure that we do not reevaluate a previously evaluated attribute, we associate a flag with each node. The flag of a node tells us what attributes have already been evaluated at the node. Hence if a node labeled X is flagged with q and we visit this node with input attributes T_X, we can predict in advance what additional attributes can be evaluated, and if we know the production used at the node, then we can make a "plan" to evaluate only these additional attributes. Therefore, to know which plan to execute at a node on a visit we need three things: the flag at the node, the current input attributes, and the production used at the node. This information can be stored into a table, which we will call the "NEXTMOVE" table. In order to reduce the number of parameters for table lookup we shall combine the flag of a node with the production used at the node into one value and call it the *state* of the node. Formally :

Definition : A state of a node denoted by q is a pair (p,A) where $p \in P$ and A is a set of attribute occurrences of p.

The set of all states is denoted by Q.

"NEXTMOVE" can be defined as follows.

Definition : NEXTMOVE is a partial function assigning a plan and next state to a node for certain state/current-input attributes pair.

NEXTMOVE : Q X $\mathbb{I}$ → PLAN X Q, where $\mathbb{I}$ is a set of all current-input attribute sets.

Now we are ready to define a tree-walk evaluator. It works as follows.

Initially we flag each internal node of the parse tree with an initial state indicating that the node has never been visited and therefore, no attributes are available. Then we visit the root (of the complete parse tree) with the inherited attributes of S (the start symbol). From the definition of attributed grammar we know that inherited attributes of the start symbol are supplied before we start evaluating attributes of a parse tree. Formally :

Definition : A tree walk evaluator is a 5-tuple (Q, q_0, $\mathbb{I}$, NEXTMOVE, PLAN) where

Q : is a finite set of states

q_0 : P → Q assigns an *initial state* $q_0(p)$ to each production $p \in P$

$\mathbb{I}$: is a finite set of *current-input sets* , containing an empty set as an element

PLAN : is a finite set of "plans" (as defined before)

NEXTMOVE : Q X $\mathbb{I}$ → PLAN X Q is a partial function mapping state/current-input set pair into a plan/state pair. The output plan is the plan we want to execute at the node and assign the output state before returning from the node.

The complete tree-walk evaluator is

```
Procedure Evaluate(t,I); (*where t is a node and I is
                    a current-input-set *)
    Let q be the current state at the node t;
    Let NEXTMOVE(q,I) be equal to <plan,q'>;
    Execute plan;
    Set current state at the node to q';
    end; (* end evaluate *)
 Evaluate(ROOT,$I_S$); (* S is the start symbol and ROOT is
                    the root of the parse tree *)
end.
```

2.8. General Attributed Grammar and the GHR Parser

Now we will show how to combine the tree-walk attribute evaluator proposed by Kennedy *et. al.* [KW78] (briefly described above) with the GRH parsing method so that they can run in parallel. An important thing to observe in the GHR parser is that if we predict a production with symbol X on the left hand side and certain inherited attributes T_X (a subset of I(X)), then we can predict in advance the sequence of events that will take place as we recognize the right-hand side of the production. More specifically, suppose we predict

$$X \Rightarrow X_1 \dots X_n$$

With inherited attributes T_X of X, then we can tell which synthetic attributes of X can be evaluated before we recognize any part of the right hand-side. (only those attributes which are dependent only on the T_X in $D^*(p)$). From these attributes and T_X we can tell which inherited attributes of X_1 can be evaluated; call these T_{X_1}. From T_{X_1} we can tell which synthetic attributes of X_1 will be available once we recognize X_1 completely and move the dot over the symbol X_1. We can then tell which inherited attributes of X_2 could be evaluated when we predict it, and so on.

In the tree-walk evaluator described above there are two kinds of visits to a node: an initial visit, when we visit a node for the first time, and revisits. When we completely recognize a symbol X which was predicted with inherited attributes T_X, we would have those synthetic attributes of X evaluated which would have been evaluated by an initial visit in the tree-walk evaluator with current input T_X, but they will be done in a number of steps (not by a single plan), as we recognize the right-hand side of the production. Therefore, we will have to divide the plans associated with the initial visits into a number of sub-plans; each will be executed as we recognize one more symbol from the right-hand side. No modification is required for the plans associated with revisits.

We now discuss briefly how to evaluate attributes while parsing with the GHR parser. For each production $p: X_0 \Longrightarrow X_1 \dots X_n$ and a current input set I_X such that NEXTMOVE($q_0(p), I_X$) is defined, we make following set of plans.

One plan will be executed when the production is predicted; this will compute all those synthetic attributes of X_0 which are dependent only on I_X. Further, this plan will also compute all the possible inherited attributes of X_1. We will also associate one plan with each X_i from the right-hand side; this plan will be executed as we move the dot over X_i. It will compute additional attributes of X_0 through X_i which may become ready to evaluate. It will also compute all the possible inherited attributes of X_{i+1} before predicting it.

To find out which plan is to be executed when we move the dot over a symbol, we need to know three things: the production p used, the inherited attributes of the left-hand side when the production was predicted, and the position of the dot in the right-hand side. The plan to be executed when we move the dot over X_k in a production p: $X \Longrightarrow X_1 \ldots X_n$, which was predicted with the inherited attribute I_X, will be denoted by plan<p,I_X,k> ($0 \leq k \leq n$).

Before we can describe how to modify plans associated with initial visits, we will briefly describe how the plans are built. The complete planning algorithm is given in appendix A.

A plan (as given in [KW78]) for a given state and current-input set is an instruction sequence which will accomplish all attribute evaluations in the subtree which are permitted by the dependencies constraints.

Definition : A semantic function $f_{p,a}$ is *ready to evaluate* in the current evaluation state (p,A) if $a \not\subset A$ but D_a^p is a subset of A.

($f_{p,a}$ is a semantic function associated with production p to evaluate attribute occurrence a in p. D_a^p is the set of attributes occurrences from p which are used in evaluating semantic function $f_{p,a}$).

Definition : The *yield* of a symbol X_K in the current evaluation state (p,A) is the following set of attribute occurrences

$\{b \mid b \not\subset A, b \in S(X_k)$, and for every i with an arc from i to b in IO_{X_k}, $i \in A\}$

Now we are ready to define an algorithm to construct a plan to be executed when we visit a node with an entry state (p,A). (An entry state is $(p, I_X \cup A')$ where (p,A') is the current state of the node and I_X is the current-input set).

Algorithm Make a Plan for Entry State (p,A);

```
1. Let S be an empty sequence of instructions and
   let A' = A;
2. Let current evaluation state be (p,A');
3. If some semantic function $f_{p,a}$ is ready to
   evaluate then append the instruction $f_{p,a}$ to
   S, add a to A', and go to step 2;
4. If $k^{th}$ subtree has a nonempty yield y
   (if we have more than one subtree with nonempty
   yield, then choose the left-most subtree) then
       append VISIT(k,I) to S, where
       $I = \{ a \mid a \in I(X_k) \text{ and } a \in A' \}$
       Let $A' = A' \cup y$; and go to step 2;
5. Let q be the evaluation state (p,A'). The
   resulting plan is S and nextstate is q.
```

Now we are ready to describe how to modify plans associated with initial visits. Let $X_0 \Rightarrow X_1 \ldots X_n$ be a production. Let NEXTMOVE(q_0, I_X) be $(plan_1, q_1)$ for some $I_X \in \mathbf{I}$. Remove $plan_1$ from the PLAN set and include the following set of plans.

1. Let S be an empty set of instructions and $A = I_X$;
2. If some semantic function $f_{p,a}$ is ready to evaluate then append the instruction

```
        $f_{p,a}$ to S, add a to A, and goto step 2;
3. S is the resulting plan for plan<$I_X$,p,0>
4. For i := 1 To n Do
            Begin
            4.1  Let $I_{X_i}$ be the set of inherited attributes
                 which have already been evaluated;
            4.2  For each $P_j \in P$ such that left-hand side
                 of $p_j$ is $X_i$ Do
                 If NEXTMOVE($q_0(p_j)$, $I_{X_i}$) is undefined then
                 Begin
                   1.  Add $I_{X_i}$ to I; (* include new current
                                   input to current-set *)
                   2.  Make_Plan($q_0(p_j)$, $I_{X_i}$);
                 End; (* else plan already exists *)
            4.3  Let y be the yield of $X_i$ with current-input $I_{X_i}$,
                 Let A' = A' $\cup$ y;
            4.4  Let S be an empty sequence of instructions.
            4.5  If some semantic function $f_{p,a}$ is ready to
                 evaluate then append the instruction $f_{p,a}$ to S,
                 add a to A', go to step 4.5
            4.6  If the kth (only for $k \le i$) subtree has a nonempty
                 yield y, then append VISIT(k,I) to S, where
                 $I = \{ a \mid a \in I(X_k) \text{ and } a \in A' \}$
                 Let A' = A' $\cup$ y; goto step 4.5;
            4.7  Associate resulting plan S with plan<$I_X$,p,i>;
```

For each production $p \in P$ and each current-input set I_X such that NEXTMOVE($q_0(p), I_X$) is defined, we have replaced the initial visit of the tree-walk evaluator by n+1 plans if the production p is

$$X \Rightarrow X_1 \ldots X_n$$

Now we will redefine * and x products and PREDICT function so that appropriate plans are executed as we recognize part of the right-hand side of a production.

One plan will be executed when the production is predicted. This plan will compute all those synthetic attributes of X_0 which are dependent only on I_X. It will also compute all the possible inherited attributes of X_1. We will also associate one plan with each X_i from the right-hand side. This plan will be executed as we move the dot over X_i. It will compute additional attributes of X_0 through X_i which may become ready to evaluate. It will also compute all the possible inherited attributes of X_{i+1} before predicting it. Finally, if we move the dot over a sequence of symbols because they all derive λ, we shall move the dot over one symbol at a time and execute the plan associated with that symbol before considering the next symbol.

If we have an absolutely non-circular attributed grammar, we could predict a symbol several times if it is predicted by different dotted rules. (Even if the symbol has the same inherited attributes, we predict it more than once as explained before). Therefore when we paste an attributed symbol $B{\downarrow}I_B{\uparrow}S_B$ into a dotted rule, we have to make sure that it was predicted by this dotted rule. One possible method for checking would be to maintain a back pointer to the dotted rule which predicted $B{\downarrow}I_B{\uparrow}S_B$ and paste only into that dotted rule. Even if we use the implementation suggested by Graham at. el. [GHR80], we can enforce this restriction easily by checking the back pointer after pasting and remove it if the back pointer does not match. Or we could use this back pointer in selecting the dotted rule to paste the symbol $B{\downarrow}I_B{\uparrow}S_B$.

Finally, as we move the dot over a symbol, we execute some plan as explained before, evaluating some additional attributes in the dotted rule. In all the definitions given below we shall use $\bar{I}$ to denote the attributes obtained from I after executing some plan as the dot was moved (similarly $\bar{S}$ is obtained from S after executing some plan).

Now we will define the x and * products for absolutely non-circular attributed grammars.

Definition : Let AG = (V_N, V_T, Q, S, P, F_Q) be an absolutely non-circular attributed grammar. Let Q be a set of dotted rules and let R be a set of attributed symbols with some of its attributes evaluated. Define

$Q{\times}R = \{A{\downarrow}\bar{I}_A{\uparrow}\bar{S}_A \Longrightarrow \bar{\alpha}B{\downarrow}\bar{I}_B{\uparrow}\bar{S}_B\bar{\beta}.\gamma \mid A{\downarrow}I_A{\uparrow}S_A \Longrightarrow \alpha.B{\downarrow}I_B\beta\gamma \in Q,\ \beta \overset{*}{\Longrightarrow} \lambda$, and $B{\downarrow}\bar{I}_B{\uparrow}\bar{S}_B \in R$. Attributes of B are also included with the symbol. Let $|\alpha B|$ be k ($|\alpha|$ represents the length of α), $A \Longrightarrow \alpha B\beta\gamma$ be production p in P, and $|\beta|$ be i.

```
Execute plan<p, $I_A$, k>
For j := 1 To i Do
    execute plan<p, $I_A$, k+j>;  }
```

$Q{*}R = \{A{\downarrow}\bar{I}_A{\uparrow}\bar{S}_A \Longrightarrow \bar{\alpha}B{\downarrow}\bar{I}_B{\uparrow}\bar{S}_B\bar{\beta}.\gamma \mid A{\downarrow}I_A{\uparrow}S_A \Longrightarrow \alpha.B{\downarrow}I_B\beta\gamma \in Q,\ \beta \overset{*}{\Longrightarrow} \lambda$, and $B{\downarrow}\bar{I}_B{\uparrow}\bar{S}_B \overset{*}{\Longrightarrow} C{\downarrow}I_C{\uparrow}S_C$ for some $C{\downarrow}I_C{\uparrow}S_C \in R$. Evaluate the attributes of B before pasting it. Let $|\alpha B|$ be k, $A \Longrightarrow \alpha B\beta\gamma$ be production p in P, and Let $|\beta|$ be i.

```
Execute plan<p, $l_A$, k>
For j := 1 To i Do
    execute plan<p, $l_A$, k+j>;   }
```

Note that there are two alternatives for computing attributes of B from the attributes of C if $B \Rightarrow^* C$, we can either make a plan during the planning phase which gives a sequence of instruction to execute to get these attributes, or we can implement x and * products such that the parsing algorithm does not have to know about the chain rules and the symbols deriving λ, computing them at parse time while building the parse matrix. For more details and other variations for implementing x and * products see [GHR80]. From our implementation experience, we find the second alternative to be very easy to implement and the execution speed to be quite satisfactory. Furthermore, this technique can deal with the predicates without any undue complication (we will discuss predicates in more detail in the section 2.9).

