SYNTACTIC SPECIFICATION AND ANALYSIS WITH ATTRIBUTED GRAMMARS

by

Donn Robert Milton

Computer Sciences Technical Report #304

August 1977

SYNTACTIC SPECIFICATION AND ANALYSIS WITH ATTRIBUTED GRAMMARS

BY

DONN ROBERT MILTON

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

DOCTOR OF PHILOSOPHY

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

1977

ABSTRACT

Attributed grammars have traditionally been used for the semantic specification of programming languages and for the implementation of the translation phase of compilers. We investigate attributed grammars as an efficiently parsable syntactic specification mechanism that can handle many of the non-context-free aspects of programming language syntax.

A formal definition of attributed grammars is provided, and the notion of an attributed derivation is examined. We identify a parsable class of attributed grammars, called strong ALL(k), and the corresponding parser is developed as an extension of the strong LL(k) technique. Algorithms are presented for testing the strong ALL(k) property and for generating the strong ALL(k) parser. Finally, a number of applications are considered, establishing ALL(k) grammars as an effective tool for handling context-sensitivity in programming languages, and for reducing the size of programming language grammars.

ACKNOWLEDGEMENTS

I am particularly grateful to my advisor, Professor Charles N. Fischer, for his aid and guidance in transforming an idea into a thesis.

Professor Marvin H. Solomon deserves special thanks for his dedicated reading of the drafts, and for suggesting numerous improvements.

My fellow graduate students have made a variety of contributions to this work -- I am indebted to Lee Kirchhoff, Bob Mitze, Sam Quiring, Bruce Rowland, and, especially, Carol Sanna. I would also like to thank Frank Horn for his meticulous proofreading.

Above all, I wish to express my gratitude to my parents, Leonard and Hilda Milton, for their encouragement, support, and implicit faith.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The specification of programming languages has long been an issue of major consequence for both language users and language implementors. Language users require a readable specification with the syntax and semantics clearly and completely defined, while language implementors desire a specification that will facilitate the automatic generation of compilers.

The ALGOL 60 Report ([Nau63]) popularized a specification method that has been widely adopted. The Report factors language definition into two parts: a formal (BNF) specification of the context-free syntax, and an informal specification of the semantics. A particular advantage of this factorization is that context-free grammars (with some restrictions) can easily be used to automatically generate efficient parsers. The difficulty is that there are many aspects of programming language syntax that cannot be specified with a context-free grammar. A language, for example, that requires variables to be declared before they are used, or insists on type-compatibility between the source and destination of an assignment, will require some context-sensitive

specification mechanism. Usually this kind of syntactic restriction has not been formalized at all, but has been included in the informal "semantic" description.

Of course, there is no universally accepted division between a programming language's syntax and its semantics. For the purposes of this thesis, we will consider as "syntax" those facets of language specification that determine membership in the language, and relegate to "semantics" those aspects which define translation. The syntax and semantics may thus overlap, and the delineation of the syntax will be heavily dependent on the definition of language membership. Syntax also embodies the notion that some form of grammatical or "surface" structure is being placed upon the source text, although distinguishing this from the purely semantic structure is problematical. A syntactic specification may be relatively simple if membership is defined completely by a context-free grammar, or it may be highly complex if context-sensitive restrictions need to be expressed. Membership may even be undecidable: the ALGOL 60 restriction that "the kind and type of each actual parameter be compatible with the kind and type of the corresponding formal parameter" has been shown by Langmaack ([Lan73]) to be undecidable when formal procedure parameters are taken into account. This thesis will investigate a technique that fully spans this range of syntactic specifications.

Context-free derivation trees, however, are particularly attractive to language implementors, since they are of proven value in organizing translations. Accordingly, efforts have been made to extend the power of context-free grammars. Examples include matrix grammars, time-varying grammars, and programmed grammars, (see [Sal73] for a survey of these and other forms of "regulated" context-free grammars). More recently, the notion of a two-level, or van Wijngaarden, grammar has been developed, and used in the specification of ALGOL 68 ([Wij75]). Although these devices generate the familiar context-free derivation trees, they all share the same two problems: they are of dubious utility to the language user, and of no utility to the language implementor. In fact, there is an important connection between the users' and implementors' criteria. One measure of the readability and understandability of a syntactic specification is the ease with which it may be determined (by humans) whether a given construct is syntactically correct. It is to be expected that such a determination will be facilitated by a specification form that lends itself to automatic parser implementation. However, the above techniques have been designed primarily as generative mechanisms. With respect to recognition, or parsing, there have been virtually no results. So while these forms of programming language specification define

how a program may be generated, they provide no algorithms for determining whether a given program is in the language.

The solution we propose to the problem of providing a readable and parsable programming language specification mechanism involves the use of attributed grammars. Attributed grammars were invented by Knuth ([Knu68]), as a "simple technique for specifying the 'meaning' of languages defined by context-free grammars." Previous work has explored the use of attributed grammars as a tool for semantic specification, and for the implementation of the translation phase of compilers. This thesis will examine their use in syntactic specification and automatic parser generation.

Chapter 2 presents a formal definition of attributed grammars and develops the notion of an attributed derivation. A parsable class of attributed grammars, called ALL(k) grammars, is defined in Chapter 3, and algorithms for parsing and for parser generation are given. Chapter 4 examines the application of ALL(k) grammars to a variety of problems in syntactic specification. Chapter 5 discusses the impact of this work on compiler construction, and suggests directions for future research.

The reader is assumed to be familiar with the theory of formal grammars. A short tutorial on LL parsing is presented in chapter 3, but some familiarity with context-free parsing is also expected. We will use terminology that is consistent with that of Hopcroft and Ullman ([HU69]) with respect to formal grammars, and with that of Aho and Ullman ([AU72]) with respect to parsing.

CHAPTER 2

ATTRIBUTED GRAMMARS

2.1. Introduction

Informally, an attributed grammar is a context-free grammar where attributes may be attached to each nonterminal and terminal symbol. An attribute is a (name,value) pair, and is used to associate auxiliary information with the construct represented by a grammatical symbol. For example, a programming language grammar may provide the nonterminal "EXPRESSION" with attributes to indicate its mode and its run-time address. The evaluation of attribute values is specified with attribute evaluation rules, which are supplied with each production. Each rule defines an attribute value as a function of other attribute values of symbols appearing in the same production.

The discovery of attributed grammars (for programming languages) is generally credited to Irons ([Iro61],[Iro63]), who employed what are now called synthesized attributes. The use of synthesized attributes for natural languages was developed independently by Katz and Fodor ([KF63]). A synthesized attribute of a symbol

is computed as a function of the attributes of its immediate descendants. Much of the current interest in attributed grammars is due to Knuth's invention of inherited attributes ([Knu68]). An inherited attribute of a symbol is a function of the attributes of parent or brother symbols, and can thus be used to convey context into a subtree in a derivation, while synthesized attributes could only convey information out of a subtree.

The next section will give a formal definition of attributed grammars. The definition was designed primarily to facilitate the definition of the language generated by an attributed grammar. This latter definition, and the associated concept of an attributed derivation, will be presented in section 2.3.

2.2 Formal Definition of Attributed Grammars

Our definition is not far removed from Knuth's original definition ([Knu68]), but is somewhat more explicit in its treatment of attribute evaluation. An important difference is that contextual predicates, which

determine the applicability of a production, have been factored from attribute evaluation rules. A secondary difference is that we have followed other authors ([LRS74]) in the inclusion of 'action' symbols.

Definition 2.2.1 An attributed grammar is an 11-tuple, $(N,T,K,A,V,Z,IS,P,R,C,F_K)$, where:

N is a finite set of nonterminal symbols.

T is a finite set of terminal symbols.

K is a finite set of action symbols.

A is a finite set of attribute names. A is given an arbitrary, but fixed ordering, $\{a_1,\ldots,a_m\}$.

V is a finite set of attribute value sets, for each $a \in A$ there corresponds a value set V_a (not necessarily finite), and $V = \{V_{a_1},\ldots,V_{a_m}\}$, where $A = \{a_1,\ldots,a_m\}$.

Z is the start symbol, $\in N$, and does not appear on the right-hand side of any production in P.

IS is a pair of <u>attribute selection functions</u>, (i,s).

i,s: $N \cup T \cup K \dashrightarrow 2^A$ such that for each $X \in N \cup T \cup K$, $i(X) \quad s(X) = \emptyset$. Also, for $X \in T \cup \{Z\}$, $i(X) = \emptyset$. Members of $i(X)$ are <u>inherited attributes</u> of X, and members of $s(X)$ are <u>synthesized attributes</u>. The ordering of A induces an ordering on the attributes of X.

As a notational device:

$$IS(X) = i(X) \cup s(X) = \{a_{j_1},\ldots,a_{j_n}\}.$$

P is a finite sequence of <u>productions</u> of the form $X \dashrightarrow w$, where $X \in N$, $w \in (N \cup T \cup K - \{Z\})^*$.

R is a finite set of <u>attribute evaluation rules</u>, r_{ijk}. Let the ith production be:

$$X_{io} \dashrightarrow X_{i1} \cdots X_{in}$$

where $IS(X_{ij}) = \{a_1,\ldots,a_m\}$. (Note that $n = n(i)$ and $m = m(i,j)$). Let $X(i,j).a_k$ denote an <u>attribute instance</u>, that is, the specific presence of attribute a_k as an attribute of the jth symbol of the ith production.

Define a tuple $(h_1,\ldots,h_t)$ where each h is an attribute instance of production i, and $t = t(i,j,k)$. Then:

$$r_{ijk} = (X(i,j).a_k \dashleftarrow f_{ijk}(h_1,\ldots,h_t))$$

where $f_{ijk}: V_{a_1} \times \ldots \times V_{a_t} \dashrightarrow V_{a_k}$ is total

recursive and each a_i is the attribute name of which h_i is an instance.

There is a rule r_{ijk} exactly when i,j,k satisfy:

a) $a_k \in s(X_{i\emptyset})$, $(j = \emptyset)$;

<u>or</u> b) $a_k \in i(X_{ij})$, and $j > \emptyset$.

C is a finite set of <u>contextual predicate</u> sets, C_i, $1 \leq i \leq |P|$. Each C_i consists of a finite number of predicates, C_{ij}. Each C_{ij} is a total recursive predicate on a tuple of attribute instances $(g_1,\ldots,g_c)$, where each g_k is an attribute instance of production i and $c = c(i,j)$.

$$C_{ij}: V_{a_1} \times \ldots \times V_{a_c} \dashrightarrow \{\underline{true},\underline{false}\}$$

where a_k is the attribute name corresponding to the attribute instance g_k.

F_K is a finite set of <u>action functions</u>.

For each action symbol $X \in K$, let

$$i(X) = \{a_1,\ldots,a_n\}$$

$$s(X) = \{b_1,\ldots,b_m\}$$

then $f_X: V_{a_1} \times \ldots \times V_{a_n} \dashrightarrow V_{b_1} \times \ldots \times V_{b_m}$ is in F_K.

It is important to realize that we are considering attributes strictly as a means for increasing the expressive power of context-free grammars with respect to the specification of syntax. Traditionally, attributes have been used primarily to define semantics and translations of programming languages. We will not restrict the traditional usage of attributes, but will be concerned only with "syntactic" attributes. Thus, a prime consideration is to define attributed grammars so as to facilitate the definition of an attributed derivation, and to allow the construction of attributed parsers.

While these applications do require that the definition of attributed grammars be precise, we do not want to discourage their use as a practical tool for language specification. In fact, attributed grammars seem to be inherently more comprehensible than other forms of two-level grammars. The literature contains a number of relatively readable notations for presenting attributed grammars, at the end of this section we will present the notation that will be used for the remainder of this thesis.

An attributed grammar is based upon an underlying context-free grammar, which can be extracted trivially from the above definition. Ignoring the attributes, the context-free grammar is given by (N,T,Z,P). An attributed grammar is, of course, a kind of "two-level" grammar. The

second level is used to regulate the application of the productions of the first level, and here the second level consists of attributes, attribute evaluation functions, attributed action symbols, and contextual predicates. This use of contextual predicates to determine the validity of a production application is in contrast with Wilner's technique of providing each nonterminal with an 'error' attribute ([Wil71]). Although the methods are equivalent, the factorization of contextual predicates from attribute evaluation rules emphasizes their unique role in controlling attributed derivations.

IS defines the attributed vocabulary, associating an ordered set of attribute names with each terminal, nonterminal, and action symbol. The restrictions provided in the definition of the i and s functions assure that:

1) the association of attribute names is unique, (i.e., each symbol is associated with exactly one attribute name set);

2) each associated attribute name set is disjointly partitioned into sets of <u>inherited</u> and <u>synthesized</u> attributes.