Let Q,R be sets of dotted rules then define x and * products as follows

QxR = $\{A{\downarrow}\bar{I}_A{\uparrow}\bar{S}_A \Rightarrow \bar{\alpha}B{\downarrow}\bar{I}_B{\uparrow}\bar{S}_B\bar{\beta}.\gamma \mid A{\downarrow}I_A{\uparrow}S_A \Rightarrow \alpha.B{\downarrow}I_B\beta\gamma \in Q,\ \beta \Rightarrow^* \lambda$, and $B{\downarrow}\bar{I}_B{\uparrow}\bar{S}_B \Rightarrow \delta. \in R$. Let $B \Rightarrow \delta$ be production p_1 in P and $|\delta|$ be n_1, $A \Rightarrow \alpha B\beta\gamma$ be production p in P, $|\alpha B|$ be k, and $|\beta|$ be i.

```
Execute plan<$p_1$, $l_B$, $n_1$>;
For j := 0 To i Do
    execute plan<p, $l_A$, k+j>;   }
```

Q*R = $\{A{\downarrow}\bar{I}_A{\uparrow}\bar{S}_A \Rightarrow \bar{\alpha}B{\downarrow}\bar{I}_B{\uparrow}\bar{S}_B\bar{\beta}.\gamma \mid A{\downarrow}I_A{\uparrow}S_A \Rightarrow \alpha.B{\downarrow}I_B\beta\gamma \in Q,\ \beta \Rightarrow^* \lambda$, and $B{\downarrow}\bar{I}_B{\uparrow}\bar{S}_B \Rightarrow^* C{\downarrow}I_C{\uparrow}S_C$ for some $C{\downarrow}I_C{\uparrow}S_C \Rightarrow \delta. \in R$. Evaluate the attributes of B before pasting it. Let $C \Rightarrow \delta$ be production p_1 in P, $|\delta|$ be n_1, $A \Rightarrow \alpha B \beta \gamma$ be production p in P, $|\alpha B|$ be k, and $|\beta|$ be i.

```
Execute plan<$p_1$, $I_C$, $n_1$>;
For j := 0 To i Do
    execute plan<p, $I_A$, k+j>;   }
```

Now we will define the PREDICT function so that it executes the appropriate plans before predicting a production.

Definition : Let AG = (V_N, V_T, Q, S, P, F_Q) be an absolutely non-circular attributed grammar and let R be a set of attributed symbols with some of the inherited attributes evaluated. Define

PREDICT(R) = $\{C{\downarrow}I_C{\uparrow}S_C \Rightarrow \alpha.\beta \mid C \Rightarrow \alpha\beta \in P, \alpha \overset{*}{\Rightarrow} \lambda$, and $B \overset{*}{\Rightarrow} C\rho$ for some $B{\downarrow}I_B \in R$ and some $\rho \in V^*$,

1. If we predict $B{\downarrow}I_B{\uparrow}S_B \Rightarrow .C{\downarrow}I_C{\uparrow}S_C\rho$ then we execute plan<p_i,I_B,0> (where $B \Rightarrow C\rho$ is production p_i in P) before we include any production with C on the left-hand side (this way we shall get all the possible inherited attributes of C before we predict it), and
2. If we predict a production $C{\downarrow}I_C{\uparrow}S_C \Rightarrow \alpha.\beta$, we execute plan<$p_j$,$I_C$,0> (where $C \Rightarrow \alpha\beta$ is production p_j in P), and
3. If we have predicted $C{\downarrow}I_C{\uparrow}S_C \Rightarrow \alpha.\beta$, and $|\alpha|$ = k then,

```
    For i := 1 To k Do
        execute plan<$p_j$,$I_C$,k>;}
```

If R is a set of dotted rules then define

PREDICT(R) = PREDICT($\{B{\downarrow}I_B \mid A{\downarrow}I_A{\uparrow}S_A \Rightarrow \alpha.B{\downarrow}I_B\beta$ is in R$\}$

2.9. A Few Missing Details

We still have to workout some details. Normally the productions may contain some action symbols. These action symbols can have side effects. Since we cannot say for sure whether a production will be used or not in the final parse, such side effects have to be disallowed.

2.9.1. Predicates in S-attributed and L-attributed Grammars

Attribute grammars also allow predicates that control production application. These can be included in our method without much problem. Assume P is a predicate symbol and f_p is a predicate function associated with the P, further

$P{\downarrow}a_1{\downarrow}a_2 \dots {\downarrow}a_n \Longrightarrow \lambda$ iff
$f_p(a_1, a_2, \dots, a_n) \rightarrow \text{True}$
then we insert dotted rule
$P{\downarrow}a_1{\downarrow}a_2 \dots {\downarrow}a_n \Longrightarrow \lambda.$ iff
$f_p(a_1, a_2, \dots, a_n) \rightarrow \text{True}$

This restrictions can be enforced if we treat predicate symbols differently then the other nonterminals in the grammar. Since all the predicates symbols derive λ, we can include this modification easily in our parsing algorithm. When we move the dot over a symbol because it drives a λ, we check if the last symbol used in the derivation was a predicate symbol, if so, then we call the corresponding predicate function and move the dot over the symbol if and only if the predicate is *true*. If a predicate is not *true*, that production will never be recognized by the parser because the dot will never move past the predicate symbol $P{\downarrow}a_1$ ...

$\downarrow a_n$, which evaluates to *false*.

One advantage in the GHR parsing method over the CYK is that it tries to eliminate unnecessary dotted rules from the parsing matrix. It does this by using the PREDICT function to delete all dotted rules except those that can follow the parsed left context.

We can further reduce the number of dotted rules from the parse matrix by deleting a dotted rule when some of the predicates associated with it return *false*. This way we can eliminate unnecessary dotted rules when we realize that a production cannot be used in the final derivation.

Suppose we have a dotted rule $X \Longrightarrow \alpha X_{i-1}.X_i\beta$ and suppose that $X_i \overset{*}{\Longrightarrow} \lambda$. If the last symbol used in the derivation is $q_j \in Q$ (set of predicate symbols), then before we move the dot over X_i, we evaluate the primitive predicate f_{q_j}. If f_{q_j} evaluates to false then we do not move the dot over the symbol X_i and eliminate the dotted rule $X \Longrightarrow \alpha X_{i-1}.X_i\beta$ from the parse matrix, as this will never be recognized completely.

The amount of savings obtained by the above method will not be significant unless we eliminate all the dotted rules used in recognizing αX_{i-1}. All these dotted rules can be deleted by making a depth-first left-to-right visit to all the symbols used in recognizing this dotted rule.

But we need to consider one problem in this method. A symbol could be used by more than one dotted rule. We can solve this problem if we keep a reference count with each symbol. We start with reference count equal to zero. Every time we paste a symbol into a dotted rule we increment the reference count of that symbol by one so that the reference count tells us how many dotted rules are predicting a symbol. When we delete a dotted rule, we visit all subnodes immediately below it (in any order, say left-to-right) and decrement the reference count by one. If it becomes zero then we delete this symbol and recursively visit all the subnodes immediately below it. After subtracting one, if the reference count is greater than zero then we just return. (We need not visit the subtree below this symbol as this subtree is being used by some other dotted rule).

To achieve even greater saving we can write each contextual predicate in conjunctive normal form and break it into subpredicates, which can be distributed across the right-hand side of the production such that we evaluate a subpredicate as soon as its inherited attributes become available. As we recognize part of the right-hand side of a production, some attribute values will become available allowing evaluation of some of the subpredicates. By distributing these subpredicates across the right-hand side of the production we can insure that a subpredicate will be evaluated as soon as possible. Useless dotted rules will be eliminated as soon as possible from the parsing matrix. Context sensitive errors are also detected as soon as possible, giving more flexibility to error corrector. We shall see in the next chapter that it is very important to detect an error as soon as possible if we hope to make good error correction.

For a context-free grammar $G = (V_n, V_t, P, S)$, the standard LL(k) parsing function is a mapping

$$M : (V_n \cup V_t) \times V_t^{*k} \rightarrow \{\textit{predict } i, \textit{pop}, \textit{error}, \textit{accept}\}$$

That is, the parser uses the top of the stack symbol $A \in V_n \cup V_t$, and the k-symbol lookahead, $u \in V_t^{*k}$, to determine the next parse move.

Milton and Fischer [MF79] have suggested using evaluated attributes of A (that is, all the inherited attributes of A for an L-attributed grammar) and attributes of u in the parsing function. This is done by associating a predicate (termed as disambiguating predicate) with each production. More specifically, disambiguating predicate is a mapping

$$DP : X{\downarrow}I_X \times a_1{\uparrow}S_{a_1} \ldots a_k{\uparrow}S_{a_k} \rightarrow \{\mathit{true},\mathit{false}\}$$

Before predicting a production, the associated disambiguating predicate is evaluated and the production is predicted only if the associated disambiguating predicate evaluates to *true*.

Our algorithm can support these predicates as mentioned before. If there are some disambiguating predicates, then before we predict a production using the PREDICT function, we evaluate the associated contextual predicate, and predict only those productions whose associated predicates evaluate to *true*. This technique will work well when we have an L-attributed grammar and will further reduce the number of unnecessary dotted rules. By including disambiguating predicates in our implementation, we could save more than 30% of the execution time over the implementation which did not use such predicates. We also saved about 30% of the memory space requirement for storing the parse matrix (as we had to store fewer dotted rules in each parse matrix element). In fact in our implementation we went one step further and for each dotted rule we associated a follow set. This follow set was computed using the head grammar (hence ignoring all the context sensitive errors). While taking x or * product we check the lookahead symbol and include a dotted only if the lookahead symbol is in the follow set of the dotted rule.

2.9.2. Predicates in Absolutely Non-Circular Attributed Grammars

If we are parsing an absolutely non-circular attributed grammar, then we may not have all the attributes available when we move the dot over a predicate symbol, therefore, we will use following extended definition of predicates.

$f(a_1, \dots ,a_n) \rightarrow$ True if for each attribute position which contains 'unknown', there exist one value from the corresponding attribute domain such that

$f(\bar{a}_1 , \bar{a}_2 , \dots , \bar{a}_n) \rightarrow$ True
Where $\bar{a}_i = a_i$ if $a_i \neq$ 'unknown'
otherwise $\bar{a}_i$ = some value from D_i (Attribute domain)

(It may not always be possible to compute f with some values 'unknown'. In such a case, we might have to say that if there is no conflict among the values known then the function evaluates to true)

We will not move the dot over a predicate symbol if the associated predicate function evaluates to *false* but delete that dotted rule from the parse matrix. We also delete all the symbols used by this dotted rule using the method described above.

But we also have to consider the situation where predicate becomes *false* some time after we move the dot over it. When more attributes become available for a predicate symbol we reevaluate the predicate function and that time it could evaluate to *false*. In this situation unnecessary dotted rules can be detected as follows.

When we move the dot over a symbol we execute some plan to evaluate additional attributes (as explained before). Before we start executing this plan we set a global variable 'PREDICATE' to *true*. While executing the plan if we visit a predicate symbol with additional attributes, we reevaluate the associated predicate function and if it evaluates to false; we set the global variable 'PREDICATE' to *false*. This way after executing a plan, the global variable 'PREDICATE' will be *false* iff some predicate in the subtree has become *false* and, can be deleted from the parse matrix. The method described above can then be used to delete all the symbols used by this dotted rule.

2.10. Implementation Results

The above described planning and attribute evaluation algorithms were implemented in pascal on a VAX/780 computer. Implementation results are quite encouraging. Planning program is about 5000 lines of code and the execution speed is quite acceptable. In our test cases we had several small grammars and two grammars each having approximately 40 productions (one of these tow grammars is given in the next chapter section 3.6). Our planning algorithm took about 9 seconds of cpu time for the grammar with 40 productions. In addition to creating various table files, the program produces a formatted listing of the full grammar, goto table, and complete plan set.

Attribute evaluation is also reasonably fast. In our test cases we had a grammar with about 40 productions. Parsing time was about 80 msec for 5 symbols and 2.3 seconds for 50 symbol. We did not try to parse more than 50 symbols since we feel that this algorithm will be most useful for parsing small inputs. Because of the quadratic time of the parsing algorithm, it may not very useful to parse very large inputs using the GHR parser. For some practical grammars if we use disambiguating predicates and lookahead sets, we may see very close to linear behavior. Furthermore, as we shall see in the next section, there are several applications where we want to parse fairly small inputs (e.g. for parsing a small right context past the error token in an error correction, or code generation where we want to parse a number of small expression trees) and need the generality of the above described algorithm.

2.11. Conclusion

The attribute evaluator described above tries to evaluate as many attributes as possible at any given time and avoids any unnecessary visit to a subtree. Therefore, the algorithm is linear in number of nodes, excluding attribute evaluation time. Informally, we can prove this bound by observing that each symbol (hence node) has a constant number of attributes. Each time we visit a node we evaluate at least one more attribute. Hence time spent visiting a node is bounded by a constant (the number of attributes of that symbol). The algorithm is quite general and can be used with any absolutely non-circular attributed grammar.

Attributed grammars have been suggested for automating compiler construction. There are very few error correcting algorithms which can use attributes effectively for finding a good context-sensitive error correction. Our principle motivation was to develop an efficient attribute evaluation technique in the GHR parser to apply it to context-sensitive error correction. Using the above described attribute evaluation method we have developed a high quality error correcting algorithm. We use the above described attribute evaluation algorithm to parse a small context past the error token for making context sensitive error correction.

This error correction algorithm is described in detail in the chapter 3.