For each attribute name 'a' there is a corresponding value set V_a identifying the values which an instance of 'a' may

possess. If the same attribute name is associated with more than one symbol, the same value set still corresponds. This facility enables the size of a grammar definition to be reduced, and improves readability: attributes used in similar ways by different symbols can still possess the same name.

The distinction between inherited and synthesized attributes is, syntactically speaking, in how they are used to convey context. This is most easily expressed in terms of a derivation tree: the inherited attributes of a node convey context from attributes of parent and sibling nodes to be used in the expansion of a subtree; the synthesized attributes of a node are used to return usable context from the subtree it roots. The distinguished symbol and all terminal symbols are not permitted to possess inherited attributes. This restriction is derived from the above concept of an inherited attribute: for the distinguished symbol, there are no parent or sibling nodes from which context may be conveyed; for terminal symbols, there is no subtree to which context may be conveyed.

The definition allows both attribute evaluation rules and action symbols. This is for flexibility and ease of expression. The readability of attributed grammars is greatly enhanced if non-trivial attribute evaluation rules are permitted, while it is easier to implement an attributed parser if they are prohibited ([LRS76]).

Fortunately, action symbols and attribute evaluation rules are completely interchangeable: an action symbol can always be replaced by an attribute evaluation rule, and an attribute evaluation rule can always be replaced by an action symbol and a set of <u>copy rules</u>. A copy rule is a special case of attribute evaluation rule where the associated evaluation function is the identity function.

The role of the contextual predicate sets is to determine the applicability of the productions. For each production there is an optional set of contextual predicates. The use of contextual predicates will be discussed in detail when attributed derivations are defined in section 2.3.

Notation for Attributed Grammars

The notation that will be used for the informal presentation of attributed grammars will be demonstrated with a small example. The vocabulary is specified first, and the inherited and synthesized attributes are given for each symbol. Along with each action symbol is provided a definition of the associated function. Action symbols will conventionally be enclosed in square brackets. The attribute value sets are given for each attribute name, and then any non-trivial attribute evaluation functions

are defined. Finally, the productions, along with their associated contextual predicates and attribute evaluation rules, are specified. Contextual predicates will be identified with the prefix 'CP'. The attribute instances (that is, the arguments of the contextual predicates and attribute evaluation rules) will be denoted with the form, X.a, where X is a symbol in the production and a an associated attribute name. Where a symbol X appears more than once in a production, X^i.a will denote an attribute instance of the ith occurrence.

The following grammar generates a language containing all arithmetic expressions involving the operator '+' and integer constants, where the value of the expression is less than 10. Integer constants will be represented by the symbol 'const', with a synthesized attribute containing the corresponding value.

Nonterminals

Z vs: synthesized
E vi: inherited; vs: synthesized

Terminals

const vs: synthesized

Actions

[add] v1,v2: inherited; vs: synthesized
(vs := v1 + v2)

Attribute Value Sets

vi,vs,v1,v2: {...,-2,-1,0,1,2,3,...}

Productions

1) Z --> const E

CP: (Z.vs < 10)

E.vi <-- const.vs

2) E^1 --> + const [add] E^2

[add].v1 <-- E^1.vi
[add].v2 <-- const.vs
E^2.vi <-- [add].vs
E^1.vs <-- E^2.vs

3) E --> ε

E.vs <-- E.vi

An example of a derivation using this grammar will be presented in the next section, after the concept of an attributed derivation has been defined.

2.3 Attributed Languages and Attributed Derivations

We now need to define the language generated by an attributed grammar, and the associated concept of an attributed derivation. But first we need a few more definitions:

Definition 2.3.1 An <u>evaluated instance</u> of a symbol X is a pair:

$$(X(i,j),\{(a_1,v_1),\ldots,(a_n,v_n)\})$$

denoting an occurrence of X as the jth symbol of the ith production with values assigned to all of its attributes. $X \in N \cup T \cup K$, $\{a_1,\ldots,a_n\} = IS(X)$, and $v_k \in V_k$ (where V_k is the value set associated with a_k).

Definition 2.3.2 A <u>partially evaluated instance</u> of a symbol X is an evaluated instance of X with each attribute value v_k chosen from the set $V_k \cup \{\perp\}$.

$\perp$ is a reserved value, not in any V_a, signifying "undefined". An attribute instance is considered to be <u>evaluated</u> if its value is not $\perp$. Notationally, a

(partially) evaluated instance will be denoted as $X(i,j)\langle v_{i_1},\ldots,v_{i_n};v_{s_1},\ldots,v_{s_m}\rangle$, where the v_i values correspond to the ordered list of inherited attributes, and the v_s values correspond to the ordered list of synthesized attributes. X" will be used to denote an arbitrary (partially) evaluated instance of X. This notation will be extended to strings: where $w = X_1\ldots X_n$, $w'' = X_1''\ldots X_n''$. X^I will denote the set of all partially evaluated instances of X with all inherited attributes evaluated and all synthesized attributes unevaluated. Similarly, X^S will denote the set of all partially evaluated instances of X with all synthesized attributes evaluated and all inherited attributes unevaluated. The notations I and S will also be extended to sets; thus, for example, where $N = \{X_1,\ldots,X_n\}$, $N^I = \{X_1^I,\ldots,X_n^I\}$.

We base our notion of an attributed derivation tree upon the usual definition of a derivation tree for a context-free grammar.

Definition 2.3.3 An (ordered) <u>tree</u> is a pair, (A,E), where A is a set of nodes and E is a set of linearly ordered lists of directed edges such that each element of E is of the form $((a,b_1),\ldots,(a,b_n))$, (where the b_i are all distinct), indicating that there are n edges leaving node a, and entering nodes $b_1,\ldots,b_n$. In addition, the following conditions must be satisfied:

a) there is exactly one node, the <u>root</u>, with indegree 0;

b) there is a directed path from the root to every other node;

c) except for the root, every node has indegree 1.

The set of all nodes to which there is a directed edge from a given node n, are the <u>direct descendants</u> of n. A node with no direct descendants is called a <u>leaf</u>.

A derivation tree labelled with evaluated symbol instances, according to the rules defined below, will be called an <u>attributed derivation tree</u>. An evaluated instance of a symbol X as the label of a specific node

will be called an <u>occurrence</u> of X, and each of its associated attribute instances will be called an <u>attribute occurrence</u>.

Definition 2.3.4 A tree is an <u>attributed derivation tree</u> of an attributed grammar G if:

a) every node is labelled with an evaluated symbol instance, or with ϵ;

b) the root is labelled with an evaluated instance of the distinguished symbol, Z;

c) all and only those nodes with outdegree 0 (leaves) are labelled as evaluated instances of terminal or action symbols, or with ϵ;

d) if the direct descendants $n_1,\ldots,n_k$ of a node n labelled with an evaluated instance of a nonterminal X, are themselves labelled with evaluated instances of symbols $X_1,\ldots,X_k$, then

$$X \dashrightarrow X_1 \ldots X_k$$

(called the <u>corresponding production</u> of node n) must be a production of P, (a node labelled with ϵ is permitted as the unique descendant of a node labelled with an evaluated instance of a nonterminal X only if $X \dashrightarrow \epsilon$ is in P);

e) each occurrence of an attribute instance, with which there is associated an attribute evaluation or action function, must possess the value obtained from the evaluation of that function on the corresponding occurrences of its arguments;

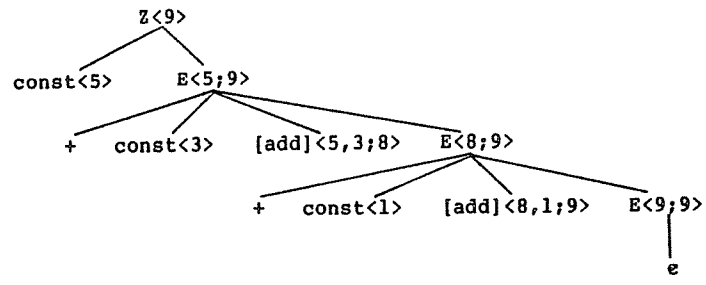
f) the values of the attribute occurrences of every node along with its direct descendants, must satisfy the contextual predicates associated with the corresponding production.

The <u>result</u> of an attributed derivation tree is the string obtained by concatenating (from left to right) the labels of those leaves labelled with evaluated instances of terminal symbols. Thus, strictly, the result consists of a string of evaluated instances of terminals. Informally, and where the meaning is clear, we will often consider the result to be a string of the terminal symbols themselves.

Definition 2.3.5 Given an attributed grammar G,

$$L(G) = \{w'' \mid w'' \text{ is the result of an attributed derivation tree of } G\}$$

Using the grammar presented at the end of the previous section we can provide an example of an attributed derivation tree:



While the definition of an attributed derivation parallels the definition of a derivation tree for context-free grammars, the concept of an attributed "derives" relation, ==>, corresponds more closely with context-sensitive derivations. For a reduced cfg $G = (N,T,P,Z)$, if $Z \Rightarrow^* w$, $w \in (N \cup T)^*$, then there exists an $x \in L(G)$ such that $w \Rightarrow^* x$. That is, context-free derivations allow any non-terminal in a sentential form to be rewritten independently of context. In addition, every sentential form of G is guaranteed to generate an element of L(G). This guarantee does not exist for context-sensitive derivations, where no production may be applicable to a sentential form, and a related situation is present in attributed derivations. There is, however, an important difference. A context-sensitive derivation terminates in failure precisely when no production is applicable to a sentential form which contains nonterminals. With an attributed sentential form, consisting of partially evaluated symbol instances, the application of a production may cause a contextual predicate to fail where that predicate is associated with a production applied in a <u>previous</u> derivation step. The mechanism of an attributed derivation, i.e., the definition of what constitutes a valid production application, will be in terms of partial derivation trees.

Definition 2.3.6 A tree is a <u>partial attributed derivation tree</u> of an attributed grammar G, if:

a) every node is labelled with a partially evaluated symbol instance, or with ϵ;

b) the root is labelled with a partially evaluated instance of the distinguished symbol, Z;

c) all nodes labelled with partially evaluated instances of terminal or action symbols, or with ϵ, are leaves;

d) if the direct descendants $n_1,\ldots,n_k$ of a node labelled with a partially evaluated instance of a nonterminal X, are themselves labelled with partially evaluated instances of symbols $X_1,\ldots,X_k$, then

$$X \rightarrow X_1 \ldots X_k$$

must be a production of P, (a node labelled with ϵ is permitted as the unique descendant of a node labelled with a partially evaluated instance of a nonterminal X only if $X \rightarrow \epsilon$ is in P;

e) each attribute occurrence must be evaluated if there is a corresponding attribute evaluation rule all of whose arguments are evaluated;

f) if all of the inherited attributes of an occurrence of an action symbol are evaluated, then all of its synthesized attributes must be evaluated;

g) each occurrence of an evaluated attribute instance, to which there is associated an attribute evaluation or action function, must possess the value obtained from the evaluation of that function on the corresponding occurrences of its arguments;

h) for each contextual predicate associated with a production corresponding to the expansion of a nonterminal node, if all of its arguments have been evaluated, then its value must be <u>true</u>.

Essentially, a partial attributed derivation tree represents a partial derivation where all attributes which <u>can</u> be evaluated <u>are</u> evaluated, and no contextual

predicates have been violated. The concatenation of the labels of the leaves of a partial attributed derivation tree will be called an <u>attributed sentential form</u>. Note that attributed action symbols are included. The notation $w'' = asf(t)$ will denote that w'' is the attributed sentential form defined by the (partial) attributed derivation tree t.

Attributed sentential forms are, however, not very meaningful entities. In abstracting the sentential form from the derivation tree, a considerable amount of necessary information is lost. All of the expanded nonterminal nodes, i.e., the interior nodes of the tree, along with their associated evaluated and unevaluated attributes have disappeared. In addition, there is no longer any way of determining what contextual predicates remain to be evaluated, and there is no way to evaluate their arguments, which may not even appear in the sentential form. For these reasons, the attributed derives relation, ==>, will be defined in terms of attributed derivation trees, and not in terms of attributed sentential forms.