Glanville [GG78] has suggested using context-free grammars for a machine description in the development of universal compiler system. Ganapathi [GA80] has suggested attributed context-free grammars for a machine description for generating a retargetable optimizer. Because the parsing method used in [GA80] is LR-based, inherited attributes are not allowed. But by using the above algorithm, we can include inherited attributes as well. Making it possible to describe some fairly complex instructions. This method can also yield better optimization. We can fix some value n as the window size of a peephole optimizer. We can alter the window of peephole optimizer by changing the value of n, the number of symbols to be parsed by the GHR algorithm. While still preserving the linearity of the code generation algorithm.

This algorithm should be easier to use than the one given in [GA80]. The user can simply associate a time and space cost with each machine instruction. Currently a user has to put contextual predicates in the proper places such that the predicates associated with the cheaper instruction are evaluated before the predicates associated with more expensive instruction. As we parse and paste instructions together we can add the time and space costs for these instructions. At the end of the parsing we can pick the one which is optimum in time or space, or any combination that may be desirable. This application will require some further investigation but method seems promising.

Another significant use of this attribute evaluation algorithm could be in experimental programming languages where the grammar is not fully known and needs to be changed for experimentation. Even if we have the complete grammar, it may not be in a suitable S-attributed or L-attributed form. The above described algorithm can free the user from all concern about the form of the grammar and can also allow the user to make modifications.

Chapter 3

Error Correction

3.1. Programming Errors

As explained in the chapter 1, error correction is an important aspect of compiler design. A substantial portion of a programmer's time is spent in correcting errors. Errors that can be detected at compile time can be divided into three categories. 1) Lexical errors: those that are detected by the scanner; 2) Syntax errors: those that are detected by the parser (syntax recognizer); 3) Semantic errors: those that are detected by the "semantic" phase of the compiler. In this chapter, the problem of automatic correction of latter two kinds of errors (syntax and semantic) is discussed. Our error-correction algorithm was developed with the following objectives in mind.

1: An Error free program should entail little or no error-correction overhead.

2: The algorithm should be able to handle all the possible error situations.

3: The correction selected should be regionally optimal [MA82] based on some cost function.

4: The correction made should not cause other error messages.

The first three objectives can be realized without much difficulty. To minimize the cascading of error messages we need a global error correction algorithm. Our algorithm could be used for global error correction, but it would be very expensive to do so. Even without global error correction, however, using our error correction algorithm we can realize the first three objectives and reduce the number of cascaded error messages.

3.2. Assumptions

Before we present our error correction algorithm, we will state all the assumptions we make about the grammar and the parsing algorithm used for parsing programs.

We are assuming that we have an L-attributed grammar AG = (V_t, V_n, P, Q, S, F_Q) (as defined in the chapter 2) and we are using the LL(1) parsing method. We also put some restrictions on the LL(1) parsing algorithm.

Since our error-correction algorithm does not attempt to change the parsed left context (we feel that changing already parsed left context unduly complicates the corrector and does not appear necessary to obtain useful corrections), we require that the parser has *correct prefix property*. That is, the sequence of symbols to the left of the erroneous symbol is always a prefix of some $\overline{\omega} \in L(G)$ [FMQ80]. This restriction insures that symbols that have been accepted by the parser can be considered correct.

Some LL(1) parsing methods do not detect an error upon first encounter of an erroneous symbol, allowing several parse moves before the error is detected, and leaving very little useful information on the parse stack to guide an error-correction algorithm. We require a parsing algorithm which will never make a transition on an erroneous input symbol, thus detecting an error when the error symbol is first encountered. This property is termed as the *Immediate Error Detection Property* (IEDP) [FMQ80]. It insures that all the parsing (syntax) errors are detected when an error symbol appears in the lookahead for the first time.

To make sure that we have the IEDP for semantic errors as well, we relax rules for L-attributes grammars slightly. We let predicate functions use the attributes of the symbol following it. That is, inherited attributes of a predicate symbol can be dependent on the attributes of the symbol following it. We also assume that the predicates are placed in the productions such that semantic errors are detected as soon as possible. Informally, an attribute parser has the correct prefix property if and only if any attributed symbol in AV that can ever be predicted can derive an attributed terminal string. A procedure to decide if the correct prefix property holds for a given attributed grammar has been suggested by Dion [DI79]. We shall illustrate below how to place predicates such that the correct prefix property is maintained. To insure that the final grammar does have the correct prefix property, we have to run the grammar through the algorithm suggested by Dion [DI79].

We also assume that all the attribute domains are *finite*. Later we shall explain informally how to deal with infinite attribute domains in some limited situations (e.g. to deal with symbol tables). Finally, we assume three cost functions IC and DC, giving the cost of inserting and deleting, respectively, an attributed terminal symbol, and a replacement function which gives the cost of replacing an attributed terminal symbol by another attributed terminal symbol.

3.3. Error Correction Strategy

We propose the following error correction strategy. We use context past the point of error in choosing an error correction. When an error is detected, we scan ahead, suppressing any listing. We employ two levels of corrections. The most common changes termed as "first level correction", are one symbol corrections (one symbol insertion before the error token, deletion of the error token, or replacement of the error token by any symbol a $\in AV_t$) validated over the next k input symbols. We try first-level correction first. If we cannot find a suitable one-symbol correction (because there are additional errors around the point of error or because we need more than a one-symbol correction to fix the current error), then we try a more elaborate "second level" error correction.

Second level correction is very powerful and elaborate; it tries to transform the next k input symbols (where the value of k is determined using criteria explained later) into a string ω such that ω can legally (semantically as well as syntactically) follow the parsed left context, and such taht the cost of transforming the next k input symbols into ω is minimal in the sense that there is no other string $\overline{\omega}$ that can legally follow the parsed left context with a transformation cost less than the cost of ω.

3.4. First Level Error Correction

In first level correction, we try all the possible one-symbol insertions before the error symbol, replacing the error symbol by another attributed symbol, and deletion of the error symbol. To see which one-symbol correction is most suitable in this situation. we use context past the point of error. We scan ahead, suppressing any listing, reading the next k symbols $u_1 \ldots u_k$ from the input. We validate each one-symbol correction over the next k input symbols.

When an error is detected, suppose the parsing stack contains $X_n \ldots X_1$ (with X_n on the top). For simplicity, we will associate attributes of a symbol with the symbol. Therefore, the parsing stack contains

$$X_n \downarrow a_1^n \ \ldots \downarrow a_{N_n}^n \ \uparrow b_1^n \ \ldots \uparrow b_{M_n}^n \ \ldots X_1 \downarrow a_1^1 \ \ldots \downarrow a_{N_1}^1 \ \uparrow b_1^1 \ \ldots \uparrow b_{M_1}^1$$

If X_n is a nonterminal, we know the attribute values for a_1^n , ... , $a_{N_n}^n$; other values may not be known at this moment (If X_n is a terminal then, of course, it will not have any inherited attributes). Let u_1 be the error symbol, and let Σ be the input alphabet (attributed terminal symbols of the grammar). Now we shall build a (k+2)x(k+2) upper triangular matrix t as follows.
In the element $t_{0,0}$ we shall put

$$\text{PREDICT}(\{S \Rightarrow .X_n \downarrow a_1^n \ldots \downarrow a_{N_n}^n \uparrow b_1^n \ldots \uparrow b_{M_n}^n \ldots X_1 \downarrow a_1^1 \ldots \downarrow a_{N_1}^1 \uparrow b_1^1 \ldots \uparrow b_{M_1}^1\})$$
$$\cup \{S \Rightarrow .X_n \downarrow a_1^n \ldots \downarrow a_{N_n}^n \uparrow b_1^n \ldots \uparrow b_{M_n}^n \ldots X_1 \downarrow a_1^1 \ldots \downarrow a_{N_1}^1 \uparrow b_1^1 \ldots \uparrow b_{M_1}^1\};$$

(PREDICT is the same as defined in the GHR parsing algorithm except that now it will pass along the inherited attributes of the symbol being predicted and further, for all the λ productions, it will make sure that the corresponding predicate is true.)

The above PREDICT will put into $t_{0,0}$ all the dotted rules which can follow the parsed left context. Now we shall try to parse the string $(\{\Sigma - u_1\} \cup \{\lambda\})(\{u_1 \cup \{\lambda\})u_2 \ldots u_k$.
$(\{\Sigma - u_1\} \cup \{\lambda\})$ represents any attributed symbol from the input alphabet (AV_t) except u_1 (the error symbol), and λ (the empty string). Since Σ represents any attributed symbol, each terminal symbol will be included several times with all the possible attribute values. (Since all the attribute domains are finite, this set will be a finite set).

(λ in first and second columns give deletion of the error symbol, Σ in first column and λ in second column give replacement, and insertion is obtained by Σ in first column and u_1 in second column).

To parse the above string we shall change the GHR algorithm slightly. Instead of parsing a sequence of symbols, we shall parse a sequence of sets. This modification is very easy to make in the GHR algorithm without any significant increase in the execution time. In the second level error correction we shall find it to be very useful. Hence we shall be parsing $S_1 \ldots S_{k+1}$, a sequence of sets where

$$S_1 = \{\Sigma - u_1\} \cup \{\lambda\}$$
$$S_2 = \{u_1\} \cup \{\lambda\}$$
$$S_j = \{u_{j-1}\} \quad \text{for } 3 \le j \le k+1$$

We have to modify the * product used by the GHR parser as follows :

Let Q be a set of dotted rules, then $Q * \{\lambda\} = Q$

The following algorithm will try all the possible one symbol insertions before the error symbol, deletion of the error symbol, and replacement of the error symbol by all the symbols from AV_t acceptable in the parsed left context. We validate each one-symbol correction over the next k input symbols by parsing the all the corrections and the next k input symbols. After parsing the next k input symbols, if we cannot select a unique insertion, deletion, or replacement, then we make the choice using the user defined insertion, deletion, and replacement cost functions, and select a correction which has least cost.

We therefore, get the following parsing algorithm.

```
begin
$t_{0,0}$ := PREDICT({S $\Longrightarrow$
        $.X_n \downarrow a^n_1 \dots \downarrow a^n_{N_n} \;\uparrow b^n_1 \dots \uparrow b^n_{M_n} \dots X_1 \downarrow a^1_1 \dots \downarrow a^1_{N_1} \;\uparrow b^1_1 \dots \uparrow b^1_{M_1}$ })

  $\cup$ {S $\Longrightarrow .X_n \downarrow a^n_1 \dots \downarrow a^n_{N_n} \;\uparrow b^n_1 \dots \uparrow b^n_{M_n} \dots X_1 \downarrow a^1_1 \dots \downarrow a^1_{N_1} \;\uparrow b^1_1 \dots \uparrow b^1_{M_1}$ };

(* Build column 1 thru k+1 *)
for j := 1 to k+1 do
 (* Using column 0...j-1 build column j *)
 begin
  $t_{j-1,j} := t_{j-1,j-1} * S_j$;

  for i := j-2 downto 0  do
   begin
   r := ( $\bigcup_{i<k<j-1} (t_{i,k} \times t_{k,j})) \cup (t_{i,j-1} \times (t_{j-1,j} \cup S_j))$;

   $t_{i,j} := r \cup t_{i,i} * r$;

   end;
  $t_{j,j}$ := PREDICT( $\bigcup_{0 \le k < j} t_{i,j}$);

      (* While taking the x or * product, we have to compute attri-
      butes of the symbol over which the dot is being moved. Since all
      the information is available to do so (because the grammar is L-
      attributed), it will not be a problem. When we include a nonter-
      minal as an argument to the PREDICT function, we include its
      inherited attributes. Hence a nonterminal could be included more
      than once with different inherited attributes. *)

 end;
if $\exists$ i such that $t_{i,k+1} \neq \Phi$ then

   (* we have found at least one correction *)
     Let $c_1,\dots,c_n$ be all the possible legal corrections found.  Using

     the insertion, deletion, and replacement cost functions defined

     by the user select the one that costs least.

 else
   Call Level-2 corrector
end;
```

3.5. Second Level Error Correction

The second level error corrector is invoked when we cannot correct an error by a single insertion, deletion, or replacement. (Even if it is not strictly needed, the second level corrector might be called to find a cheaper correction). In this algorithm we shall transform the next k input symbols into a string ω such that ω can legally (both semantically as well as syntactically) follow the parsed left-context. For this transformation also we shall use the GHR parsing algorithm. This transformation is similar to the Aho and Peterson algorithm, but we shall not use error productions for obtaining insertions, deletions, and replacements, as done by the Aho and Peterson algorithm. Rather we shall simulate insertion by moving the dot over the symbol we want to insert without consuming any input symbol, we simulate deletion by consuming the input symbol without moving the dot, and replacement by moving the dot over a symbol b ($b \in AV_t$) when the actual input is a ($a \neq b$). That is, our algorithm is very similar to the algorithm proposed by Mauney [MA82], except that while parsing the right context we shall use the semantic information and evaluate the attributes while parsing using the algorithm outlined in the previous chapter.

When we move the dot over a symbol, if some of its attributes cannot be evaluated, we put the 'unknown' value in those attribute positions. For example, we insert a nonterminal symbol by moving the dot over it. In such a situation we try to get as many inherited attributes as possible from the left context. If some are not available then we put the 'unknown' value in that position. Most of the synthetic attributes will not be available and again we shall put the 'unknown' value in those positions.

Now we define various cost functions used by the second level error corrector. Let S be insertion cost function defined as follows

$S(a{\uparrow}S_a) = IC(a{\uparrow}S_a)$ if $a{\uparrow}S_a \in AV_t$

(where IC is a user-defined insertion cost function.)