Given a partial attributed derivation tree t, the application of a production

$$p_i = (X_{i0} \dashrightarrow X_{i1} \ldots X_{im}) \in P$$

is possible if there exists a leaf node n of t labelled with a (partially) evaluated instance of X_{i0}. The application involves the following steps:

1) add nodes $n_1,\ldots,n_m$ to t;

2) add the edge list $((n,n_1),\ldots,(n,n_m))$ to t;

3) label each new node n_j with a partially evaluated instance of X_{ij}: if $X_{ij} \in N \cup K$, assign $\perp$ to each of its associated attributes; if $X_{ij} \in T$, assign some $v_k \in V_k$ to each of its associated attributes a_k;

4) perform all possible attribute evaluations, as per Definition 2.3.6, parts e and f.

If the resultant tree t_2 is a partial attributed derivation tree (see Definition 2.3.6, part g), then the application of production p_i is considered valid and we may write $t_2 \in p_i(t_1)$.

Definition 2.3.7 Given an attributed grammar G, the attributed derives relation exists between two (partial) attributed derivation trees, $t_1 \Longrightarrow t_2$, if and only if there exists a $p_i \in P$ such that $t_2 \in p_i(t_1)$.

We can now give equivalent formulations for the language generated by an attributed grammar. First, define the result homomorphism "res" on attributed sentential forms:

a) $res(\epsilon) = \epsilon$

b) $res(X'') = X''$ if X" is a (partially) evaluated instance of $X \in N \cup T$

c) $res(X'') = \epsilon$ if X" is a (partially) evaluated instance of $X \in K$

res(w"), then, is simply w" with all of the action symbol instances erased.

Definition 2.3.8 Given an attributed grammar G, and the attributed derivation tree t_0 consisting of a single node labelled with an unevaluated instance of Z, then:

$$L(G) = \{res(asf(t)) \mid t_0 \Longrightarrow^+ t \text{ , where } t \text{ is an attributed derivation tree}\}$$

L(G) can be also expressed in terms of attributed sentential forms. There is, however, a complication introduced by the potential circularity of attributed grammars. Given a partial attributed derivation tree t of an attributed grammar G, we can create a directed graph D(t) whose nodes are the attribute occurrences in t. An arc exists from an attribute occurrence h_1 to an occurrence h_2 if and only if h_1 is an argument of the attribute evaluation rule for h_2. G will be called <u>circular</u> if and only if there exists a partial attributed derivation tree t (formed by ignoring all contextual predicates) such that D(t) contains an oriented cycle. The existence of an attributed derivation tree t implies that D(t) contains no oriented cycles, since all of the attributes of t are evaluated. It is possible, however, for unevaluated attributes occurrences to remain on a partial attributed derivation tree t with no nonterminal leaves, if D(t) contains an oriented cycle. Since

attributed sentential forms do not indicate which nonterminal attribute occurrences have been evaluated in the corresponding tree, we will insist for the remainder of this thesis that attributed grammars be non-circular. This will guarantee the evaluability of all attributes in all derivation trees with no nonterminal leaves. Circularity has been shown by Knuth to be a testable property ([Knu68]).

Definition 2.3.9 Given two attributed sentential forms, w_1'' and w_2'', of an attributed grammar G, define $w_1'' \Longrightarrow w_2''$ if and only if there exist two (partial) attributed derivation trees t_1 and t_2 such that $w_1'' = asf(t_1)$, $w_2'' = asf(t_2)$, and $t_1 \Longrightarrow t_2$.

Let Z" denote the unevaluated instance of Z.

Definition 2.3.10 Given an attributed grammar G:

$$L(G) = \{res(w'') \mid Z'' \Longrightarrow^+ w'' \text{ and } res(w'') = X_1''\ldots X_m'' \text{ where each } X_i'' \text{ is in } T^S\}$$

Definitions 2.3.8 and 2.3.10 are merely restatements of definition 2.3.5 in terms of derivation sequences, on trees and sentential forms respectively. There is an

important difference, however, between the derives relation defined on trees and the similar relation defined on sentential forms: the first is effective and the second is not. For any pair of (partial) attributed derivation trees t_1 and t_2, it is decidable whether there exists a valid production application such that $t_2 \in p_i(t_1)$: simply attempt to apply each production on every nonterminal leaf node of t_1. (Note that the insistence that attribute evaluation and action functions be total recursive is necessary for decidability). For an attributed sentential form w", however, it is first necessary to decide if there exists a derivation tree t such that w" = asf(t), and this is clearly equivalent to the membership problem for attributed grammars. Unfortunately, this problem is algorithmically unsolvable, as the following theorem will show.

Theorem 2.3.11 The class of languages defined by attributed grammars is the class of type-0 languages.

Proof: That every language defined by an attributed grammar is a type-0 language follows immediately from the equivalence of type-0 languages with Turing machine languages ([Cho59]), and Turing's thesis ([Tur36]). To show that every type-0 language can be defined by an attributed grammar, we will construct an attributed

grammar G that generates the same language accepted by a deterministic Turing machine with a two-way infinite tape. Let $M = (Q,T,H,d,q_0,F)$, where Q is the state set, T the input alphabet, H the tape alphabet, d the next-move function -- $d: Q \times H \rightarrow Q \times H \times \{L,R\}$, q_0 the initial state, and F the set of final states. We designate a special "blank" symbol, 'B', in H but not in T. G will operate by nondeterministically generating a string w in T*, and then simulating the operation of M on w. Productions 2 and 3 will generate w as an attribute of the nonterminal 'R', and production 1 will transfer w to an attribute of the nonterminal 'TM'. 'TM' will have three inherited attributes which embody a configuration of the Turing machine: <u>state</u>, identifying the state; <u>ltape</u>, representing the contents of the tape to the left of the tape head; and <u>rtape</u>, representing the contents of the tape directly under and to the right of the tape head. To simplify the construction, ltape and rtape will possess as values infinite strings: $\bar{B}$ will denote the infinite string of blanks. (It would be a straightforward modification to delimit the "input" with endmarkers and generate blanks when required.) The production, TM --> TM, will be applied for every move made by M, with attributes evaluated accordingly. If a final state is reached, the production, TM --> ϵ, will

be applied, completing the derivation. We construct G as follows:

Nonterminals

Z
R ws: synthesized
TM state,ltape,rtape: inherited

Terminals

$a_1,\ldots,a_n$ (where $T = \{a_1,\ldots,a_n\}$)

Attribute Value Sets

ws: T^*
state: $\underline{Q}$
ltape: $\bar{B}\cdot T^*$
rtape: $T^*\cdot\bar{B}$

Attribute Evaluation Functions

remr(s): (s = '$s_1 s_2 s_3\ldots$' , where $s_i \in H$)
  remr := '$s_2 s_3\ldots$' ;

reml(s): (s = '$\ldots s_3 s_2 s_1$' , where $s_i \in H$)
  reml := '$\ldots s_3 s_2$' ;

last(s): (s = '$\ldots s_3 s_2 s_1$' , where $s_i \in H$)
  last := 's_1' ;

Productions

1) Z --> R TM

  TM.ltape <-- $\bar{B}$
  TM.rtape <-- R.ws cat $\bar{B}$
  TM.state <-- $q_\emptyset$

2) for each $a_i \in T$:

  R^1 --> a_i R^2

  R^1.ws <-- 'a_i' cat R^2.ws

3) R --> ϵ

  R.ws <-- ϵ

4) for each $d(q_i,a_j) = (q_k,a_r,R)$,
  (read, write, and move right):

  TM^1 --> TM^2

  CP: (TM^1.state = q_i and TM^1.rtape = '$a_j\ldots$')

  TM^2.state <-- q_k
  TM^2.ltape <-- TM^1.ltape cat 'a_r'
  TM^2.rtape <-- remr(TM^1.rtape)

5) for each $d(q_i,a_j) = (q_k,a_r,L)$,
  (read, write, and move left):

  TM^1 --> TM^2

  CP: (TM^1.state = q_i and TM^1.rtape = '$a_j\ldots$')

  TM^2.state <-- q_k
  TM^2.ltape <-- reml(TM^1.ltape)
  TM^2.rtape <-- last(TM^1.ltape) cat 'a_r'
    cat TM^1.rtape

6) TM --> ϵ

  CP: (TM.state = $q_f \in F$)

Q.E.D.

CHAPTER 3

ATTRIBUTED TOP-DOWN PARSING

3.1 Introduction

This chapter will be concerned with the recognition of languages defined by attributed grammars. One approach to this problem has been taken by Fang ([Fan72]), who used Earley's parsing algorithm ([Ear68]). Earley's algorithm generates all possible parses for ambiguous grammars, and Fang used a rudimentary form of contextual predicate (just on synthesized attributes) to select the correct parse of an ambiguous construct. Earley's algorithm, however, requires $O(n^3)$ time and $O(n^2)$ space, where n is the length of the input string. We are interested in deterministic parsers, which never need to "back-up", and which operate in linear time and space. The central idea that makes this possible for attributed grammars is to permit the parser to make use of attribute values in computing each parsing decision. We shall find that we are able to generate such a parser automatically from certain classes of attributed grammars.

Top-down processing has particular advantages from the point of view of attribute handling. These result

from the fact that a top-down parser constructs a derivation tree from left to right in a depth-first manner. At every point in the parse, there exists a partial derivation tree, and the attribute evaluation rules will serve to communicate context across this tree for use in further expansion. On the other hand, bottom-up parsers construct a derivation tree by piecing together forests of smaller trees. It is easy to see that in such a scheme the availability of inherited attributes to convey context to the parser becomes problematical. Thus, we will concentrate on attributed top-down parsing. Since we are interested in efficiency and practicality, the class of LL(k) grammars immediately presents itself: they permit the automatic generation of deterministic parsers that never back-up. The next section will present a short tutorial on LL(k) parsing.

It will be necessary to define the subclass of attributed grammars that will be susceptible to attributed LL parsing. This is accomplished in section 3.3 with the definition of ALL(k) grammars. Section 3.4 will present algorithms for testing grammars for the ALL(k) property, and for constructing the parser tables for the parser to be defined in section 3.5. The chapter will conclude with a proof of the correctness of the parsing algorithm and an investigation into its efficiency.

The previous chapter defined the language of an attributed grammar as a string of evaluated terminal symbol instances. This definition will continue to hold when we discuss the problem of recognizing such a language.

Typically, the "front-end" of a compiler is composed of a scanner and a parser. It is rarely the case that the parser operates by examining individually each character of the input string. Instead, the scanner is responsible for combining substrings into "tokens", and identifying each token as being a member of a token class. The scanner then passes the token class index (and possibly the actual token) to the parser. In conventional table-driven parsers, the grammar has been defined with its terminal symbols corresponding exactly with the token classes. The actual token is generally ignored by the parser, and preserved only for use by the "semantic" routines. The reason that it is useful to separate the recognizer into a scanner and a parser is that identification of tokens is a task usefully performed by a GSM (generalized sequential machine), which can be characterized by a set of regular expressions defining the token classes. The parser is thus relieved of a considerable amount of work that can be more efficiently performed by a much simpler kind of machine.

This factorization has a natural application to attributed parsing: the output of a scanner can be considered to be a sequence of evaluated terminal symbols. Of course, terminals are permitted to possess only synthesized attributes, so typically there will be only one attribute, whose value will be the token itself or some encoding thereof. The terminal symbol will represent the corresponding token class.

3.2. Introduction to LL Parsing

LL(k) grammars are capable of describing a large class of "naturally" left-parsable languages. They were first described by Foster ([Fos68]), and the theory was further developed by Lewis and Stearns ([LS68]), Rosenkrantz and Stearns ([RS70]), and Knuth ([Knu71]). Given an LL(k) grammar it is possible to construct a deterministic top-down parser. Top-down processing presents a number of attractive features to the compiler writer, all a result of knowing what production is to be applied before its components are actually processed. One

must, however, be fortunate enough to possess an LL(k) grammar for the language in question, and LL(k) grammars are somewhat restrictive (compared with LR(1) grammars) in the class of languages definable. This situation is exacerbated by the fact that in practice only LL(1) grammars are ever used, and LL(1) grammars have great difficulty expressing certain common programming language constructs. A major goal of this thesis is to eliminate these problems through the definition of attributed LL(k) grammars, which will not only facilitate the expression of context-free syntax but will handle the context-sensitive aspects of language definition as well. But first we will give a short introduction to basic LL(k) parsing, closely following the treatment by Aho and Ullman ([AU72]).

Intuitively, an LL(k) parser is able to work in the following fashion: given a left-sentential form wAy which occurs during a parse of the terminal string wx, the parser can always decide which production to use to expand A knowing only w and the first k symbols of x. The <u>sequence</u> of left-sentential forms that occurs during a parse will constitute a <u>leftmost</u> <u>derivation</u>. That is, at each derivation step in the sequence, y ==> z, the <u>leftmost</u> nonterminal in y is rewritten to obtain z. Throughout our discussion of top-down parsing, all derivations will be assumed to be leftmost derivations.