$S(y) = IC(a_1{\uparrow}S_{a_1}) + \dots + IC(a_n{\uparrow}S_{a_n})$

if $y \in AV_t^*$ and

$y = a_1{\uparrow}S_{a_1} \dots a_n{\uparrow}S_{a_n}$

$S(A{\downarrow}I_A{\uparrow}S_A) = \text{Min}\ \{\ IC(y)\ |\ A{\downarrow}I_A{\uparrow}S_A \Rightarrow^* y$

for some $y \in AV_t^*\ \}$

if $A{\downarrow}I_A{\uparrow}S_A \in AV_n$

(AV_t is the set of attributed terminal symbols and

AV_n is the set of attributed nonterminal symbols)

An algorithm for computing the S table for a given attribute grammar is given in appendix B.

SM defines the minimum cost of insertion for a nonterminal symbol over all its attribute combinations, as follows

$$SM(a) = \underset{S_a \in D_a^S}{\text{Min}} \{IC(a{\uparrow}S_a)\} \text{ if } a \in V_t$$

$$SM(A) = \underset{\substack{I_A \in D_A^I \\ S_A \in D_A^S}}{\text{Min}} \{S(A{\downarrow}I_A{\uparrow}S_A)\} \text{ if } A \in V_n$$

The SM table gives minimum cost of insertion of a nonterminal A over all possible attribute values.

Function C is defined over a string of symbols. It gives the minimum cost of insertion of the given string over all possible attribute values, as follows

$$C(\beta) = C(B_1 \dots B_n) = SM(B_1) + \dots + SM(B_n)$$

These cost functions (SM and C) are used to get an approximate cost of correction. All the potential context sensitive errors are checked by the parsing algorithm. When we insert a symbol (terminal or nonterminal) we may not have all its attributes available, so it is very difficult to find out actual cost of insertion. We therefore use the above defined cost functions (SM and C) to get an approximate cost of correction. After parsing the next k input symbols we shall try to fully evaluate each correction tree and use the S table to get the exact cost of correction (These functions, SM and C, are used by the * and x products).

In FMQ [FMQ80] and other related error correction algorithms [DI79, FMD79, FMM79] the E table is used to generate a valid prefix of a given terminal from a given nonterminal. More specifically

$$E(A,a) = \text{Min} \{ S(\alpha) \mid A \overset{*}{\Longrightarrow} \alpha a \beta \}$$

To achieve the same effect we modify the GHR's PREDICT function as follows.

Let R be a set of symbols then

$$\text{PREDICT}(R) = \{ C \Longrightarrow \alpha.\beta \mid C \Longrightarrow \alpha\beta \in P, \text{ and } B \overset{*}{\Longrightarrow} \omega C \gamma \text{ for some } B \in R \text{ and } \omega, \gamma \in V^* \}$$

If R is a set of dotted rules then

$$\mathrm{PREDICT}(R) = \mathrm{PREDICT}(\{B \mid A \Longrightarrow \alpha.B\gamma \in R\})$$

(Which is same as defined in the GHR parsing algorithm)

That is, if we have a nonterminal symbol right of the dot, we predict all the symbols which are reachable (directly or indirectly) from that symbol ($B \overset{*}{\Longrightarrow} \omega C\gamma$ forces that). This way if we have $B \overset{*}{\Longrightarrow} \ldots a \ldots$ for some nonterminal symbol B, the nonterminal symbol which directly derives 'a' will also be predicted. If we have $B \overset{*}{\Longrightarrow} \rho a\ldots$ and we want to match the 'a' from the input then we should insert ρ. The above defined PREDICT does so by including $C \Longrightarrow \alpha.\beta$ for all the possible value of α whenever we predict a nonterminal symbol C. Therefore, as much string as required will be inserted before matching a symbol from the input.

DC is the deletion cost function defined as follows.

$$DC(a{\uparrow}S_a) = DELETE(a{\uparrow}S_a) \text{ if } a{\uparrow}S_a \in AV_t$$

(where DELETE is a user defined deletion cost function.)

$$DC(y) = DC(a_1{\uparrow}S_{a_1}) + \ldots + DC(a_n{\uparrow}S_{a_n})$$

if $y \in AV_t^*$ and

$$y = a_1{\uparrow}S_{a_1} \ldots a_n{\uparrow}S_{a_n}$$

$$DC(A{\downarrow}I_A{\uparrow}S_A) = \mathrm{Min}\,\{\, DC(y) \mid A{\downarrow}I_A{\uparrow}S_A \overset{*}{\Longrightarrow} y \text{ for some } y \in AV_t^* \,\}$$

if $A{\downarrow}I_A{\uparrow}S_A \in AV_n$

R is a user-defined replacement cost function. That is $R(a{\uparrow}S_a\,,\, b{\uparrow}S_b)$ gives the cost of replacing $a{\uparrow}S_a$ by $b{\uparrow}S_b$. In order to insure that we always select the cheapest correction, we require that directly replacing $a{\uparrow}S_a$ by $b{\uparrow}S_b$ is the cheapest way to obtain $b{\uparrow}S_b$ from $a{\uparrow}S_a$. That is

$$R(a{\uparrow}S_a\,,\, b{\uparrow}S_b) \le DC(a{\uparrow}S_a) + IC(b{\uparrow}S_b) \text{ and}$$

$$R(a{\uparrow}S_a\,,\, b{\uparrow}S_b) \le R(a{\uparrow}S_a\,,\, c{\uparrow}S_c) + R(c{\uparrow}S_c\,,\, b{\uparrow}S_b) \qquad \forall \quad c{\uparrow}S_c \in AV_t$$

If an error is detected and it cannot be corrected by a single insertion, deletion, or replacement then we create k sets $S_1 \ldots S_k$ from the next k input symbols $u_1 \ldots u_k$. Elements of these sets are pairs <a,c>, where $a \in \Sigma \cup \{ \lambda \}$ and c is an integer giving the cost associated with that symbol. $S_1 \ldots S_k$ are obtained as follows

$$S_i = \{<a,c> \mid R(u_i,a) = c \} \cup \{<\lambda\,, DC(u_i)> \}$$

(* Assume that $R(u_i,u_i) = 0$ for all i's *)

These sets will take care of deletion and replacements; insertions are obtained by moving the dot over the symbol we want to insert. For example, if we have a dotted rule $A \Longrightarrow \alpha.B\beta$ in column j and we want to insert a symbol B (B could be a terminal or a nonterminal symbol) before the next input symbol a_{j+1}, we can include the dotted rule $A \Longrightarrow \alpha B.\beta$ in column j without consuming any input. However when we insert a symbol, we may not be able to evaluate all of its attributes. When we insert a symbol, we try go to evaluate as many attributes as possible and for all those attributes which cannot be evaluated we place the value 'unknown' in those attribute positions. After parsing the next k input symbols we shall come back and fix all these attribute values as explained later.

Using these cost functions we now redefine the x and the * products used by the GHR algorithm.

In the definitions given below we are assuming that some attribute positions will have the 'unknown' value if we could not determine a unique value for these positions. Therefore, we use the extended definition of the predicate functions. To be more specific:

When we call a predicate function with some attribute values 'unknown' then it will return true if for each attribute place which contains the value 'unknown', there exist one value from the corresponding attribute domain such that the predicate is true. (This definition is the same as the one given in chapter 2). Moreover if the predicate is *true* for exactly one value of an attribute position which contains the value 'unknown', then we put that unique value there. For example

> $f_p(q,s) \rightarrow$ *true* iff $q = s$ and the domain of $s = \{x, y, z\}$
>
> When we call the predicate f_p with $q = x$ and $s =$ 'unknown'; the f_p will return *true* and assign value x to s.

It may not be possible to assign a unique value to an attribute position which contains the value 'unknown', but we try to replace as many of the attribute places as possible for efficiency. The algorithm will work correctly even if we do not change any 'unknown' value at this stage.

As we parse some of the 'unknown' values will be replaced by some legal value from the corresponding domains. $\bar{T}$ is obtained after evaluating some predicates using input I. In case of synthetic attributes, when we predict a symbol, its synthetic attributes are not known. After matching a symbol from the input we should be able to evaluate some of its synthetic attributes (We may not be able to evaluate all the synthetic attributes because some inherited attributes could have the 'unknown' value). Those attributes values are reflected in $\bar{S}$. All the attribute places will be empty in S whereas in $\bar{S}$ we shall have most attribute values fixed.

Let Q be a set of dotted rules and R be a set of pairs <a,c> where a $\in$ AV $\cup$ $\{ \lambda \}$ (we have pairs involving terminals as well as nonterminals) and c is an integer, then

$$Q\times R = \{ A{\downarrow}\bar{I}_A{\uparrow}\bar{S}_A \Rightarrow \alpha B{\downarrow}\bar{I}_B{\uparrow}\bar{S}_B\bar{\beta}.\gamma \ ; \ c \ | \ A{\downarrow}I_A{\uparrow}S_A \Rightarrow \alpha.B{\downarrow}I_B{\uparrow}S_B\beta\gamma \ ; \ c_1 \in Q,$$
$$B{\downarrow}\bar{I}_B{\uparrow}\bar{S}_B \ ; \ c_2 \in R, \text{ and } c = c_1 + c_2 + C(\beta) \}$$

Where α, β, and $\gamma \in AV^*$ (attributed symbols).

$$Q^*R = \{ A{\downarrow}\bar{I}_A{\uparrow}\bar{S}_A \Rightarrow \alpha B{\downarrow}\bar{I}_B{\uparrow}\bar{S}_B\bar{\beta}.\gamma \ ; \ c \ | \ A{\downarrow}I_A{\uparrow}S_A \Rightarrow \alpha.B{\downarrow}I_B{\uparrow}S_B\beta\gamma \ ; \ c_1 \in Q,$$
$$B{\downarrow}\bar{I}_B{\uparrow}\bar{S}_B \Rightarrow^* D{\downarrow}\bar{I}_D{\uparrow}\bar{S}_D, D{\downarrow}\bar{I}_D{\uparrow}\bar{S}_D \ ; \ c_2 \in R, \text{ and } c = c_1 + c_2 + C(\beta) \}$$

Let Q and R be two sets of dotted rules then

$$Q\times R = \{ A{\downarrow}\bar{I}_A{\uparrow}\bar{S}_A \Rightarrow \alpha B{\downarrow}\bar{I}_B{\uparrow}\bar{S}_B\bar{\beta}.\gamma \ ; \ c \ | \ A{\downarrow}I_A{\uparrow}S_A \Rightarrow \alpha.B{\downarrow}I_B{\uparrow}S_B\beta\gamma \ ; \ c_1 \in Q,$$
$$B{\downarrow}\bar{I}_B{\uparrow}\bar{S}_B \Rightarrow \delta. \ ; \ c_2 \in R, \text{ and } c = c_1 + c_2 + C(\beta) \}$$

$$Q^*R = \{ A{\downarrow}\bar{I}_A{\uparrow}\bar{S}_A \Rightarrow \alpha B{\downarrow}\bar{I}_B{\uparrow}\bar{S}_B\bar{\beta}.\gamma \ ; \ c \ | \ A{\downarrow}I_A{\uparrow}S_A \Rightarrow \alpha.B{\downarrow}I_B{\uparrow}S_B\beta\gamma \ ; \ c_1 \in Q,$$
$$B{\downarrow}\bar{I}_B{\uparrow}\bar{S}_B \Rightarrow^* D{\downarrow}\bar{I}_D{\uparrow}\bar{S}_D \ , \ D{\downarrow}\bar{I}_D{\uparrow}\bar{S}_D \Rightarrow \delta. \ ; \ c_2 \in R, \text{ and } c = c_1 + c_2 +$$
$$C(\beta) \}$$

In the Aho and Peterson algorithm V_t^* was partitioned into disjoint classes E_0, E_1 .... such that

$x \in E_i$ iff x can be transformed into a valid string using exactly i error productions and

$x \not\in E_0 \cup \cdots \cup E_{i-1}$

To find out which class the given input x belongs the Aho and Peterson algorithm uses Earley's parsing algorithm and in each list I_0 through I_n it keeps only one copy of a dotted rule configuration. It keeps the copy that uses minimum number of error productions. We can do a similar partitioning in our algorithm. In each set we keep only one copy of a dotted rule, and we keep the one that cost least. Two dotted rules having same symbols but different attribute values will be considered different dotted rules.