Formally, we need some definitions:

Definition 3.2.1 For a CFG G = (N,T,P,S), given an integer k and $x \in (N \cup T)^*$:

$$FIRST_k(x) = \{w \in T^* \mid x \Rightarrow^* wz \text{ and } |w| = k \text{ or } x \Rightarrow^* w \text{ and } |w| < k\}.$$

Definition 3.2.2 For a CFG G = (N,T,P,S), given an integer k and $v,x,z \in (N \cup T)^*$:

$$FOLLOW_k(x) = \{w \mid S \Rightarrow^* vxz \text{ and } w \in FIRST_k(z)\}.$$

Definition 3.2.3 Let G = (N,T,P,S) be a CFG. Then G is LL(k) for some fixed k, if for every pair of leftmost derivations:

(a) $S \Rightarrow^* wAz \Rightarrow wuz \Rightarrow^* wx$

(b) $S \Rightarrow^* wAz \Rightarrow wvz \Rightarrow^* wy$

such that $FIRST_k(x) = FIRST_k(y)$, it follows that $u = v$, ($w \in T^*$, $A \in N$, $u,v,z \in (N \cup T)^*$).

We also need some simple theorems, all due to Lewis, Rosenkrantz, and Stearns, [LS68] and [RS70]:

Theorem 3.2.4 Every LL(k) grammar is unambiguous.

Theorem 3.2.5 A CFG G is LL(k) if and only if given any pair of distinct productions in G, A --> x and A --> y, $FIRST_k(xz) \cap FIRST_k(yz) = \emptyset$ for all wAz such that S ==>* wAz, ($w \in T^*$, $x,y,z \in (N \cup T)^*$).

Theorem 3.2.6 Given a CFG G and an integer k, it is decidable whether G is LL(k).

Definition 3.2.7 An LL(k) grammar G is <u>strong</u> LL(k) if and only if for any pair of productions A --> x and A--> y we have:

$$FIRST_k(x \cdot FOLLOW_k(A)) \cap FIRST_k(y \cdot FOLLOW_k(A)) = \emptyset$$

Theorem 3.2.8 Every LL(1) grammar is strong LL(1).

Theorem 3.2.9 Given an LL(k) grammar, one can construct a strong LL(k) grammar which generates the same language.

Given the above, we can restrict our attention to a k-predictive parsing algorithm that can be constructed from a strong LL(k) grammar. The LL(k) parsing algorithm utilizes a pushdown stack with stack alphabet $N \cup T \cup \{\$\}$, ($\$ \notin N \cup T$), and a parsing table M, which is a mapping

from $(N \cup T \cup \{\$\}) \times (T \cup \{\$\})^k$ to a set containing the following elements:

(a) <u>predict</u> i, where i indexes a production in P;

(b) <u>pop</u>;

(c) <u>accept</u>;

(d) <u>error</u>.

A configuration of the parsing algorithm will be a triple, (x,Xw,m), where x is the unused portion of the input string, Xw is the contents of the stack, with X on top, and m is a string of production indices. m is initially ε, upon acceptance m will constitute the leftmost parse of the input. Let $u = FIRST_k(x)$. The stack is initialized with S$, and the input string is terminated with the endmarker, $\$^k$. The algorithm proceeds as follows:

Algorithm 3.2.10

(a) $(x,Xw,m) \vdash (x,zw,mi)$ if $M(X,u)$ = <u>predict</u> i and production i is (X ==> z);

(b) $(x,aw,m) \vdash (x',w,m)$ if x = ax' and $M(a,u)$ = <u>pop</u>;

(c) if the algorithm reaches a configuration (x,Xw,m) where $M(X,u)$ = <u>accept</u>, then stop and accept;

(d) if the algorithm reaches a configuration (x,Xw,m) where M(X,u) = error, then stop and reject.

The algorithm is straightforward: step (a) replaces the non-terminal on top of the stack by one of its right-hand sides (i.e., a prediction), and step (b) matches and pops predicted terminals. It is a simple matter to embody this algorithm in a deterministic pushdown automaton (DPDA), by using the finite control to keep track of the k-symbol lookahead.

3.3 Grammars for Attributed LL Parsing

As was indirectly noted in the previous section, there are two approaches to the construction of an LL(k) parsing table. The strong approach computes, for a production X --> x, the lookahead set $FIRST_k(x \cdot FOLLOW_k(X))$, where $FOLLOW_k$ consists of all k-symbol terminal strings that may follow X in any sentential form. The "full" approach involves,

essentially, characterizing all of the right contexts in which a nonterminal may appear, and computing a distinct lookahead set for each right context class. The difference between strong and "full" LL(K) is only the precision with which the lookahead sets are computed. In either case, the parsing prediction function may be extended to consider the attributes of the symbols it processes. That is, given the symbol X on top of the parsing stack, and the k-symbol lookahead u, the prediction function can be a function not only of X and u but also of the evaluated attributes of X and u. The remainder of this section will be devoted to the definition of grammars susceptible to this form of "attributed" parsing.

Since LL parsing proceeds left-to-right and with no backup, we need to constrain the attribute evaluation rules so that the attribute values needed by the parser will be available at the right time. There is no problem with the attributes of the lookahead, since the lookahead consists entirely of terminal symbols whose attributes are constrained by definition to be synthesized (i.e., evaluated by the scanner). If the stack symbol is a terminal, there is also no problem: its synthesized attributes will assume the values of the attributes of the input symbol it matches. The attributes of a nonterminal stack symbol present a greater difficulty. The

synthesized attributes cannot be available to the parser, since these are computed as a function of righthand-side attributes, and the righthand-side has yet to be predicted. The inherited attributes <u>may</u> be available, but only if they are a function strictly of previously evaluated attributes. Since we are processing left-to-right, this implies that the available inherited attributes of a nonterminal instance will be a function only of its left context. That is, given a left sentential form wAy, where $w \in (T \cup K)^*$, $A \in N$, and $y \in (N \cup T \cup K)^*$, the inherited attributes of A will be available if and only if they are a function of the attributes of w.

Restricting all inherited attributes to depend only on their left context results in an <u>L-attributed</u> grammar, as defined by Lewis, Rosenkrantz, and Stearns ([LRS74]):

Definition 3.3.1 An attributed grammar is <u>L-attributed</u> when for each attribute evaluation rule:

$$r_{ijk} = (Y(i,j).a_k \text{ <-- } f_{ijk}(h_1,\ldots,h_t))$$

the following restrictions hold:

1) if $j > 0$, then

$$\{h_1,\ldots,h_t\} \subseteq \{X(i,u).a_v \mid 1 \le u < j,\ 1 \le v \le m(i,u)\} \cup \{X(i,0).a_v \mid 1 \le v \le m(i,0),\ a_v \in i(X_{i0})\}$$

2) if $j = 0$, then

$$\{h_1,\ldots,h_t\} \subseteq \{X(i,u).a_v \mid 1 \le u \le n(u),\ 1 \le v \le m(i,u)\} \cup \{X(i,0).a_v \mid 1 \le v \le m(i,0),\ a_v \in i(X_{i0})\}$$

Restriction 1 insists that an attribute evaluation rule for an (inherited) attribute of a righthand-side symbol can be a function only of inherited attributes of the lefthand-side symbol, plus arbitrary attributes of any symbol appearing to the left of the given symbol on the righthand-side. Restriction 2 requires that an attribute evaluation rule for a (synthesized) attribute of the lefthand-side symbol can be a function of the inherited attributes of the lefthand-side symbol, plus arbitrary attributes of any righthand-side symbol.

The L-attributed restrictions assure that the inherited attributes of a nonterminal stack symbol will have been evaluated by the time they are needed to guide the parser in making a prediction. For one-pass top-down parsing, we need the further restriction that the above inherited attributes along with the synthesized attributes of the lookahead will be sufficient to define a single-valued prediction function. Toward this end, we need the attributed counterpart of the FIRST function:

Definition 3.3.2 For an attributed grammar G, with the underlying context-free grammar $\bar{G}$, and $x \in (N \cup T)^*$,

$$\mathrm{AFIRST}_k^G(x) = \{w \mid w = X_1'' \ldots X_n'',\ X_1 \ldots X_n \in \mathrm{FIRST}_k^{\bar{G}}(x),\ \text{and each } X_i'' \text{ is an evaluated instance of } X_i\}$$

AFIRST_k is thus determined by computing all possible evaluations of the strings in FIRST_k. The notation $\bar{G}$ will be used consistently to denote the context-free grammar underlying the attributed grammar G. When $x \in (N \cup T \cup K)^*$ we define:

$$\mathrm{FIRST}_k(x) = \mathrm{FIRST}_k(\mathrm{res}(x)),$$

and $$\mathrm{FOLLOW}_k^G(x) = \mathrm{FOLLOW}_k^{\bar{G}}(\mathrm{res}(x)).$$

That is, action symbols are totally ignored.

It must be emphasized that AFIRST is based on derivations in the underlying context-free grammar. Such derivations, of course, will include the attributed derivations of the attributed grammar, and, in non-trivial cases, the inclusion will be proper. It would certainly be preferable to restrict AFIRST (and FOLLOW, which will be used below) to only the attributed derivations, but this cannot be done. The attributed derivations being considered here are not complete: they actually represent subtrees of possible attributed derivation trees. As

such, they cannot be taken out of context: we would have to consider the undecidable question, "does there exist an attributed derivation tree which contains a given subtree?" To avoid this difficulty, we have defined AFIRST as above, and recognize that it will contain many strings that could not actually occur in an attributed derivation.

The parsing table for the underlying context-free grammar $\bar{G}$ ignores all attributes. If $\bar{G}$ is strong or "full" LL(k) there is no need to consider the attributes at all in making a prediction. In this case, attributes will only be needed for the evaluation of contextual predicates: whenever a contextual predicate evaluates to <u>false</u>, the parse will terminate in error.

A grammar is not strong ("full") LL(k) precisely when the k-symbol lookahead does not uniquely determine a prediction. It may be, however, that the inherited attributes of the top stack symbol along with the synthesized attributes of the lookahead will enable the parser to discriminate between a choice of predictions. Attributes used in this manner can serve to "disambiguate" the parse of an underlying grammar which is not LL(k), and underlying ambiguous grammars can be handled as well. They may also be used to construct a parser which uses a smaller lookahead than would otherwise be necessary. In order to integrate attributes into the parsing function, we will first define "disambiguating" predicates.

Definition 3.3.3 For each production $p_i = (X \dashrightarrow x)$ in an attributed grammar G, a <u>d^k-predicate</u> for p_i is a total recursive predicate:

$$d_i^k: \; X^I \times AFIRST_k(x \cdot FOLLOW_k(X)) \dashrightarrow \{\underline{true}, \underline{false}\}$$

The d^k-predicates will be used as follows: with the partially evaluated nonterminal X" on top of the parsing stack, and the attributed k-symbol lookahead u", production p_i may be predicted if $d_i^k(X'',u'') = \underline{true}$.

Definition 3.3.4 A <u>d^k-predicated</u> <u>attributed</u> <u>grammar</u> is a pair (G,D^k) where G is an attributed grammar, and D^k is a function that assigns to each $p_i \in P$ a predicate d_i^k.

In practice, for a given production, d^k-predicates need only be specified for those lookaheads which will cause an LL(k) conflict. If a d^k-predicate is not supplied for a production, a "default" predicate returning a constant <u>true</u> is assumed. If a d^k-predicate is supplied for only a subset of the possible lookahead strings, the predicate is assumed to be augmented to return <u>true</u> for the lookaheads not considered (see example 3.5.2). Before examining the implications of the above definition of the

d_i^k-predicates, we will first use them in defining ALL(k) grammars.

Definition 3.3.5 A <u>strong</u> <u>ALL(k)</u> <u>grammar</u> is a d^k-predicated attributed grammar such that for every pair of productions of the form:

$$p_1 = (X \dashrightarrow x_1)$$
$$p_2 = (X \dashrightarrow x_2)$$

if $u \in FIRST_k(x_1 \cdot FOLLOW_k(X)) \cap FIRST_k(x_2 \cdot FOLLOW_k(X))$

then for all $u'' \in u^S$, and all $X'' \in X^I$:

$$d_1(X'',u'') \; \underline{and} \; d_2(X'',u'') = \underline{false}$$

We need to examine the notion of an attributed derivation in the presence of d^k-predicates. An ALL(k) grammar is not simply an attributed grammar -- it is a pair (G,D^k), and the d^k-predicates form an intrinsic part of the grammatical specification. There may be strings in L(G) which are not accepted by an ALL(k) parser which recognizes $L((G,D^k))$. In some cases the d^k-predicates may be inferred from contextual predicates, but in many instances they impose restrictions which contextual predicates cannot naturally represent, (consider production 5 of example 3.5.2). The difficulty results from the fact that contextual predicates can only be

applied to attributes within a production, while d^k-predicates involve attributes of the lookahead. The non-inclusion of d^k-predicates in the original definition of an attributed grammar is motivated by the hypothesis that their form is inextricably linked to the particular parsing algorithm.

It is possible to construct an attributed grammar that will generate the language which we will define as $L((G,D^k))$. The construction involves adding attributes that will identify the set of terminal strings (of length k) that may be generated by (and follow) each nonterminal in a derivation.

We will be using a slightly modified version of the left-quotient operator, "\":

for $L \subseteq (T^S)^k$, $w'' \in (T^S)^*$,

1) if $|w| \geq k$, then $w''\backslash L = \{\epsilon\}$;

2) if $|w| < k$, then $w''\backslash L = \{v'' \mid w''v'' \in L\}$.

Definition 3.3.6 Given an ALL(k) grammar, (G,D^k), the <u>corresponding</u> attributed grammar, G', is constructed by modifying G as follows:

Add two new attributes to each $X \in N$: fi, inherited, and fs, synthesized, with values in $(T^S)^*$. Add the production, $Z' \dashrightarrow Z$, with the attribute evaluation rule: $Z.fi \leftarrow\!\!- (T^S)^k$. Then for each production,

$$p_t = (X_0 \dashrightarrow w_0 X_1 w_1 \ldots X_n w_n)$$

(where $X_i \in N$, $w_i \in T^*$), add the following attribute evaluation rules:

$$X_1.fi \leftarrow\!\!- \{AFIRST_k(u''(T^S)^k) \mid u'' \in w_0''\backslash X_0.fi \ \underline{\text{and}}\ d_t(X_0'', AFIRST_k(w_0''u'')) = \underline{\text{true}}\}$$

for $1 < i \leq n$,

$$X_i.fi \leftarrow\!\!- \{AFIRST_k(u''(T^S)^k) \mid u'' \in w_{i-1}''\backslash X_{i-1}.fs\}$$

if the righthand-side contains a nonterminal (that is, $n > 0$):

$$X_0.fs \leftarrow\!\!- \{AFIRST_k(u''(T^S)^k) \mid u'' \in w_n''\backslash X_n.fs\}$$

if the righthand-side does not contain a nonterminal (that is, $n = 0$):

$$X_0.fs \leftarrow\!\!- \{AFIRST_k(u''(T^S)^k) \mid u'' \in w_0''\backslash X_0.fi \ \underline{\text{and}}\ d_t(X_0'', AFIRST_k(w_0''u'')) = \underline{\text{true}}\}$$

Also add the following contextual predicates to C_t:

$$(AFIRST_k(w_0'') \in INIT(X_0.fi))$$

and for $0 < i \leq n$,

$$(AFIRST_k(w_i'') \in INIT(X_i.fs)) \quad .$$

We define $L((G,D^k)) = L(G')$.

The grammar G' is essentially the original grammar G, with some additional attributes, attribute evaluation rules, and contextual predicates to ensure that all derivations are "restricted" by the d^k-predicates. Each derivation

tree in G' is a derivation tree in G (ignoring the additional attributes and the augmenting production). However for a derivation tree in G to be in G', each step in the associated leftmost derivation, $w''A''y'' \Longrightarrow w''x''y''$, ($w \in (T \cup K)^*$, $A \in N$, $x,y \in (N \cup T \cup K)^*$), must have the following property:

for all $z \in (T \cup K)^*$ such that $x''y'' \Longrightarrow^* z''$, let $p_t = (A \rightarrow x)$, and let $u'' \in AFIRST_k(z'')$, then $d_t(A'',u'') = \underline{\text{true}}$.

We thus have the notion of a <u>disambiguated</u> leftmost derivation. We can show that for any ALL(k) grammar (G,D^k) there is only one disambiguated leftmost derivation of each $z'' \in L((G,D^k))$.

Theorem 3.3.7 No strong ALL(k) grammar is ambiguous.

Proof: Let $Z'' \Longrightarrow^+ z''_a$ and $Z'' \Longrightarrow^+ z''_b$ (where $z'' = res(z''_a) = res(z''_b)$) be two disambiguated leftmost derivations of $z'' \in L((G,D^k))$. Consider the first point in which the derivations differ: $w''A''y'' \Longrightarrow w''x''_a y''$ and $w''A''y'' \Longrightarrow w''x''_b y''$. Let $z''_a = w''z''_1$ and $z''_b = w''z''_2$. Then the first k symbols of $res(z''_1)$ must be the same as the first k symbols of $res(z''_b)$, call these u". But then for productions $p_a = (A \rightarrow x_a)$ and $p_b = (A \rightarrow x_b)$ we have:

1) $u \in FIRST_k(x_a \cdot FOLLOW_k(A))$ and $d_a(A'',u'') = \underline{\text{true}}$

and

2) $u \in FIRST_k(x_b \cdot FOLLOW_k(A))$ and $d_b(A'',u'') = \underline{\text{true}}$

in violation of definition 3.3.5. <u>Q.E.D.</u>

Theorem 3.3.8 The class of languages defined by strong ALL(1) grammars is the class of type-Ø languages.

Proof: The grammar constructed in the proof of Theorem 2.3.11 can be transformed into an ALL(1) grammar by changing each contextual predicate into a d^1-predicate of exactly the same form. <u>Q.E.D.</u>

3.4 Strong ALL(k) Testing and Parser Table Construction

The strong ALL(k) condition asks whether given pairs of d^k-predicates can return <u>true</u> for identical arguments. Without restrictions on the predicates, this question is undecidable.

It is likely that, in practice, the d^k-predicates could be expressed in the first-order predicate calculus with equality. Over countably-infinite domains the

first-order predicate calculus is undecidable. Over finite domains the first-order predicate calculus can be reduced to the propositional calculus, for which satisfiability is decidable, but of NP-complete complexity ([Coo71]). Therefore, the cardinality of the domain of attribute values must be considered, keeping in mind that we are only concerned with attributes that will play a role in disambiguating the LL(k) prediction function. Unfortunately, empirical evidence on this point is lacking. It is of only formal interest whether a given attributed grammar could be rewritten to reduce the size of the attribute value domain. For a programming language grammar, wholesale rewriting cannot be considered, since such a grammar is designed with the primary criterion being its utility in guiding a translation. In virtually all of the examples we have examined, we have found that the particular attributes used by the d^1-predicates have a small finite domain, while many other attributes (such as those used by the contextual predicates) have infinite domains. It appears that restricting d^k-predicates to take as arguments only attributes with small domains will not unduly restrict the practical applicability of ALL(k) grammars.

If, however, it is found that infinite (or impractically large) attribute value sets are required, there are two possible approaches. First, it is likely

that the predicates will be simple enough to allow their disjointness to be verified by a heuristic mechanical theorem prover (see, for example, [Nil71]). Barring this solution, the parser generator could at least supply to the grammar designer a list of the theorems to be proved that will establish the validity of the parser.

If attribute value sets are finite, there is no need for the d^k-predicates to be specified in a formal language. We need only require that the predicates be supplied (along with the grammar to be tested) in a form such that they may be conveniently evaluated independently of an actual parse. In addition, they must be total and recursive over their given domains. Under these conditions, an algorithm for strong ALL(k) testing follows immediately from definition 3.3.5:

Algorithm 3.4.1

Input: an L-attributed d^k-predicated attributed grammar (G,D^k) where each attribute that is an argument of a d^k-predicate has a finite attribute value set.

Output: <u>true</u> if (G,D^k) is ALL(k), <u>false</u> otherwise.

```
for each nonterminal A
  do for each pair of distinct A-productions,
     A --> x1 and A --> x2
       do
         begin
           U := FIRST_k(x1 · FOLLOW_k(A))
                  ∩ FIRST_k(x2 · FOLLOW_k(A));
           for each A" ∈ A^I
             do for each u" ∈ U^S
                  do if d_1^k(A",u") and d_2^k(A",u")
                       then return false
         end;
return true;
```

The efficiency of this algorithm can be calculated quite simply. Let m bound the number of A-productions for any nonterminal A, let v bound the size of each attribute value set, and let n bound the number of attributes per symbol. Then the execution time of algorithm 3.5.1 is

$O((|P|/m)\cdot\binom{m}{2}\cdot(v^n)(|T|\cdot v^n)^k)$, which reduces to $O(|P|\cdot m\cdot|T|^k v^{nk+n})$. This bound is acceptable when we consider the parameters which will be encountered for typical programming language grammars. Taking m to be a small constant, and k = 1, we get a bound of $O(|P|\cdot|T|\cdot v^{2n})$, which can still be considered a very gross upper bound since $|U^S| << |T|\cdot v^{2n}$.

The parsing table M to be utilized by the ALL(k) parser is defined analogously to the table described in section 3.2, with the addition of attributes. Thus M is a mapping from $(N^I \cup T \cup \{\$\}) \times (T^S \cup \{\$\})^k$ to a set containing the following elements:

(a) <u>predict</u> i, where i indexes a production in P;

(b) <u>pop</u>;

(c) <u>accept</u>;

(d) <u>error</u>.

The attributes included in the arguments to M are used only by the disambiguating predicates. If the domains of these attributes are infinite, then M cannot be

directly expressed as a table of finite size. However, even if the domains are finite, the size of the table may still be too large to be considered practical. We will therefore use a technique for implementing the parser table that requires "parse-time" evaluation of the disambiguating predicates.

First, we need an (unattributed) strong LL(k) parsing table, modified to allow multiply defined entries. The following algorithm is similar to algorithm 5.3 in [AU72]:

Algorithm 3.4.2 Given an augmented context-free grammar $G = (N \cup \{\bar{Z}\}, T \cup \{\$\}, P \cup \{\bar{Z} \text{ --> } Z\$^k\}, \bar{Z})$, construct a (modified) strong LL(k) parser table, M':

```
(initialize entries to ∅)
for each A ∈ N
  do for each u ∈ (T ∪ {$})^k
       do M'(A,u) := ∅;

(compute the predict entries)
for each production p_i = (A --> x) ∈ P
  do for each u ∈ FIRST_k(x · FOLLOW_k(A))
       do M'(A,u) := M'(A,u) ∪ {predict i};
```



```
(fill in pop entries)
for each av ∈ T (T ∪ {$})^(k-1)
  do M'(a,av) := {pop};

(fill in accept entry)
M'($,$^k) := {accept};

(change empty entries to error's)
for each A ∈ N
  do for each u ∈ (T ∪ {$})^k
       do if M'(A,u) = ∅
            then M'(A,u) := {error};
```

To implement the ALL(k) parsing function M for an ALL(k) grammar G, first construct M' for the underlying context-free grammar. Then M may be derived as follows:

Algorithm 3.4.3 Given an ALL(k) grammar (G,D^k), and M', the modified strong LL(k) table for $\bar{G}$, compute M(A",u") for A" $\in N^I \cup T \cup \{\$\}$ and u" $\in (T^S \cup \{\$\})^k$ as follows:

```
if M'(A,u) = {error} then M(A",u") := error
elif M'(A,u) = {pop} then M(A",u") := pop
elif M'(A,u) = {accept} then M(A",u") := accept
else
  begin
   M(A",u") := error;
    for each predict i ∈ M'(A,u)
      do if d_i(A",u") then M(A",u") := predict i
end;
```

When M' is single-valued, M assumes the same value (if a predict entry, only if the corresponding d^k-predicate is true). When M' is multiply defined (i.e., multiple predict entries), M calls on the d^k-predicates to choose the proper prediction. Since the ALL(k) property guarantees that at most one d^k-predicate can be true for a given (A",u"), the order of evaluation of the d^k-predicates is unimportant. Note that M is preset to error in case all of the d^k-predicates return false.

3.5 The ALL(k) Parser

The parser is embodied in a machine constructed from the grammar and the parsing function, M. There are a number of similarities between the machine we describe and the attributed pushdown machine described in [LRS74]. The primary differences are that we handle attribute instances in a much more explicit manner through the use of registers, and that the transition function, M, includes attributes among its arguments. The ALL(k) parser is essentially a deterministic pushdown automaton modified in the following ways:

1) associated with each stack symbol is an m-tuple of registers, where m is the largest number of attribute instances present in any production;
2) associated with each input (terminal) symbol is an n-tuple of registers, where n is the number of attributes associated with that symbol;

3) associated with each transition there may be a mapping between register tuples, and/or a contextual predicate evaluation on a register tuple.