Now we are ready to define the second level error correction algorithm. Initial steps in this algorithm are same as in Mauney's [MA82] algorithm (except that while parsing we have to consider attribute values as described before). That is, using the new definitions of * and x products we parse the sequence of sets S_1 ... S_k. After parsing these k sets we shall have all the possible corrections. With each correction we shall also have minimum cost of correction (this cost will be minimum because when we insert a nonterminal or terminal with some attributes with unknown values, we add the minimum cost of insertion over all the possible attribute combinations). We also maintain some links similar to the one used in Earley's algorithm so that we can build parse tree for each correction. For completeness we shall give the complete GHR algorithm below.

```
begin
$t_{0,0}$ := PREDICT({$S \Rightarrow .X_n{\downarrow}a_1^n \ldots {\downarrow}a_{N_n}^n \; {\uparrow}b_1^n \ldots {\uparrow}b_{M_n}^n \ldots X_1{\downarrow}a_1^1 \ldots {\downarrow}a_{N_1}^1 \; {\uparrow}b_1^1 \ldots {\uparrow}b_{M_1}^1$ })

    $\cup \{S \Rightarrow .X_n{\downarrow}a_1^n \ldots {\downarrow}a_{N_n}^n \; {\uparrow}b_1^n \ldots {\uparrow}b_{M_n}^n \ldots X_1{\downarrow}a_1^1 \ldots {\downarrow}a_{N_1}^1 \; {\uparrow}b_1^1 \ldots {\uparrow}b_{M_1}^1 \}$;
(* Build column 1 thru k+1 *)

 for j := 1 to k+1 do
 (* Using column 0...j-1 build column j *)
 begin
  (* the set $S_j$ is same as defined above *)
  $t_{j-1,j} := t_{j-1,j-1} * S_j$;

  for i := j-2 downto 0 do
   begin
    r := ( $\bigcup_{i<k<j-1} (t_{i,k} \times t_{k,j})) \cup (t_{i,j-1} \times (t_{j-1,j} \cup S_j))$;

    $t_{i,j} := r \cup t_{i,i} * r$;

   end;
  $t_{j,j}$ := PREDICT( $\bigcup_{0 \leq k < j} t_{i,j}$);

      (* While taking the x or * product, we have to compute various
      attributes of the symbols of the dotted rule. These attributes
      are evaluated by executing the appropriate plan. When we
      include a nonterminal as an argument to the PREDICT function,
      we include its inherited attributes. Therefore, a nonterminal
      could be included more than once with different inherited attri-
      butes. *)

end;
```

After finding all the possible corrections we pick one correction with minimum cost as follows. In all the corrections found so far, some of the attributes might have the value 'unknown'. While parsing we try to replace as many attribute positions as possible from the value 'unknown' to some fixed value from the corresponding attribute domain, as explained before. We *hope* that there will be very few attribute positions with the value 'unknown'. Performance of the following algorithm depends on the number of attribute positions with the value 'unknown' and the size of their attribute domains. Fixing as many attribute values as possible while parsing is very critical, especially for the attribute positions whose domain is rather large. From our implementation experience we feel that could be done in most cases. For example, the attribute domain of the attribute 'id name' (see the example in the section 3.6) is very large (we can say it is finite, if we put an upper limit on the number of characters allowed in an id name), but our experience is that this attribute value can always be fixed while parsing as we evaluate various predicate functions.

We now consider each correction tree in the increasing order of minimum cost of correction and try to fully evaluate it. If we can fully evaluate a correction tree without any semantic error then we chose that correction; otherwise we take the next correction tree and try to fully evaluate that tree, and so on. Eventually we will find one semantically correct parse tree (because we assume that the parser has the correct prefix property, therefore, there will always be at least one semantically correct correction). Let these correction trees be T_1 ... T_n in the order of increasing minimum cost of correction. In this algorithm we shall use '?' to represent an undefined string and we assume that the correction cost of '?' is ∞.

```
(* Algorithm to pick the least cost correction *)
begin
 CORRECTION := ? (* best correction found so far *)
  for j := 1 to n do
    if  cost(CORRECTION) $\leq$ minimum_cost($T_j$) then
       return(CORRECTION) (* no cheaper correction possible *)
     else
       begin
          Let $j^{th}$ correction tree be
             $B_j \downarrow I_B \uparrow S_B$
                .
               /|\
              / | \
              .....
        Some attribute positions of $B_j$ will have the value 'unknown'; for
        these we place the full attribute domain.  Take all the possible
        combinations of $I_B$ and $S_B$ and consider them in the order of
        increasing insertion cost. Since all the attribute domains are fin-
        ite, there are only finitely many combinations to consider.  Sup-
        pose there are m combinations $<I_B^1 , S_B^1>, \ldots ,<I_B^m , S_B^m>$
        (* These combinations can be obtained
           very efficiently by using the S table *)
       for i := 1  to m do
        if  cost(CORRECTION) $\leq$ $S(B_j \downarrow I_B^i \uparrow S_B^i)$ then
          exit loop (* no better correction
                              possible from this tree *)
      else
        begin
        Evaluate all the attributes of the tree
        using $i^{th}$ combination for $B_j$'s attributes
        (* All the subtrees are also evaluated
          in the order stated above.          *)
        if cost_correction < cost(CORRECTION) then
          CORRECTION := new_correction
        end;
    end;
```

3.6. An Example

We now illustrate various ideas using an example. This grammar is a part of a Pascal grammar, where four types of scalar variables (Integer, Real, Boolean, and Character) can be declared and used in various statements. No variable can be declared more than once in the same scope and no variable can be used without being declared.

The predicate *<CHECK DUP>* insures that a variable is not declared more than once in the same scope. This predicate has been placed such that a duplicate definition is detected as soon as the offending identifier appears in the lookahead. The following production illustrates.

<DECL LIST>↓symtab↑symtabout ⟹ DECL

<CHECK DUP>↓symtab <ID>↑name : <TYPE>↑typ

<DECL LIST>↓symboltab2↑symtabout

<DECL LIST>[2].symboltab2 := <DECL LIST>[1].symtab $\cup$

{(<ID>.name, <TYPE>.typ)};

The predicate *<CHECK DEF>* insures that a variable has been declared before being used. It has been placed so that the correct prefix property is maintained. The following production illustrates this idea.

<LEFT HAND>↓symtab↑type ⟹ *<CHECK DEF>*↓symtab↓'unknown'↑type
<ID>↑name

(Here the value 'unknown' for the attribute ltype of the predicate *<CHECK DEF>* indicates that an id of any type is acceptable as long as it has been declared before, in other contexts we shall be looking for an id of a specific type as we shall see below).

In this grammar we have only two kinds of statements, assignment and *if*. In an assignment statement the type of right hand side expression must match the type of the variable on the left hand side (except when the left hand side is real; then the right hand side may be an integer expression). This semantic rule is enforced using inherited attribute "ltype" which passes down the type of the left hand side variable. As the right hand side expression is recognized, the type is checked at all stages to make sure that there is no conflict. For example consider the following production.

<PRIMARY>↓symtab↓ltyp↑typ ⟹
<CHECK DEF>↓symtab↓ltyp↑typ <ID>↑name

Predicate *<CHECK DEF>* is *true* if and only if the lookahead identifier has been declared before and its type is compatible to the inherited attribute "ltype". The value of the inherited attribute "ltype" was obtained from the type of the left hand side variable and the partially parsed right hand side expression. For example, if the left hand side is Boolean and we have already parsed an integer variable and a relational operator then the inherited attribute 'ltype' of the nonterminal <PRIMARY> will be integer (for details see the complete grammar given below).

The expression used in an *if* statement must of the type Boolean, and this restriction is enforced by assigning the value Boolean to the inherited attribute "ltype" of the nonterminal <EXPR>. For example, consider the following production (we enclose the constant attribute values in single quotes).

<STMT>↓symtab ⟹ IF <EXPR>↓symtab↓'Boolean'↑typ
THEN <STMT>↓symtab <ELSE CLAUSE>↓symtab

As the nonterminal <EXPR> is recognized, the type is checked at all stages to make sure that there is no conflict. The final type of the <EXPR> (given by the attribute 'typ') is not important in this context. (The same nonterminal <EXPR> is used in other context also, were the 'ltype' and the 'typ' could have different values, e.q. in an assignment statement).

Finally, in this grammar we allow arithmetic operations +, -, *, and / only on Integer and Real variables. This semantic rule in enforced by the predicate *<CHECK ARITH>*. Here also this predicate is put before the +, -, *, and / operators so that an error can be detected as soon as possible, leaving enough information on the stack for error corrector.

We now give the complete grammar G = (V_n, V_t, <prog>, Q, F_Q, P) where

V_t = { <ID>, PROGRAM, BEGIN, END, IF, THEN, ELSE, DECL, :=, ",", ; , (,), :, .,
+, -, *, /, INTEGER, REAL, CHAR, BOOLEAN, <constant>}

V_n = {<PROG>, <PROG HEAD>, <STMT PT>, <FILE ID LIST>, <EXPR>, <LEFT HAND>, <TYPE>, <STMT LIST>, <STMT L TAIL>, <STMT>, <ELSE CLAUSE>, <DECL LIST>, <CHECK DEF>, <CHECK DUP>, <CHECK TYP>, <TERM>, <E TAIL>, <PRIMARY>, <T TAIL>, <ADD OP>, <MULT OP>, <EX TYP>}

Q = {*<CHECK DUP>*, *<CHECK TYP>*, *<CHECK DEF>*, *<CHECK DEF>*, *<CHECK ARITH>*}

P contains following productions

<PROG>↑symtabout ⟹ <PROG HEAD> <STMT PT>↓Φ↑symtabout .

<PROG HEAD> ⟹ PROGRAM <ID>↑name ;

<STMT PT>↓symtabin↑symtabout⟹ BEGIN
<DECL LIST>↓symtabin↑symtabout <STMT LIST>↓symtabout END

<DECL LIST>↓symtab↑symtabout ⟹ DECL *<CHECK DUP>*↓symtab
<ID>↑name : <TYPE>↑typ <DECL LIST>↓symboltab2↑symtabout
<DECL LIST>[2].symboltab2 := <DECL LIST>[1].symtab ∪
{(<ID>.name, <TYPE>.typ)};

<DECL LIST>↓symtab↑symtab ⟹ λ

<TYPE>↑typ ⟹ INTEGER
<TYPE>.typ := Integer;

<TYPE>↑typ ⟹ REAL
<TYPE>.typ := Real;

<TYPE>↑typ ⟹ CHAR

    <TYPE>.typ := Character;

<TYPE>↑typ ⟹ BOOLEAN

    <TYPE>.typ := Boolean;

<STMT LIST>↓symtab ⟹ <STMT>↓symtab <STMT L TAIL>↓symtab

<STMT>↓symtab ⟹ <LEFT HAND>↓symtab↑ltyp :=

    <EXPR>↓symtab↓ltyp↑typ2 *<CHECK TYP>*↓ltyp↓typ2

<LEFT HAND>↓symtab↑typ ⟹ *<CHECK DEF>*↓symtab↓'unknown'↑typ

    <ID>↑name

(The value 'unknown' for the attribute ltype of the predicate *<CHECK DEF>* indicates that id of any type is acceptable as long as it has been declared before).

<STMT>↓symtab ⟹ IF <EXPR>↓symtab↓'Boolean'↑typ

    THEN <STMT>↓symtab <ELSE CLAUSE>↓symtab

<STMT>↓symtab ⟹ BEGIN <DECL LIST>↓symtab2↑symtabout

    <STMT LIST>↓symtabout END

    <DECL LIST>.symtab2 := <STMT>.symtab $\cup$ {empty new scope}

<STMT>↓symtab ⟹ λ

<ELSE CLAUSE>↓symtab ⟹ ELSE <STMT>↓symtab

<ELSE CLAUSE>↓symtab ⟹ λ

<STMT L TAIL>↓symtab ⟹ ; <STMT>↓symtab

    <STMT L TAIL>↓symtab

```
<STMT L TAIL>↓symtab ⟹ λ

<EXPR>↓symtab↓ltyp↑typ ⟹ <TERM>↓symtab↓ltyp↑typ1
        <E TAIL>↓symtab↓ltyp2↑typ2 <EX TYP> ↓typ1↓typ2↑typ
     <E TAIL>.ltype2 := Compute_ltype_of_Tail(<TERM>.typ1,
                                              <EXPR>.ltyp);

<E TAIL>↓symtab↓ltyp↑typ ⟹ <CHECK ARITH>↓ltyp
      <ADD OP>↑op <TERM>↓symtab↓ltyp↑typ1
        <E TAIL>↓symtab↓ltype2↑typ2 <EX TYP> ↓typ1↓typ2↑typ
     <E TAIL>[2].ltype2 := Compute_ltype_of_Tail(<TERM>.typ1,
                                                 <EXPR>.ltyp);

<E TAIL>↓symtab↓ltyp↑ltype ⟹ λ

<TERM>↓symtab↓ltyp↑typ ⟹ <PRIMARY>↓symtab↓ltyp↑typ1
      <T TAIL>↓symtab↓ltyp2↑typ2 <EX TYP>↓typ1↓typ2↑typ
     <T TAIL>.ltype2 := Compute_ltype_of_Tail(<PRIMARY>.typ1,
                                              <TERM>.ltyp);

<T TAIL>↓symtab↓ltyp↑typ ⟹ <CHECK ARITH>↓ltyp
        <MULT OP>↑op <PRIMARY>↓symtab↓ltyp↑typ1
        <T TAIL>↓symtab↓ltyp2↑typ2 <EX TYP>↓typ1↓typ2↑typ
     <T TAIL>[2].ltype2 := Compute_ltype_of_Tail(<PRIMARY>.typ1,
                                                 <TERM>.ltyp);

<T TAIL>↓symtab↓ltyp↑ltyp  ⟹ λ

<PRIMARY>↓symtab↓ltyp↑typ ⟹ <CHECK ARITH>↓ltyp -
```

```
                <PRIMARY>↓symtab↑typ

<PRIMARY>↓symtab↓ltyp↑typ ⟹ <CHECK DEF>↓symtab↓ltyp↑typ
                <ID>↑name

<PRIMARY>↓symtab↓ltyp↑typ ⟹ <CHECK TYP>↓ltyp↓typ
                constant↑typ

<MULT OP>↑op ⟹ *
    <MULT OP>.op := Mult;

<MULT OP>↑op ⟹ /
    <MULT OP>.op := Divd;

<ADD OP>↑op ⟹ +
    <ADD OP>.op := Add;

<ADD OP>↑op ⟹ -
    <ADD OP>.op := Sub;

<CHECK DUP>↓symtab ⟹ λ

<CHECK TYP>↓typ1↓typ2 ⟹ λ

<CHECK DEF>↓symtab↓ltyp↑typ ⟹ λ
  <CHECK DEF>.typ := If lookahead.name ∈ <CHECK DEF>.symtab
                       and typeof(lookahead) is assignable
                       to <CHECK DEF>.ltype
                     then typeof(lookahead)
                     else Predicate := false ;
```

Where typeof(lookahead) gives the type of the lookahead variable as recorded in the symbol table.