Each stack symbol will be denoted by a pair, (i,j), and will represent a prediction (in the LL sense) of the remainder of production i after the first j symbols of the righthand-side have been processed. The registers associated with a stack symbol will contain the attribute values for the attribute instances of the associated production. For each production i there is a single one-one mapping between the attributed instances and the registers of the register tuple associated with each of the stack symbols (i,j), for all j, $\emptyset \leq j \leq n(i)$, (where n(i) is the length of the righthand-side). The registers associated with the input symbols will contain the values of the associated (synthesized) attributes. These values will be provided as part of the input, having been precomputed by the scanner (see section 3.1).

We will use the following notation in the description of the parser:

st_k: the kth stack symbol from the top of the stack, $st_\emptyset$ is on top, (st_k is an (i,j) pair);

Sym(st): the grammar symbol corresponding to st = (i,j), which is the $j+1^{st}$ symbol of the righthand-side of production i, (Sym(st) does not exist for j = n(i));

R(st): the register tuple associated with st;

a_i: the ith symbol currently in the lookahead;

$R(a_i)$: the register tuple associated with a_i;

$r(h_k)$: the value contained in the register (in R((i,j)), $\emptyset \leq j \leq n(i)$) associated with the attribute instance h_k of production i;

in(st): where st = (i,j), the tuple of registers (included in R(st)) associated with the inherited attribute instances of the $j+1^{st}$ symbol of the righthand-side of production i;

sy(st): as above, the tuple of registers associated with the synthesized attribute instances of the $j+1^{st}$ symbol of production i;

I(st): the tuple of registers associated with the inherited attribute instances of the lefthand-side of production i, (st = (i,j));

S(st): the tuple of registers associated with the synthesized attributed instances of the lefthand-side of production i.

We will consider the attributed grammar to be augmented with the production $\bar{Z}$ --> Z $\$^k$, which will have index Ø. The parsing function, M, is the same one defined in the previous section. The form given below for its arguments are merely an encoding of $A'' \in N^I$ and $u'' \in (T^S \cup \{\$\})^k$. The input is terminated with the endmarker $\$^k$.

Algorithm 3.5.1 The ALL(k) Parser

1. Push st = (Ø,Ø).

2. <u>Case</u> $M((Sym(st_\emptyset), in(st_\emptyset)), ((a_1, R(a_1)), \ldots, (a_k, R(a_k))))$ <u>of</u>

   <u>predict</u> i: push st = (i,Ø);
   $I(st_\emptyset)$ <-- $in(st_1)$;

   <u>pop</u>: $sy(st_\emptyset)$ <-- $R(a_1)$;
   advance input pointer;
   increment the j-component of $st_\emptyset$;
   (now, $st_\emptyset$ = (i,j+1))

   <u>accept</u>: halt, accepting;

   <u>error</u>: halt, rejecting.

3. <u>for</u> each attribute evaluation rule of production i ($st_\emptyset$ = (i,j)), h_p <-- $f(h_1,\ldots,h_q)$, where h_p is an attribute instance of the $j+1^{st}$ symbol of the righthand-side, (or, if j = n(i), a synthesized attribute instance of the lefthand-side),

   <u>do</u>
   $r(h_p)$ <-- $f(r(h_1),\ldots,r(h_q))$.

4. <u>for</u> each contextual predicate

$C(h_1,\ldots,h_k) \in C_i$
where all of $h_1,\ldots,h_k$ have been evaluated
do
if $C(r(h_1),\ldots r(h_k))$ = false
then halt, rejecting;

5. if $st_\emptyset = (i,n(i))$ (where n(i) is the length of the righthand-side of production i)
then
begin
$sy(st_1)$ <-- $S(st_\emptyset)$;
pop;
replace (the new) $st_\emptyset = (i,j)$ with $(i,j+1)$;
goto step 3
end.

6. if $Sym(st_\emptyset) = X$, an action symbol where $(g_1,\ldots,g_p) = f_X(h_1,\ldots,h_q)$,
then:
$(r(g_1),\ldots r(g_p))$ <-- $f_X(r(h_1),\ldots,r(h_q))$;
replace $st_\emptyset = (i,j)$ with $(i,j+1)$;
goto step 3
end
else goto step 2.

The operation of the ALL(k) parser closely parallels that of the LL(k) parser, with additional machinery to handle the attributes. Whenever a parsing action (push or pop) occurs, step 3 is executed to evaluate any attribute instances (of the production currently being expanded) which are now evaluable (in the L-attributed sense). Step 4 evaluates the contextual predicates (of the current production) all of whose arguments have been evaluated. Step 5 checks to see if a production has been completely expanded. If so, the synthesized attribute values of the lefthand-side are copied to the appropriate registers of the previous stack symbol, and a pop is executed. This represents the return of synthesized attributes from a subtree. Step 6 executes action symbols when necessary and pops them.

The following example will illustrate the operation of an ALL(1) parser. The language represented by this grammar includes all expressions involving constants (integers), addition, and multiplication, such that the resulting value is less than 100.

Example 3.5.2

Nonterminals

Z
E pe: inherited; vs:synthesized
T p,vi: inherited; vs: synthesized

Terminals

const vs: synthesized
op p: synthesized

Actions

[xqt] v1,v2,p: inherited; vs: synthesized
(if p = 2 then vs := v1 + v2 else vs := v1 * v2)

Attribute Value Sets

pe: {1,2,3} p: {2,3}
vi,vs,v1,v2: {...,-2,-1,0,1,2,3,...}

Productions

1) Z --> E

CP: (E.vs < 100)

E.pe <-- 1

2) E --> const

d(E",const"): (E.pe = 3)

E.vs <-- const.vs

3) E^1 --> E^2 T

d(E^1",const"): (E^1.pe ≠ 3)

E^2.pe <-- E^1.pe + 1 T.vi <-- E^2.vs
T.p <-- E^1.pe + 1 E^1.vs <-- T.vs

4) T^1 --> op E [xqt] T^2

d(T^1",op"): (T^1.p = op.p)

E.pe <-- T^1.p T^2.p <-- T^1.p
[xqt].p <-- op.p T^2.vi <-- [xqt].vs
[xqt].v1 <-- T^1.vi T^1.vs <-- T^2.vs
[xqt].v2 <-- E.vs

5) T --> ε

d(T",op"): (T.p ≠ op.p)

T.vs <-- T.vi

We adopt the convention that op<2> will indicate "+" and op<3> will indicate "*". Thus the "p" attributes of E, T, and op are used by the d-predicates to ensure that the precedence of the operators is preserved, (a more elaborate example based on this idea is presented in

section 5.2). The "v" attributes are all used to convey values around the derivation tree: the contextual predicate attached to the first production checks that the final value is less than 100. The action "[xqt]" is responsible for performing the operation appropriate to the "p" attribute it receives from the associated "op".

The unattributed parsing table, M', appears below. Note that since the underlying grammar is not LL(1), the entries for M'(T,op) and M'(E,const) will need to rely on the d-predicates for disambiguating.

	op	const	$
Z		1	
E		2,3	
T	4,5		5
op	pop		
const		pop	
$			accept

The grammar is ALL(1) since it is L-attributed and an exhaustive examination of the d-predicates reveals no conflicts:

d_2(E<1;⊥>,const<->) = false
d_3(E<1;⊥>,const<->) = true

d_2(E<2;⊥>,const<->) = false
d_3(E<2;⊥>,const<->) = true

d_2(E<3;⊥>,const<->) = true
d_3(E<3;⊥>,const<->) = false

d_4(T<2,-;⊥>,op<2>) = true
d_5(T<2,-;⊥>,op<2>) = false

d_4(T<2,-;⊥>,op<3>) = false
d_5(T<2,-;⊥>,op<3>) = true

d_4(T<3,-;⊥>,op<2>) = false
d_5(T<3,-;⊥>,op<2>) = true

d_4(T<3,-;⊥>,op<3>) = true
d_5(T<3,-;⊥>,op<3>) = false

We will trace the execution of the parser on the input string, "5+2*4", which, after scanner preprocessing, will appear as:

const<5> op<2> const<2> op<3> const<4> $.

The trace will be a sequence of snapshots of the parsing stack. Each level of the stack, i.e., an (i,j) entry and its associated attribute registers, will be represented by writing out production i with its symbols replaced by their corresponding (partially) evaluated instances. All righthand-side symbols to the left of the jth symbol will be underlined. Most of the steps in the trace will indicate the state of the stack just before a parse operation (a predict or a pop in step 2 of the algorithm). In a few cases, intermediate stack snapshots are given to indicate the progress of attribute evaluation.

lookahead = const<5>

```
1.  Z'          -->  Z  $

2.  Z           -->  E<1;⊥>
    Z'          -->  Z  $

3.  E<1;⊥>      -->  E<2;⊥>  T<2,⊥;⊥>
    Z           -->  E<1;⊥>
    Z'          -->  Z  $

4.  E<2;⊥>      -->  E<3;⊥>  T<⊥,⊥;⊥>
    E<1;⊥>      -->  E<2;⊥>  T<⊥,⊥;⊥>
    Z           -->  E<1;⊥>
    Z'          -->  Z  $

5.  E<3;5>      -->  const<5>
    E<2;⊥>      -->  E<3;⊥>  T<⊥,⊥;⊥>
    E<1;⊥>      -->  E<2;⊥>  T<⊥,⊥;⊥>
    Z           -->  E<1;⊥>
    Z'          -->  Z  $
```

lookahead = op<2>

```
6.  E<2;⊥>      -->  E<3;5>  T<3,5;⊥>
    E<1;⊥>      -->  E<2;⊥>  T<⊥,⊥;⊥>
    Z           -->  E<1;⊥>
    Z'          -->  Z  $

7.  T<3,5;5>    -->  ε
    E<2;⊥>      -->  E<3;5>  T<3,5;⊥>
    E<1;⊥>      -->  E<2;⊥>  T<⊥,⊥;⊥>
    Z           -->  E<1;⊥>
    Z'          -->  Z  $

8.  E<2;5>      -->  E<3;5>  T<3,5;5>
    E<1;⊥>      -->  E<2;⊥>  T<⊥,⊥;⊥>
    Z           -->  E<1;⊥>
    Z'          -->  Z  $

9.  E<1;⊥>      -->  E<2;5>  T<2,5;⊥>
    Z           -->  E<1;⊥>
    Z'          -->  Z  $

10. T<2,5;⊥>    -->  op<⊥>  E<⊥;⊥>  [xqt]<⊥,⊥,⊥;⊥>  T<⊥,⊥;⊥>
    E<1;⊥>      -->  E<2;5>  T<2,5;⊥>
    Z           -->  E<1;⊥>
    Z'          -->  Z  $
```



```
11. T<2,5;⊥>    -->  op<2>  E<2;⊥>  [xqt]<⊥,⊥,⊥;⊥>  T<⊥,⊥;⊥>
    E<1;⊥>      -->  E<2;5>  T<2,5;⊥>
    Z           -->  E<1;⊥>
    Z'          -->  Z  $
```

lookahead = const<2>

```
12. E<2;⊥>      -->  E<3;⊥>  T<⊥,⊥;⊥>
    T<2,5;⊥>    -->  op<2>  E<2;⊥>  [xqt]<⊥,⊥,⊥;⊥>  T<⊥,⊥;⊥>
    E<1;⊥>      -->  E<2;5>  T<2,5;⊥>
    Z           -->  E<1;⊥>
    Z'          -->  Z  $

13. E<3;2>      -->  const<2>
    E<2;⊥>      -->  E<3;⊥>  T<⊥,⊥;⊥>
    T<2,5;⊥>    -->  op<2>  E<2;⊥>  [xqt]<⊥,⊥,⊥;⊥>  T<⊥,⊥;⊥>
    E<1;⊥>      -->  E<2;5>  T<2,5;⊥>
    Z           -->  E<1;⊥>
    Z'          -->  Z  $
```

lookahead = op<3>

```
14. E<2;⊥>      -->  E<3;2>  T<3,2;⊥>
    T<2,5;⊥>    -->  op<2>  E<2;⊥>  [xqt]<⊥,⊥,⊥;⊥>  T<⊥,⊥;⊥>
    E<1;⊥>      -->  E<2;5>  T<2,5;⊥>
    Z           -->  E<1;⊥>
    Z'          -->  Z  $

15. T<3,2;⊥>    -->  op<3>  E<3;⊥>  [xqt]<⊥,⊥,⊥;⊥>  T<⊥,⊥;⊥>
    E<2;⊥>      -->  E<3;2>  T<3,2;⊥>
    T<2,5;⊥>    -->  op<2>  E<2;⊥>  [xqt]<⊥,⊥,⊥;⊥>  T<⊥,⊥;⊥>
    E<1;⊥>      -->  E<2;5>  T<2,5;⊥>
    Z           -->  E<1;⊥>
    Z'          -->  Z  $
```

lookahead = const<4>

```
16. E<3;4>      -->  const<4>
    T<3,2;⊥>    -->  op<3>  E<3;⊥>  [xqt]<⊥,⊥,⊥;⊥>  T<⊥,⊥;⊥>
    E<2;⊥>      -->  E<3;2>  T<3,2;⊥>
    T<2,5;⊥>    -->  op<2>  E<2;⊥>  [xqt]<⊥,⊥,⊥;⊥>  T<⊥,⊥;⊥>
    E<1;⊥>      -->  E<2;5>  T<2,5;⊥>
    Z           -->  E<1;⊥>
    Z'          -->  Z  $
```

lookahead = $

```
17. T<3,2;⊥>    -->  op<3>  E<3;4>  [xqt]<2,4,3;8>  T<3,8;⊥>
    E<2;⊥>      -->  E<3;2>  T<3,2;⊥>
    T<2,5;⊥>    -->  op<2>  E<2;⊥>  [xqt]<⊥,⊥,⊥;⊥>  T<⊥,⊥;⊥>
    E<1;⊥>      -->  E<2;5>  T<2,5;⊥>
    Z           -->  E<1;⊥>
    Z'          -->  Z  $

18. T<3,8;8>    -->  ε
    T<3,2;⊥>    -->  op<3>  E<3;4>  [xqt]<2,4,3;8>  T<3,8;⊥>
    E<2;⊥>      -->  E<3;2>  T<3,2;⊥>
    T<2,5;⊥>    -->  op<2>  E<2;⊥>  [xqt]<⊥,⊥,⊥;⊥>  T<⊥,⊥;⊥>
    E<1;⊥>      -->  E<2;5>  T<2,5;⊥>
    Z           -->  E<1;⊥>
    Z'          -->  Z  $

19. T<3,2;8>    -->  op<3>  E<3;4>  [xqt]<2,4,3;8>  T<3,8;8>
    E<2;⊥>      -->  E<3;2>  T<3,2;⊥>
    T<2,5;⊥>    -->  op<2>  E<2;⊥>  [xqt]<⊥,⊥,⊥;⊥>  T<⊥,⊥;⊥>
    E<1;⊥>      -->  E<2;5>  T<2,5;⊥>
    Z           -->  E<1;⊥>
    Z'          -->  Z  $

20. E<2;8>      -->  E<3;2>  T<3,2;8>
    T<2,5;⊥>    -->  op<2>  E<2;⊥>  [xqt]<⊥,⊥,⊥;⊥>  T<⊥,⊥;⊥>
    E<1;⊥>      -->  E<2;5>  T<2,5;⊥>
    Z           -->  E<1;⊥>
    Z'          -->  Z  $

21. T<2,5;⊥>    -->  op<2>  E<2;8>  [xqt]<5,8,2;13>  T<2,13;⊥>
    E<1;⊥>      -->  E<2;5>  T<2,5;⊥>
    Z           -->  E<1;⊥>
    Z'          -->  Z  $

22. T<2,13;13>  -->  ε
    T<2,5;⊥>    -->  op<2>  E<2;8>  [xqt]<5,8,2;13>  T<2,13;⊥>
    E<1;⊥>      -->  E<2;5>  T<2,5;⊥>
    Z           -->  E<1;⊥>
    Z'          -->  Z  $

23. T<2,5;13>   -->  op<2>  E<2;8>  [xqt]<5,8,2;13>  T<2,13;13>
    E<1;⊥>      -->  E<2;5>  T<2,5;⊥>
    Z           -->  E<1;⊥>
    Z'          -->  Z  $

24. E<1;13>     -->  E<2;5>  T<2,5;13>
    Z           -->  E<1;⊥>
    Z'          -->  Z  $
```

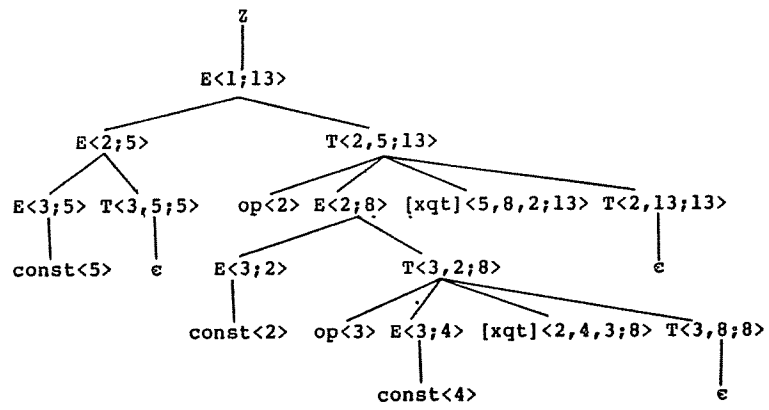


```
25. Z           -->  E<1;13>
    Z'          -->  Z  $

26. Z'          -->  Z  $
```

accept

The corresponding attributed derivation tree is:



3.6 Properties of the ALL(k) Parser

We will first demonstrate the correctness of the ALL(k) parsing algorithm, and then go on to examine the conditions under which its execution time is a linear function of the length of the input string.

Informally, the "correctness" of the parser is the property that it accepts exactly those strings which are generated by the given grammar. Unfortunately, we cannot guarantee that the ALL(k) parser will halt announcing "error" for illegal strings. The "Turing machine grammar" of Theorem 2.3.11 can be transformed into an ALL(1) grammar by changing the contextual predicates into d^1-predicates. An ALL(1) parser for this grammar which always halted would solve the halting problem for Turing machines. The difficulty is that with infinite attribute value sets, derivations are of unbounded length. We will return to this problem in our discussion of the linearity properties of the parsing algorithm.

Theorem 3.6.1 Given an ALL(k) grammar, (G,D^k), there exists a disambiguated leftmost derivation:

$$Z'' \underset{m_1}{\Longrightarrow} z''_1 \underset{m_2}{\Longrightarrow} \cdots \underset{m_t}{\Longrightarrow} z''_t \ , \ res(z''_t) \in L((G,D^k)),$$

if and only if the strong ALL(k) parser for (G,D^k) halts accepting on input $z'' = res(z''_t)$ after making the predictions $m_1,m_2,\ldots,m_t$.

Proof: The ALL(k) parser is a modification of the standard strong LL(k) parser, whose correctness is proved in chapter 5 of [AU72]. We need to show that the modifications do not affect correctness. We will also use a result, available from [Boc76] and [LRS74], that the L-attributed property guarantees that all attributes can be evaluated in a single left-to-right pass. In particular, whenever a nonterminal appears "on top" of the parsing stack, all of its inherited attributes are known to be evaluable. For the remainder of this proof, all derivations are considered to be leftmost derivations.

<u>if</u>: For each pair, (A'',u''), if $M(A'',u'')$ = <u>predict</u> m_i then <u>predict</u> $m_i \in M'(A,u)$, (where M' is the modified strong LL(k) parsing table). Thus, if predictions $m_1,\ldots,m_t$ are made on input z, the correctness of the strong LL(k) parsing algorithm implies:

$$Z \underset{m_1}{\Longrightarrow} \cdots \underset{m_2}{\Longrightarrow} z_t, \ z_t \in (T \cup K)^*$$

where $z = res(z_t)$. It remains to be shown that when attributes are included, the above derivation is an ALL(k) derivation. Therefore, two conditions must be satisfied:

(1) For each step $w''A''y'' \underset{m_i}{\Longrightarrow} w''x''y''$ of the attributed derivation, where $x''y'' \Longrightarrow^* z''_a$ and $p_{m_i} = (A \dashrightarrow x)$, $d_{m_i}(A'',v'') = \underline{\text{true}}$, where v'' are the first j symbols of $res(z''_a)$.

This follows immediately, since $M(A'',v'') = \underline{\text{predict}}\ m_i$ implies that $d_{m_i}(A'',v'') = \underline{\text{true}}$ by the definition of the parsing function.

(2) Each contextual predicate in each C_{m_i} is satisfied.

Step 4 of the ALL(k) algorithm evaluates contextual predicates associated with production m_i, where (m_i,j) is on top of the parsing stack. A contextual predicate can be evaluated whenever all of its arguments have been evaluated. Each prediction places an $(m_i,0)$ entry on the stack, and this entry is only popped when its second component has been incremented to $n(m_i)$. But by this time all of the attribute instances of production m_i have been evaluated. Thus all of the associated contextual predicates have also been evaluated.

<u>only if</u>: Again, the correctness of the strong LL(k) parser implies that if:

$$Z \underset{m_1}{\Longrightarrow} \cdots \underset{m_t}{\Longrightarrow} z_t$$

then for each step, $wAy \underset{m_i}{\Longrightarrow} wxy$, where $xy \Longrightarrow^* z_a$, $m_i \in M'(A,u)$, where u are the first k symbols of $res(z_a)$. (Note that M' is the modified strong LL(k) parser table of algorithm 3.4.2). Given that the derivation satisfies the ALL(k) property, we have that $d_{m_i}(A'',v'') = \underline{\text{true}}$, where v'' are the first j symbols of $res(z''_a)$. Then by the definition of the parsing function M, $M(A'',u'') = \underline{\text{predict}}\ m_i$. Also, since we are given an attributed derivation, all of the contextual predicates in C_{m_i}, which the parser must evaluate, are know to be <u>true</u>. <u>Q.E.D.</u>

It is well-known that the number of steps executed by the LL(k) (or strong LL(k)) parsing algorithm on an input of length n is O(n). This is a direct result of the fact that LL(k) grammars cannot be left-recursive: the maximum number of steps before a <u>pop</u> move must be executed (consuming an input symbol) is bounded by the length of the longest leftmost derivation of the form $A \Longrightarrow^+ Bx$, which, in turn, is bounded by the number of nonterminals in the grammar, a constant.

A similar result can be obtained for the ALL(k) parser if we ignore the time required to evaluate the contextual and d^k-predicates, and the attribute evaluation and action functions. If each of these evaluations can be performed in constant time, then the total time taken by the parser will be O(n) (with the added conditions we will provide below). In the absence of this restriction, we will simply show that the number of moves (<u>predicts</u>'s and <u>pop</u>'s) performed by the parser is O(n).

Of course, if the ALL(k) grammar has an underlying LL(k), and hence non-left-recursive, grammar, linearity follows immediately. However, the ALL(k) definition does not require that the underlying grammar be LL(k), and even left-recursion is permitted (see example 3.5.2). But there is an attributive form of left-recursion which may be proscribed:

> Definition 3.6.2 An ALL(k) grammar is <u>attributively</u> <u>left-recursive</u> if and only if there is a partial disambiguated leftmost derivation:
>
> $$Z'' \Rightarrow^* w''A''y'' \Rightarrow^+ w''A''x''y'' \Rightarrow^* \ldots$$
>
> where $A'' \in N^I$.

Banning attributive left-recursion is, however, neither sufficient nor effective. It is not sufficient

since, with attributes over infinite domains, an infinite derivation

$$A''_1x_1 \Rightarrow A''_2x_2 \Rightarrow \ldots \Rightarrow A''_ix_i \Rightarrow \ldots$$

is possible where all of the A''_i are distinct. It is not effective since, with unrestricted predicates, attributive left-recursion is an undecidable property. A solution to both of these problems is to require that all attributes examined by the d^k-predicates possess finite attribute value sets, (note that this is the same restriction discussed in section 3.4 as being sufficient for testing the ALL(k) condition).

Now, attributive left-recursion may be tested for each attributed symbol, $A'' \in N^I$. Let m bound the number of inherited attributes for each nonterminal, and let v bound the size of the attribute value sets. Then examine all partial leftmost attributed derivations, of length at most $|N| \cdot v^m$, rooted by A", in which the leftmost symbol of each sentential form is always a nonterminal. There can only be a constant number of distinct derivations with these restrictions. Then, by the pigeonhole principle, if A" is attributively left-recursive, it will so appear in one of these derivations.