<EX TYP>$\downarrow$typ1$\downarrow$typ2$\uparrow$typ $\Rightarrow \lambda$

<EX TYP>.typ := Compute_expr_type(<EX TYP>.typ1, <EX TYP>.typ2);

In this example, all the attribute domains are finite except for the attribute 'name' and 'symbol table'. Our error correction algorithm treats them differently than the rest of the attributes. When we want to insert an id, we set its attribute 'name' to some special pattern 'UNKNOWN'. When the table lookup is called by the predicate *<CHECK DEF>*, it knows that we are not looking for an id named 'UNKNOWN'. Rather any id with the correct type will be acceptable (we recall that when the table lookup is called it uses the inherited attribute ltype to match the type). Therefore, it will match this id with any other id as long as it is of the correct type and change the attribute 'name' to the proper value. Spelling correction can be incorporated easily. When we have an undefined identifier, we can call the table lookup with an option which says that any identifier whose spelling is very close the attribute 'name' is acceptable. It will match this identifier to the closest matching identifier of the correct type which has been defined. The table lookup will change the attribute 'name' accordingly.

Similarly when the table lookup is called by the predicate *<CHECK DUP>* with an id named 'UNKNOWN', it will change the id name to some pattern such that it will not cause duplicate definition in the same scope.

3.7. Determining the value of k

The performance of the above described algorithm depends on the value of the k (the amount of lookahead used). If we make the k very small then the corrector may not find the best corrections, causing spurious errors. If we make the value of k very large then the corrector might become unreasonably expensive. Mauney [MA82] has proposed several methods for selecting the value of k. His implementation results (reported in [MA82]) should be applicable for our algorithm and could be used as guidelines for our algorithm. Briefly, we can use the following approaches for selecting the value of k.

We can use some constant value of k and always examine k input symbols past the error token. A constant k has been used by several other error correcting algorithms [GHJ79, PK80]. In practice a small value of k will give fairly good results. For example, the algorithm proposed by Graham *et. al.* [GHJ79] uses 5 symbols past the error token and the authors have reported satisfactory results. It is, of course, always possible to come up with an error such that for best correction we need to examine the whole program. Moreover, if there are more errors in the examined k symbols, the later errors may not be corrected very well.

Since the value of the k can be changed dynamically in our algorithm, we can solve abovementioned problem if we always examine some constant number of symbols after the last error symbol. Tai [TA78] first suggested this method. The only problem with this approach is that if there are too many errors in the program then we might have to examine a large part of the remaining program, which could be very expensive. As pointed out by Mauney [MA82], simply counting the number of the symbols after the last error symbol is not likely to be a very good approximation, as some of the symbols could just be "syntactic sugar" (such as ',').

Rather then just counting symbols after the error symbol, we might look for special symbols such as "fiducial symbols" as defined by Pai and Kieburtz [PK80]. These symbols have 'phrase-level uniqueness' property; that is, all the sentential suffixes starting with a fiducial symbol can be derived from a single sentential suffix. If we correct all the errors until we encounter a fiducial symbol, we should be able to parse rest of the program if there are no further errors in the program.

A problem with this method is that the fiducial symbols are grammar-dependent and therefore, determining fiducial symbols requires a *priori* analysis of the grammar. Very few symbols in any grammar for a practical programming language are fiducial symbols. Therefore, we might have to examine a large part of the program before we encounter a fiducial symbol.

The last suggestion made in [MA82] is to keep expanding the error correction region until all the corrections are equivalent (this is explained in detail in [MA82]). Informally, two corrections are equivalent if they accept the same suffixes. Equivalence was first suggested by Levy [LE75]. The problem with this approach is that test for equivalence is undecidable in general. But we can still find sufficient conditions for equivalence, which are decidable in practice. Levy [LE75] has given one such condition, but it is rather difficult to test in practice. Mauney has given a slightly weaker test for equivalence (for details on this test for equivalence see [MA82]).

All these approaches can be used to limit the number of symbols to be examined. In practice no one criterion will serve well. Rather we should use some combination of these ideas. For example, we can have a constant upper bound, p, over the number of symbols we examine. We stop reading extra input symbols if we find a fiducial symbol before p. We stop reading extra input symbols if we have examined m (m<p) correct symbols after the last error symbol.

3.8. Comparison with other algorithms

One-symbol error correction has been proposed by several authors [GHJ79, PK80], but no elegant implementation has been suggested. Both methods take all possible one-symbol corrections and try one correction at a time by parsing some right context past the error token. If the correction tried is not successful they reset the parse stack to original state and try the next correction. We feel that our algorithm is quite clean and should be time-efficient. (Although true comparison is not possible as comparing cpu times of different computers is almost impossible, if not meaningless).

Most of these methods detect parsing (syntax) errors and then try a one symbol correction that is semantically correct. They fail to correct semantic errors (that is, a construct which is syntactically correct but semantically wrong will not be corrected by these algorithms). For example, type incompatibility in an assignment statement, use of an undefined variable, duplicate definitions of a variable in the same scope, wrong field selectors in a record variable (e.g. '^' instead of '.' in pascal), or wrong field selectors in a pointer variable (e.g. '.' instead of '^.' in pascal) will not be corrected by most of these methods. Since our algorithm will detect and correct these fairly common errors we feel that a larger percentage of errors will be corrected by our first-level error corrector.

Any high quality error corrector must have some means to deal with spelling mistakes. It has been reported that more than 20 percent errors in the COBOL programs are spelling errors [HD80]. Including these corrections is very easy in our algorithm. Once we find out all the possible tokens which could have been misspelled by the error symbol, we can include all these tokens in the set S_1 (or in the sets S_1 thru S_k in second level error correction).

The above algorithm is fairly general and user can tune it according to his need. The user can change the cost of insertion and deletion to fine-tune the algorithm. The value of k can be increased dynamically at run time.

We also note that above algorithm is very similar to the Aho and Peterson [AP72] algorithm. If we put PREDICT(S) in the element $t_{0,0}$ and set the value of k to n (the size of the input) then the algorithm will transform the given input string x into a legal string using least cost of correction. (To obtain exactly same results as the Aho and Peterson algorithm we must have insertion, deletion, and replacement cost 1 for all symbols).

Our algorithm has same flavor as Dion's [DI79] error correction algorithm except that it uses the next k input symbols to find a best possible correction whereas Dion's algorithm works on one symbol lookahead at a time. Therefore, we expect our algorithm to make better corrections than the corrections made by [DI79]. The E table used in Dion's algorithm could be very large and therefore could be expensive to access from the disk. The need for the E table has been eliminated in this algorithm by a slight modification to the PREDICT function. This modification will not have any significant effect on the execution time the GHR algorithm or memory requirement for storing the parse matrix. Elimination of E table should make our algorithm more practical as we do not have to store a large E table in the memory. Even the time complexity of our algorithm should be comparable to the time complexity of [DI79] provided we put a constant upper bound on the value of k, otherwise linearity for the algorithm cannot be established.

The above algorithm can also be extended to handle infinite attribute domains in some limited cases. We put the value 'unknown' for those attribute positions where we cannot put a unique value and continue to parse the right context. While parsing we let the predicates choose a specific value for these attribute positions where we have put 'unknown'. This cannot be done in general, but for some practical cases we were able to do this. For example, when we insert an identifier, we do not know what value we should have for the attribute 'id name' so we put the value 'unknown' for the attribute 'id name' and let the predicate *<CHECK DEF>* or *<CHECK DUP>* (depending on the context, we shall call one of these two predicates) change this value to some specific identifier name such that it does not cause undefined identifier or duplicate definition errors (as explained in the section 3.7).

By putting a constant upper bound on the value of k and assuming bounded length parse stack (parse stack used for parsing the whole program), we can retain the linear time and space complexity of the parser with the worst case constant of linearity equal to $k^3|G|$ (excluding the attribute evaluation time). This follows from the fact that all the attribute domains are finite, therefore there are only finite many dotted rules (including in the parse matrix element $t_{0,0}$, where we put $S \Longrightarrow .X_n$ .... Since the parse stack is of bounded length, in $t_{0,0}$ also we shall have only finite many dotted rules). If the value of k is constant, then we can have only finite many dotted rules in the whole parse matrix.

Therefore, the number of correction trees in the parse matrix is bounded by a constant. Each correction tree is of bounded height (because of constant k and n). Hence each error correction takes a constant amount of time, and the linearity follows from that. Even if we donot assume a bounded parse stack, we can say that our algorithm is linear in the size of parse stack and the size of the input. Though in practice time will be much less than the worst case performance as first-level corrector will correct a large precent of errors. Our algorithm will not degrade the performance of the parser on error-free programs, as the parser does not have to do any thing extra for the error corrector.

3.9. Possible Improvements To The Algorithm

The above algorithm is very general and can be used with any non-circular attributed grammar with finite attribute domains. This generality may not be required in practice. Some of the attribute places used in the grammar may not be important from semantic error correction point of view (e.g. if the value of a constant is one of the attributes of a constant, then the attribute domain is finite but very large. If we try all the possible attribute values to see which value is semantically correct, we could very well spend a large amount of cpu time. Further, we cannot even decide at the compile time if a value of a constant can cause divide by zero or overflow. For example, consider the following integer expression *A/(B + constant)*, here if we put 0 for the integer constant, we can get divide by zero if the value of *B* is zero. If we put 1 for the integer constant, we can get overflow if *B* has the largest possible integer value or divide by zero if the value of *B* is -1. Such attribute positions should be excluded from the error corrector). Therefore, to make above algorithm efficient we can divide all the attributes into two disjoint sets S_A^1 and S_A^2. The set S_A^1 contains only those attributes which are important for semantic correctness and the set S_A^2 contains attributes which are important for program execution but not important for semantic correctness (attribute position value of a constant is one example of such attributes). We also assume that all the predicates used by the error corrector are dependent only on the attributes from the set S_A^1 . Once we have a correction tree with attribute values fixed for all the attributes from the set S_A^1 , we can decorate the tree with attributes from the set S_A^2 without causing any attribute conflict (or semantic error).

If all the attribute domains for attributes from the set S_A^1 are fairly small then we can trade space for time and instead of using the 'unknown' value we use the full attribute domains for those places where we cannot put a unique value. We can include the same symbol a number of times with different attribute values. At the end of parsing the next k input symbols we can just pick a correction which has the least cost of correction. Since all the attribute values are fixed, no further action will be required (except selecting some attribute values for attributes from the set S_A^2).

3.10. Implementation Results

The error correction algorithm described above was implemented in pascal on a VAX/780 computer and was tested on the subset of Pascal grammar given in the section 3.6. Execution time of the first level error correction is acceptably small. In our test cases we tried to parse 5 token past the error token (total 6 tokens) and the execution time varied from 80 to 200 msecs. This execution time included time to repair the input source line and print error message. Given the quality of the correction made by the first level error corrector, execution time does seem acceptable. Some of the programs we tested are listed in appendix C to show the kind of corrections made by our algorithm.

The second level error correction is much more elaborate and expensive. In our test cases we tried to parse 4 to 7 tokens past the error token and the execution time varied from 1.4 (for four symbols) to 9.0 seconds (for 7 symbols). This high execution time indicates that the number of symbols parsed after the error token is very critical. As insertions, deletions, and replacements are included, the effective grammar as seen by the parser (even though underlying grammar is still the same, but the parser sees slight different picture as we simulate insertions, deletions, and replacements) becomes highly ambiguous (in the sense that each dotted rule can be included a number of times in the same parse matrix element) and we start to see the cubic parsing time. For example, in test cases when we counted the number of dotted rules in each element of the parse matrix, we found that the average number of dotted rules in each parse matrix element was very small (it varied from 12 to 20). From these small number of dotted rules in each parse matrix element and the execution trace we could conclude that that the execution time was high because each dotted rule was introduced several times with different corrections in the same parse matrix element. In conclusion, high execution time for the second level error corrector is due to the ambiguity in the effective grammar as seen by the parser.

Since we feel that a large percentage of errors will be corrected by the first level error corrector (which is much faster than the second level corrector), overall error correction strategy seems to be reasonably efficient and we should be able to use it in practice. Even if we use the full pascal grammar, the execution time of the first level error corrector should not change by any significantly amount. In test cases we found that the execution time of the first level error corrector depends on the context in which the error occurs. Productions which are not applicable in the current context have no effect on the execution time. For example, in a pascal program if we have an error in the declaration part and we try a one-symbol error correction, all the productions used in the expressions etc. will have no effect at all.

But the execution time of the second level error corrector will increase when we use the full pascal grammar. Here the execution time is directly proportional to the size of the grammar (because of the $|G|$ factor in $|G|k^3$ as explained before).

Chapter 4

Conclusions

4.1. Conclusions

One of the goals of this research was to develop an efficient and very general on-the-fly attribute evaluator. The attribute evaluator described in this thesis runs in parallel with the parser and tries to evaluate attributes as soon as possible, without any run time analysis of attribute dependencies. It also avoids unnecessary visits to a subtree if no additional attributes can be evaluated from that subtree.

The evaluator we have presented is very general and can be used with any context-free grammar augmented with an absolutely non-circular attributed grammar. This generality should make it very useful in several applications as mentioned in the section 2.11.

The attribute dependency analysis and attribute evaluation planning algorithm presented in this thesis is very efficient, based on some very simple data structures. It can be used for other applications as well. For example, it can be used for attribute evaluation planning in a tree walk evaluator [KW78].

The second goal of this research was to develop a good context-sensitive error correction strategy. The two level error correction algorithm presented in this thesis seems a reasonable approach. It is an automatic table-driven algorithm, and will work in all the error situations. Although the problem of error correction has previously received much attention, most of the other techniques suffer rather serious drawbacks. Very often, they fail when faced with certain syntax errors and are forced to skip ahead, completely ignoring portions of the input. Most of the error correctors can deal with syntax error only. Hence corrections made by most of these correctors can be rejected immediately when the semantic phase of the compiler is started after an error correction.

Some of the previously proposed context sensitive error correctors have rather serious drawbacks. For example, the algorithm proposed by Graham at el [GHJ79] and other one-symbol context-sensitive error correctors can detect *context free* (parsing) errors only and then try a one-symbol correction that is semantically correct. They fail to correct semantic errors (that is, a construct which is syntactically correct but semantically wrong will not be corrected by these algorithms). For example, type incompatibility in an assignment statement, use of an undefined variable, duplicate definitions of a variable in the same scope, wrong field selectors in a record variable (e.g. '^' instead of '.' in pascal), or wrong field selectors in a pointer variable (e.g. '.' instead of '^.' in pascal) will not be corrected by most of these methods. Since our algorithm will detect (because our error corrector can be invoked by semantic errors also) and correct these fairly common errors we feel that a larger percentage of errors will be corrected by our first-level error corrector.

The work presented in this thesis has both theoretical and practical significance. The two level error correction strategy makes it rather practical. But the basic error correction algorithm is very general and can even be used as a global error correction algorithm.

The error correction technique developed here has the fundamental advantage that the introduction of error correction in the translation process has very little impact on the overall structure of a compiler. This noninterference is a direct consequence of the locality of our correction model.

4.2. Directions for Future Research

This research presents a structured approach to context-free (parsing) and context-sensitive (semantic) error correction. We should be able to integrate this technique with a lexical error corrector (e.g. spelling corrector) to obtain a truly high quality error correction algorithm. For example, in our current implementation, when we encounter an undefined identifier we replace it with any identifier from the symbol table as long as it is of right type. An identifier selected this way may not be what the user wants. On the other hand, if we use a spelling corrector and replace an undefined identifier with another identifier whose spelling is very close to the offending identifier, it is more likely to be correct. We feel that integrating our error correction algorithm with a spelling corrector can greatly enhance the overall quality of the error corrector.

We have presented an automatic error correction algorithm which can deal with an attributed grammars whose domains are finite. We have also shown informally how to deal with infinite attribute domains in some limited situations. Here we find that the predicate functions play an important role. The predicates replace the 'unknown' values by some specific values from the corresponding attribute domains. More research is required to define a precise set of predicate functions such that given the 'unknown' values to some attribute positions, predicates will always select specific attribute values from the corresponding attribute domains.

Appendix A

Planning Algorithm

When a VISIT(k,l) is executed, the recursive call to EVALUATE will result in the look-up NEXTMOVE(q,l) = (plan,q'), where q is the state found on the kth son. The process of constructing an evaluator involves finding all (q,l) pairs that will ever be used to access the NEXTMOVE table and making all the relevant entries in the NEXTMOVE table and PLAN set.

To see which (q,l) pairs will be used in a *plan*, let us consider the plan associated with the start symbol (VISIT to the root). Without loss of generality, let us assume that we have exactly one production with the S (the start symbol) on the left-hand side, and let

$$S \Longrightarrow X_1 \ldots X_n$$

Suppose that the plan for VISIT(root,l_S) is $P = I_1 \ldots I_m$. Where I_k is either a semantic function or VISIT(j,l) (for $1 \leq j \leq n$) for $1 \leq k \leq m$.

To find out which NEXTMOVE(q,l) look-up might be required by the VISIT(j,l), we must know what state X_j could be left in by the most recent visit to it. To keep track of the states a offspring could be left in, we associate a set with each symbol X_j from the right hand-side. Initially each offspring could be in one of its initial states. Therefore, the set of states X_j could initially be is S_j, where

$$S_j = \{q_0\,(p_i) \mid p_i \in P \text{ and the left-hand side of } p_i \text{ is } X_j\}$$

Now suppose I_j = VISIT(k,l) is the first VISIT instruction in the plan p, then we may need to look-up one of the following entries from the NEXTMOVE table :

NEXTMOVE(q,I), Where $q \in S_k$

Therefore, we should make a plan for each (q,I) pairs such that $q \in S_k$. And we should also update the set S_k, because the next visit to the same offspring will find this node in some other state. A new set of states S'_k the symbol X_k could be in, can be computed as follows

$$S'_k = \{q' \mid \text{NEXTMOVE}(q,I) = (\text{plan}',q') \text{ for some } q \in S_k\}$$

This way we can find all the pairs for which NEXTMOVE lookup might be required by the plan associated with the visit the root instruction. Now suppose we visit a node X, which is in state q, with input I. And suppose that the production used at that node is

$$X \Longrightarrow X_1 \ldots X_m$$

From the state q of the node X, we can find a set of states each offspring could be flagged. This is due to the fact that the state of a node can tell us which attributes of a node are available. If we know which attributes of X are available, we also know which inherited attributes of X_k are available, and therefore, we know which synthetic attributes of X_k are available, for each X_k from the right-hand side.

Since the state of a node is a pair (p,A), where p is a production number and A is a set of attribute occurrences of the symbol, if we know the attribute occurrences which are available, we know the set of states that node could be in (there will be one state in this set for each production into which the node symbol can be expanded). Hence the set of states each offspring could be flagged can be uniquely determined from the state of the parent node (we will see that we do not have to recompute these sets if we associate some bookkeeping variables with each state).

Now reconsider the previous example, and suppose we visit a node X, which is in state q, with input I. Suppose that the production used at the node is

$$X \Longrightarrow X_1 \dots X_m$$

We also have a state set S_k associated with each X_k for $1 \leq k \leq m$, which says that X_k could be in one of the states from S_k. Suppose the plan corresponding to VISIT(q,I) is $p = I_1 \dots I_n$.

Now consider each instruction I_j from the plan p. If I_j is a semantic function then no further action is required. If I_j is VISIT(k,I), then we must include NEXTMOVE(q,I) for each $q \in S_k$, if it is not already present. We must update the set S_k (as explained before).

Now we are ready to define the complete planning algorithm.

With each state $q \in Q$ we shall associate a number of sets of states (one for each symbol from the right-hand side of the production). Suppose we have a state $q \in Q$ such that the node X can be flagged with state q, and

$$X \Longrightarrow X_1 \dots X_n$$

then we shall associate n sets of states $S_1 \dots S_n$. S_k, $1 \leq k \leq n$, indicates that when X is in state q, then the K^{th} offspring could be in any state $q' \in S_k$. (Note that we have a number of states possible for K^{th} offspring because the symbol X_k can be expanded by a number of different productions. We shall have one state corresponding to each production X_K can be expanded into in S_k).

Initially we shall have the following states in Q

$Q = \{ q_0(p_j) \mid p_j \in P \}$

We shall denote the set of states associated with state q by $\$(q)$. If production p_j is $X \Rightarrow X_1 \dots X_m$ then

$\$(q_0(p_j)) = S_1 \dots S_m$ where

$S_k = \{ q_0(p_i) \mid p_i \in P$ and the left-hand side of p_i is $X_k \}$

As we add new states q to Q, we shall also include the sets of states associated with q (as described above) to $\$(q)$. For completeness we shall include the "Make_plan" algorithm here again.

```
Algorithm Make_Plan(p,A);
  1. Let S be an empty sequence of instruction and
     let A' = A;
  2. Let current evaluation state be (p,A');
  3. If some semantic function $f_{p,a}$ is ready to
     evaluate then append the instruction $f_{p,a}$ to
     S, add a to A', and go to step 2;
  4. If $K^{th}$ subtree has nonempty yield y then
     append VISIT(k,I) to S, where
     $I = \{ a \mid a \in I(X_k)$ and $a \in A' \}$
     Let $A' = A' \cup y$; and go to step 2;
  5. Let q be the evaluation state (p,A'). The
     resulting plan is S and nextstate is q.
  6. return (S,q) (* Return the resulting plan and
          the next state *)
```

```
end.

function Construct_plans(q,l) : state;
    (* This function adds an entry to the NEXTMOVE table for the pair (q,l).
    It also calls Make_plan to make a plan for pair (q,l) and add it to the
    PLAN set, and finally it will update the $ array *)
begin
 Let q be the state associated with node $X \Rightarrow X_1 \ldots X_m$.
 var S[1..m] : set of states; (* work space to keep track
          of running sets of states for each offspring *)
 S',S" : set of states;
 plan',plan" : sequence of instructions;
 q',q" : states;
 for i := 1 to m do
   S[i] := $(q)[i];  (* start with set of states
                      associated with the state q *)
   (plan',q') := Make_plan(q,l);
         (* make plan for pair (q,l) *)
   NEXTMOVE(q,l) := (plan',q');
       (* Make entry in NEXTMOVE table *)
   Let plan' be $I_1 \ldots I_n$
  for i := 1 to n do
    begin (* look for visits *)
     if $I_i$ is VISIT(k,l') then do
       begin
        S' = $\Phi$;
```

```
            for each q'' ∈ S[k] do
            if NEXTMOVE(q'',l') is undefined then do
               S' := S' ⋃ Construct_plan(Q'',l');
                          (*new pair (q,l) found*)
            else
               begin
                (plan'',q) := NEXTMOVE(q'',l');
                 S' := S' ⋃ q
               end;
              S[k] := S'; (*update set of state of kth son*)
           end;
   for i := 1 to m do
         $(q')[i] := S[i];
        return(q');
       end
   end. (* end of Construct_plan *)

   (* initialize *)
   Q := { $q_0(p_j)$ | $p_j$ ∈ P };
   $($q_0(p_j)$) := $S_1$ ... $S_m$ where
    $p_j$ is X ⟹ $X_1$ ... $X_m$ and
    $S_k$ = { $q_0(p_i)$ | $p_i$ ∈ P and the left-hand
                      side of $p_i$ is $X_k$ }
    NEXTMOVE := Φ;
    Construct_plans($q_0$(S),$l_S$);
```

We can make following observations about the plan construction algorithm presented above.

Above algorithm is based on very simple data structures (sets, arrays, etc). Therefore, efficient implementation in any PASCAL like language can be obtained very easily. Our implementation of this algorithm in pascal is about 5000 lines long. Execution speed is also quite acceptable. In our test cases we had a number of small grammars and two grammars each having approximately 40 productions. Our planning algorithm took about 9 second of CPU time on VAX/780 computer. In addition to creating various table files, the program produced a formatted listing of the full grammar, goto table, and complete plan set.

The algorithm will terminate. Since on every call we include a new (q,l) pair into the NEXTMOVE table and there are only finite number of pairs possible at all. Further, before we call the function Construct_plan recursively we include the pair (q,l) into the NEXTMOVE table, and therefore, possibility of loop is also not present.

Correctness of the algorithm can be established very easily using inductive hypothesis. We can establish correctness by proving following two hypothesis.

a. Each pair (q,l) which could ever occur in attribute evaluation will be included by the algorithm.

b. And each pair (q,l) included has a possibility of occurring in some attribute evaluation (provided that the predicates functions do not block some production combination from occurring).

Appendix B

Attributed S Table Computation

The following procedure STable computes the attributed S table as defined in the section 3.5. This procedure (adopted from [DI79]) computes the S table for an L-attributed grammar whose attribute domains are finite. We shall use the symbol '?' to denote an undefined string and we assume that the insertion cost of the undefined string is ∞. This is an iterative algorithm where the main procedure keeps reevaluating S table for each grammar symbol until there is no change, then we know that we done. Correctness and efficiency of the algorithm are discussed in [DI79].

We shall use the symbol LHS_i to represent the left hand side of the production p_i. An instance of a symbol X together with its attribute values will be denoted by $X{\downarrow}I{\uparrow}S$ where I is an M_X-tuple of values, each value in the corresponding domain of I(X), and S is an N_X-tuple of values, each value in the corresponding domain of S(X). The notation $I \in I(X)$ means $I = (a_1, \dots , a_{M_X})$, $I(X) = (d_1, \dots, d_{M_X})$ and $a_k \in d_k$ for $k = 1, \dots , M_X$ ($S \in S(X)$ is similarly defined).

Procedure ReEvalS(p_i) considers the reevaluation of $S(LHS_i{\downarrow}u{\uparrow}v)$ for all $u \in I(LHS_i)$ and $v \in S(LHS_i)$, using production i. SearchProdS is a recursive procedure which assigns values to attribute positions of the symbol in RHS_i by doing a depth-first search of the tree which can be build by considering all the possible combinations of attribute values.

The attribute values of a prefix of a production p_i are kept in the array v_k and w_k, which are used as stacks in the depth-first search.

```
procedure STable(AG);
  AG : An Attributed Grammar;
  function ReEvalS; forward;

  begin (* STable *)
    (* Initialization *)
    for all $A{\downarrow}I{\uparrow}S \in AV_n$ do
        $S(A{\downarrow}I{\uparrow}S)$ := '?';
    for all $a{\uparrow}S \in AV_t$ do
        $S(a{\uparrow}S) := a{\uparrow}S$;
    for all $q{\downarrow}I{\uparrow}S \in Q$ do
      if $f_q(I) = (S, true)$ then
        $S(q{\downarrow}I{\uparrow}S) := \lambda$
      else
          $S(q{\downarrow}I{\uparrow}S)$ := '?';
    (* Main loop *)
    repeat
      Nochange := true;
      for all $p_i \in P$ do
          Nochange := Nochange and not ReEvalS($P_i$)
    until Nochange;
  end STable.
```

```
function ReEvalS($P_i$) : boolean;
  $P_i = (A\downarrow a_0 \uparrow b_0 \Longrightarrow B_1 \downarrow a_1 \ldots B_m \downarrow a_m \uparrow b_m)$ ;

 (* Note $a_k = (a_1^k, \ldots, a_{M_k}^k)$ and

         $b_k = (b_1^k, \ldots, b_{N_k}^k)$ for $k = 0, \ldots, m$ *)

var
change : boolean;
   $v_k$ : domains($a_k$), $k = 0,\ldots,m$;
   $w_k$ : domains($b_k$), $k = 0,\ldots,m$;

(* where domains($a_k$) is a tuple of domains defined as follows. If $a_j^k$ is an attribute
variable then the jth component of domains($a_k$) is $I(a_j^k)$, otherwise it is $\{a_j^k\}$ );
domains($b_j$) is defined in a similar manner *)

 procedure SearchProdS; forward;
 begin (* ReEvalS *)
  1      change := false;
  2      for all $v_0 \in$ domains($a_0$) do
  3        if m = 0 then (* $\lambda$ production *)
  4          begin    5        copy $w_0$ from defining position;
  6           $S(A\downarrow v_0 \uparrow w_0) := \lambda$;
  7           change := true
  8          end
  9        else
  10          begin
  11           compute $v_1$;
```

```
            SearchProdS(1, $\lambda$)
          end;
      return(change)
  end ReEvalS;
```

```
procedure SearchProdS(j,LC);
  j : 1..m; (* level in the tree *)
  LC : $AV_t^*$;   (* least cost string derivable from $B_1 \dots B_{j-1}$ *)
  T : $AV_t^*$;

begin
    for all $w_j \in$ domains($b_j$) do
      begin
       T := $S(B_j{\downarrow}v_j{\uparrow}w_j)$;
       if T $\neq$ '?' then
          LC := LC cat T;
       if j < m then
          begin
           Compute $v_{j+1}$; (* inherited attributes *)
           SearchProdS(j+1, LC)
           end
        else
           Compute $w_0$; (* synthetic attributes *)
        if IC(LC) < IC($S(A{\downarrow}v_0{\uparrow}w_0)$) then
           begin
            $S(A{\downarrow}v_0{\uparrow}w_0)$ := LC;
            change := true
            end
      end
end SearchProdS;
```

Appendix C

Here are some of the test programs we run through the first level and the second level error corrector. For the first level error corrector, the corrections made are indicated by the symbol I, D, and R, indicating insertion, deletion, and replacement, respectively. We have shown only some of the examples which shows the difference between our error corrector and the other error correctors.

Examples 1 and 2 show common one-symbol context free corrections, like inserting missing ':=' or deleting extra 'DECL' and so on. In example 3 we see a context sensitive correction where an undefined identifier Y has been replaced by a defined identifier X. Example 4 shows two context free errors (missing '(' and extra ':') and a context-sensitive error (using an undefined identifier Y). Example 5 shows in more details, how to deal with duplicate definitions and undefined identifiers. Finally, 6 gives an example of type incompatibility in an assignment statement.

All the errors in the last example were corrected by the second level error corrector (readers can see that the first-level error corrector cannot correct any error). We see that the second level error corrector can deal with cluster of errors and make context-free (missing '(filename);', missing ':=', and missing ':') as well as context-sensitive (replacing undefined identifier).

```
PROGRAM foo(f1);
   BEGIN
    DECL X : INTEGER
    X X - X * X
   END.
```

```
PROGRAM foo(f1);
  BEGIN
   DECL X : INTEGER
   X := X - X * X
-----^[I]
  END.
```

```
PROGRAM foo(f1);
   BEGIN
```

```
PROGRAM foo(f1);
  BEGIN
  DECL DECL X : INTEGER
  X := X - X * X
 END.
```

```
  DECL  X : INTEGER
-------^[D]
  X := X - X * X
 END.
```

```
PROGRAM foo(f1);
   BEGIN
    DECL X : INTEGER
    Y := X - X * X
   END.
```

```
PROGRAM foo(f1);
  BEGIN
   DECL  X : INTEGER
   X := X - X * X
--^[R]
  END.
```

```
PROGRAM foo f1);
   BEGIN
    DECL X : : INTEGER
    X := X - Y * X
    END.
```

```
PROGRAM foo(f1);
--------------^[I]
  BEGIN
   DECL  X :  INTEGER
------------^[D]
   X := X - 1 * X
-----------^[R]
  END.
```

```
PROGRAM foo(f1);
   BEGIN
    DECL X : INTEGER
    DECL X : INTEGER
    X := X - Y * X
   END.
```

```
PROGRAM foo(f1);
  BEGIN
   DECL  X : INTEGER
   DECL  UnknownX : INTEGER
---------^[R]
   X := X - 1 * X
-----------^[R]
  END.
```

```
PROGRAM foo(f1);
   BEGIN
    DECL X : INTEGER
    DECL Y : REAL
    X := Y * X
   END.
```

```
PROGRAM foo(f1);
  BEGIN
   DECL  X : INTEGER
   DECL  Y : REAL
   X :=  1 * X
--------^[R]
  END.
```

```
PROGRA  foo(f1);
   BEGIN
    DECL Y : REAL
    EN.
```

```
PROGRAM foo(f1);
^[R]
   BEGIN
    DECL Y : REAL
  END.
--^[R]
```

```
PROGRAM foo
   BEGIN
    DECL X   INTEGER
    X 1;
    X X * X - 10;
    Y := X + Y
   END END.
```

```
PROGRAM foo(unknownid);
           ^^^^^^^^^^
   BEGIN
    DECL X : INTEGER
           ^
    X := 1;
      ^^
    X := X * X - 10;
      ^^
    X := X + 1
    ^         ^
  END .
      ^
```

Bibliography

[AP72] Alfred V. Aho and Thomas G. Peterson , "A Minimum Distance Error Correcting Parser For Context-Free languages ," *IAM Journal of Computing* 1, 4, pp. 305-312 (1972).

[AU72] Alfred V. Aho and Jeffrey D. Ullman, *The Theory of Parsing, Translation and Compiling*, Prentice-Hall (1972) Englewood Cliffs, N. J.

[BC79] Roland C. Backhouse, *Syntax of Programming Languages, Theory and Practice*, Printice-hall (1979).

[BO76] G. V. Bochmann, "Semantic Evaluation from Left to Right," *Communication of ACM*, 19, 2, pp. 55-62 (1976).

[CA80] R. G. G. Cattell, "Automatic Derivation Of Code Generator From Machine Descriptions," *ACM Transactions on Programming Languages and Systems* 2,2, pp 173-190 (April 1980).

[DE77] A. Deremer, "A Generalized Left Corner Parsing," *Conference Record Of The Fourth Annual Symposium On Principle Of* Programming Languages, Los Angeles CA (Jan 1977), pp 170-181.

[DI78] Bernard A. Dion, "Locally Least-Cost Error Correctors for Context-Free and Context-Sensitive Parsers ," Tech. Report #344, University of Wisconsin-Madison (December 1978) Ph.D. thesis.

[EA70] J. Earley, "An Efficient Context-Free Parsing Algorithm ," *Communication of the ACM* 13, 2, pp. 94-102 (1970).

[FA72] I. Fang, "FOLDS, a Declarative Formal Language Definition System," Technical Report stan-72-329, Stanford University (1972).

[FD79] Charles N. Fischer and Bernard A. Dion, "On the Testing of Insert Correctability," Tech. Report #355, University of Wisconsin-Madison (June 1979).

[FDM79] Charles N. Fischer, Bernard A. Dion, and Jon Mauney, "A Locally Least-Cost LR Error-Corrector," Tech. Report #363, University of Wisconsin-Madison (August 1979).

[FM81] Charles N. Fischer and Jon Mauney, "On the Role of Error Productions in Syntactic Error Correction," *Computer Languages* 5, pp. 131-139 (1981).

[FMM79] Charles N. Fischer, Donn R. Milton, and Jon Mauney, "A Locally Least-Cost LL(1) Error Corrector ," Tech. Report #371, University of Wisconsin-Madison (October 1979).

[FMQ80] Charles N. Fischer, Donn R. Milton, and Sam B. Quiring, "Efficient LL(1) Error Correction and Recovery Using Only Insertions," *Acta Informatica* 13, 2, pp. 141-154 (1980).

[GA80] Mahadevan Ganapathi, "Retargetable Code Generation And Optimization Using Attribute Grammars," Tech. Report #408 University Of Wisconsin-Madison, Ph.D. thesis (December 1980).

[GG78] R. S. Glanville and Susan L. Graham, "A New Method for Compiler Code Generation," *Conference Record Of The Fifth Annual ACM Symposium on*

Principle Of Programming Languages Tucson, Arizona (1978).

[GHJ79] Susan L. Graham, Charles B. Haley, and William N. Joy, "Practical LR Error Recovery," *Sigplan Notices* 14, 8, pp.168-175 (1979), Proceedings of the Sigplan Symposium on Compiler Construction.

[GHR80] Susan L. Graham, Michael A. Harrison, and Walter L. Ruzzo, "An Improved Context-Free Recognizer," *ACM Transactions on Programming Languages and Systems* 2, 3, pp. 415-462 (July 1980).

[GR75] Susan L. Graham and Steven P. Rhodes, "Practical Syntactic Error Recovery," *Communications of the ACM* 18, 6, pp. 639-650 (June 1975).

[HD80] Patrick A. V. Hall and Geoff R. Dowling, "Approximate String Matching," *ACM Computing Surveys* 12, 4, pp. 361-380 (December 1980).

[IR63] E. T. Irons, "An Error-Correcting Parse Algorithm," *Communication of the ACM* 6, pp. 669-673 (1963).

[JA72] L. R. James, "A Syntax Directed Error Recovery Method," Tech. Report CSRG-13, Computer System Research Group, University of Toronto (may 1972) M.S. thesis.

[JP81] M. Jazayeri and Diane Pozefsky, "Space-Efficient Storage Management in an Attribute Grammar Evaluator," *ACM Transactions on Programming Languages and Systems* 3, 4, pp. 388-404 (Oct. 1981).

[JR82] C. W. Johnson and C. Runciman, " Semantic Errors - Diagnosis and Repair," *Sigplan Notices* 17, 6, pp 88-97 (1982), Proceedings of the Sigplan

'82 Symposium on Compiler Construction, Boston, Massachusetts.

[JW75] M. Jazayeri and K. G. Walter, "Alternating Semantic Evaluator," *ACM proceedings of the Annual Conference*, Minneapolis, MN, pp 230-234 (Oct. 1975).

[KA80] Takuya Katayama, "Translation of Attributed Grammar into Procedure," Technical Report CS-K8001, Department of Computer Science, Tokyo Institute Technology, Tokyo, Japan (July 1980).

[KN68] Donald E. Knuth, "Semantic of Context-free Languages," *Mathematical System Theory*, 2, pp. 127-145 (1968).

[KR79] Ken Kennedy and J. Ramanathan, "A Deterministic Attribute Grammar Evaluator Based on Dynamic Sequencing," *ACM Transactions on Programming Languages and Systems* 1, 1, pp. 142-160 (July 1979).

[KW78] Ken Kennedy and Scott K. Warren, "Automatic Generation Of Efficient Evaluators For Attributed Grammars," *Conference Record Of The Fifth Annual ACM Symposium On Principle Of Programming Languages*, Tucson,Arizona, pp 32-49 (1978).

[LE70] R. P. Leinius, "Error Detection and Recovery for Syntax Directed Compiler System," Ph.D. thesis (1970).

[LRS74] P. M. Lewis, D. J. Rosenkratz, and R. E. Stearns, "Attributed Translation," *Journal of Computer and Systems Sciences*, 9 pp. 297-307 (1974).

[MA82] Jon Mauney, "Error Correction Using Extended Right Context" University of Wisconsin-Madison (August 1982) Ph.D. thesis.

[MF79] Donn R. Milton And Charles N. Fischer, "LL(k) Parsing For Attributed Grammars," ICALP '79 *International Colloquium On Automata, Languages, and Programming* July 1979.

[MU78] Eva-Maria M. Muckstein, "A Natural Language Parser With Statistical Applications," Research Report, IBM T. J. Watson Research Center, Yorktown Heights, N.Y. 10598, February 1978.

[PA81] Frank G. Pagan, *Formal Specification of Programming Languages*, Printice-Hall (1981), Englewood Cliff, N.J.

[PK80] Ajit B. Pai and Richard B. Kieburtz, "Global Context Recovery: A New Strategy for Parser Recovery From Syntax Error," *ACM Transactions on Programming Languages and Systems* 2, 1, pp. 18-41 (January 1980).

[PD78] Thomas J. Pennello and Frank L. DeRemer, "A Forward Move Algorithm for LR Error Recovery," *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, pp. 241-254 (1978).

[RO77] Bruce Ramon Rowland, "Combining Parsing and Evaluation For attributed Grammars," Tech. Report #308 University of Wisconsin-Madison (November 1977) Ph. D. thesis.

[TA78] Kuo Chung Tai, "Syntactic error correction in programming languages," *IEEE Transactions on Software Engineering SE-4*, 5, pp.414-425 (1978).

[WA76] David A. Watt, "Irons' Error Recovery in (LL and) LR Parsers," Technical report #8, University of Glasgow (August 1976).

[WA77] David Anthony Watt, "The parsing Problem For Affix Grammars," *Acta Informatica* Springer-verlag 1977.

[WA79] David Anthony Watt, "An Extended Attribute Grammar For Pascal," *Sigplan Notices*, February 1979 pp 60-74.

[WE80] C. S. Wetherell, "Probabilistic Languages: A review and Some Open Questions," *ACM Computing Survey*, 12, 4, pp. 381-402 (December 1980).