If an ALL(k) grammar is not attributively left-recursive, and all attributes examined by the d^k-predicates have finite attribute value sets, linearity follows for reasons similar to the linearity of the LL(k)

parser. That is, the number of <u>predict</u> moves before a <u>pop</u> must be executed is bounded by the length of the longest partial derivation A" ==>+ B"x", A" ≠ B", which is bounded by the number of distinct attributed symbols, a constant. Thus we get:

Theorem 3.6.2 Given a strong ALL(k) grammar which is not attributively left-recursive, and which has finite attribute value sets for all attributes examined by the d^k-predicates, the number of moves performed by the corresponding strong ALL(k) parser on an input of length n is O(n).

CHAPTER 4

APPLICATIONS

4.1 Grammar Compression

It has been noted by several authors (see below), that ambiguous grammars, together with some disambiguating constructs, can often present a shorter and more understandable definition of a language than can be achieved with an equivalent unambiguous grammar. We shall examine two common programming language constructs to which grammar "compression", achieved with attributed grammars, can be used to great advantage. First, expression grammars will be considered, and then the problem of specifying multiple unordered options will be treated.

Expression Grammars

Unambiguous context-free expression grammars usually require a different production for each operator, as well as a separate nonterminal for each level of operator precedence. Using LL(1) grammars, the following structure is typical:

```
Expr    -->  Term  E-list
E-list  -->  +  Term  E-list
E-list  -->  -  Term  E-list
E-list  -->  ε
Term    -->  Factor  T-list
T-list  -->  *  Factor  T-list
T-list  -->  /  Factor  T-list
T-list  -->  ε
Factor  -->  Primary  F-list
          .
          .
          .
         etc.
```

Aho, Johnson, and Ullman, ([AJU75]), and Earley, ([Ear75]), have devised schemes (for bottom-up grammars) where an ambiguous grammar, supplied with a table of operator priorities, is sufficient to generate the desired parsing algorithm. We shall find that attributed grammars (specifically, ALL(1) grammars) can not only effect a greater reduction in grammar size than these previous techniques, but can also directly express the context-sensitive restrictions on the compatibility of operand and operator types.

The following grammar expresses the syntax of ALGOL 60 expressions ([Nau62]) involving simple variables and constants, with the exception of designational and conditional expressions. More properly, this is a grammar

fragment, since we make use of an action, [gettype], which presumes the existence of a symbol table constructed from declarations that would be present in a full grammar.

We require a scanner which will return the terminal class "op" with the appropriate precedence attribute "p" for the following operators:

precedence	operator
2	≡
3	⊃
4	∨
5	∧
6	¬
7	<, ≤, =, ≥, >, ≠
8	+, −
10	*, /, ÷
11	↑

For a "variable", the scanner will return the appropriate string in the attribute "name"; for unsigned numbers the scanner will return the terminal class "unsigned-number"; for the boolean values <u>true</u> and <u>false</u> the scanner will return "logical-value".

The following description of the grammar should be read in conjunction with the grammar itself, which is presented below.

As in Example 3.5.2, the "pe" and "p" attributes are used to enforce the precedence of the operators. Here, however, the unary operators require special treatment

(production 2), further complicated by the fact that "+" and "-" are also binary operators. ALGOL 60 does not permit a "+" or "-" to follow a "+" or "-", and this restriction is handled by skipping the precedence "9" in the operator table, and including some special cases in the predicates for productions 8 and 9.

The "ti" and "ts" attributes are used along with the contextual predicates to enforce the context-sensitive restrictions with respect to type ("boolean" or "arithmetic"). There is an effort to detect type violations as early as possible in the parse. Essentially, the inherited "ti" attribute is used to constrain the type of the next operand, ("variable", "unsigned-number", or "logical-value"), and a value of "any" indicates that there is no constraint. These constraints are checked by the contextual predicates for productions 5, 7, and 8. Since a boolean operand may begin with an arithmetic operand (as part of a relational expression), a "ti" value of "any" will be inherited immediately to the right of any boolean operator. The type of a right operand can only be determined when the operand has been completely parsed. The resulting type is embodied in the synthesized "ts" attribute, which is returned up the derivation tree for further checking by the contextual predicates for productions 2 and 8.

Example 4.1.1

Nonterminals

```
Z
E     pe,ti: inherited;  ts: synthesized
EL    p,ti: inherited;  ts: synthesized
```

Terminals

```
op    p: synthesized
variable     name: synthesized
unsigned-number
logical-value
)
(
```

Actions

```
[gettype]     name: inherited;  ts: synthesized
          (sets "ts" to the declared type of "name")
```

Attribute Value Sets

```
pe: {1,2,3,4,5,6,7,8,9,10,11}
p: {2,3,4,5,6,7,8,9,10,11}
ti,ts: {"any","boolean","arithmetic"}
name:  {alphanumeric strings beginning with a letter}
```

Productions

```
1)  Z  -->  E

     E.pe <-- 1
     E.ti <-- "any"
```

```
2)  E1  -->  op  E2  EL

    d(E1",op"):  (E1.pe = op.p)

    CP:  (if op.p = 6 then E2.ts = "boolean"
                      else true)

     E2.pe <-- op.p + 1
     E2.ti <-- if op.p = 8 then "arithmetic" else "any"
     EL.p <-- op.p + 1
     EL.ti <-- E2.ts
     E1.ts <-- EL.ts

3)  E1 --> (  E2  )

    d(E1",("):  (E1.pe = 11)

     E2.p <-- 1
     E2.ti <-- E1.ti
     E1.ts <-- E2.ts

4)  E1  -->  E2  EL

    d(E1",op"):  (E1.pe ≠ op.p)
    d(E1",variable"):  (E1.pe < 11)
    d(E1",unsigned-number"):  (E1.pe < 11)
    d(E1",logical-value"):  (E1.pe < 7)
    d(E1",("):  (E1.pe < 11)

     E2.pe <-- E1.p + 1
     E2.ti <-- E1.ti
     EL.p <-- E1.p + 1
     EL.ti <-- E2.ts
     E1.ts <-- EL.ts

5)  E  -->  variable  [gettype]

    d(E",variable"):  (E.pe = 11)

    CP:  ((E.ti = "any") or (E.ti = [gettype].ts))

     [gettype].name <-- variable.name
     E.ts <-- [gettype].ts
```



```
6)  E  -->  unsigned-number

    d(E",unsigned-number"):  (E.pe = 11)

     E.ts <-- "arithmetic"

7)  E  -->  logical-value

    d(E",logical-value"):  (E.pe = 7)

    CP:  (E.ti ≠ "arithmetic")

     E.ts <-- "boolean"

8)  EL1  -->  op  E  EL2

    d(EL1",op"):  (((EL1.p = op.p) and (op.p ≠ 6))
                      or ((EL1.p = 9) and (op.p = 8)))

    CP:  (((EL1.ti = E.ts = "boolean") and (op.p < 7))
         or ((EL1.ti = E.ts = "arithmetic") and (op.p ≥ 7)))

     E.pe <-- if 7 ≤ op.p ≤ 8 then EL1.p + 1 else EL1.p
     E.ti <-- if op.p < 7 then "any" else "arithmetic"
     EL2.p <-- E.pe
     EL2.ti <-- if op.p ≤ 7 then "boolean" else "arithmetic"
     EL1.ts <-- EL2.ts

9)  EL  -->  e

    d(EL",op"):  ((EL.p ≠ op.p)
                  and ((EL.p ≠ 9) or (op.p ≠ 8)))

     EL.ts <-- EL.ti
```

Derivation trees generated with this grammar are isomorphic to the trees that would be generated using an equivalent unambiguous LL(1) grammar: the ALL(1) technique of grammar compression has not resulted in shorter derivations despite the reduction in the size of the grammar and the parser.

Multiple Unordered Options

Constructs involving multiple unordered options are prevalent in many job control languages and in PL/1 and COBOL. Context-free grammars generating such constructs require large numbers of productions. For example, with three options (call them a, b, and c), an LL(1) grammar requires nineteen productions:

```
S  -->  a BC  |  b AC  |  c AB  |  ε
AB  -->  a B  |  b A  |  ε
AC  -->  a C  |  c A  |  ε
BC  -->  b C  |  c B  |  ε
A  -->  a  |  ε
B  -->  b  |  ε
C  -->  c  |  ε
```

In general, for n options, the number of productions required is:

$$\sum_{i=1}^{n} \binom{n}{i}(i+1) = (n+2)2^{n-1} - 1 \quad .$$

It is easy to see why this sort of construct is rarely encountered in languages described by context-free grammars. PL/1, which is defined by other methods, would require an impractically large context-free grammar to handle the 55 options that may be appended to a

declaration (even though many of these options will be mutually exclusive).

An ALL(1) grammar, on the other hand, requires only n+2 productions to represent n unordered options. In addition, the inevitable restrictions on combinations of options can be managed quite easily. The following grammar illustrates the general method for n options, labelled $a_1,\ldots,a_n$.

Example 4.1.2

<u>Nonterminals</u>

Z
Option seti: inherited; sets: synthesized

<u>Terminals</u>

$a_1,a_2,\ldots,a_n$

<u>Attribute Value Sets</u>

seti,sets: 2^A (where $A = \{a_1,\ldots,a_n\}$)

Productions

1) $Z \text{ --> } Option^1 \; Option^2 \; \ldots \; Option^n \; \$$

$Option^1.seti \text{ <-- } \{a_1,\ldots,a_n\}$

for $2 \leq i \leq n$:

$Option^i.seti \text{ <-- } Option^{i-1}.sets$

for $1 \leq i \leq n$:

2_i) $Option \text{ --> } a_i$

$d(Option'',a_i)$: $(a_i \in Option.seti)$
$d(Option'',\$)$: false

$Option.sets \text{ <-- } Option.seti - \{a_i\}$

3) $Option \text{ --> } \epsilon$

$d(Option'',a_i)$: false , for $1 \leq i \leq n$
$d(Option'',\$)$: true

$Option.sets \text{ <-- } Option.seti$

At each step in the derivation, $Option^i.seti$ constitutes the set of option names which $Option^i$ is permitted to derive. At the end of the derivation, the complement of $Option^n.sets$ is the set of option names which have been derived. Restrictions against particular option combinations can be enforced in two ways. The simplest method is to add a contextual predicate to production (1) -- the option names present in $Option^n.sets$ are precisely those which are absent from the recognized (or generated) text. A slightly more difficult method, which will provide earlier detection of a violated restriction, is to

alter the attribute evaluation rules of productions 2_i. The present rules pass along to Option.sets all option names which have not yet occurred. Since the option names which have occurred can be deduced from Option.seti and a_i, any desired restriction can be implemented by assigning a smaller set to Option.sets. A restriction requiring the presence of a particular option combination can, of course, only be enforced by a check that appears at the end of the option string. This is implemented by a contextual predicate that examines $Option^n.sets$ for the absence of the required combination.

4.2 Implementation of Full LL(k) Parsers

Strong ALL(k) grammars can be used to provide an efficient implementation of full LL(k) parsers. An important application of this technique is in the field of error-correction, even when LL(1) grammars are used. While every full LL(1) grammar is also strong LL(1), the strong LL(1) parser will not necessarily detect an error upon first encountering the erroneous symbol. For this

reason, a full LL(1) parser (which detects errors immediately) has been found necessary for LL(1) error-correction ([FMQ77]).

Given a context-free grammar $G = (N,T,P,S)$, a full LL(k) parser can be constructed by first building an equivalent strong LL(k) grammar $G' = (N',T,P',S')$, and then using the strong LL(k) parser table constructor (see section 3.4). The algorithm for building G' involves the creation of "new" nonterminals of the form $[X,L]$, where $X \in N$ and $L \subseteq T^{*k}$. L constitutes the set of valid local lookaheads for the nonterminal X; that is, for <u>every</u> partial leftmost derivation in G':

$$[S',-] \Rightarrow^* w[X,L]y$$

we have $FIRST_k(y) \in L$. Also, for each $z \in L$, there exists a leftmost derivation:

$$[S',-] \Rightarrow^* w[X,L]z\ldots \qquad , \text{ for some } w.$$

For a given $X \in N$ there may be many nonterminals of the form $[X,L] \in N'$, but derivations in G' will be isomorphic to those in G in the following sense. Define a homomorphism h on $T \cup N'$ such that:

$$h(a) = a \qquad \text{for all } a \in T$$

$$h([X,L]) = X \qquad \text{for all } [X,L] \in N' .$$

The following two properties can now be expressed for all leftmost derivations, (a proof that they indeed hold can be found in [AU72], chapter 5):

1) given a leftmost derivation in G':

$$w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n \quad , w_i \in (T \cup N')^*$$

the following derivation is in G:

$$h(w_0) \Rightarrow h(w_1) \Rightarrow \ldots \Rightarrow h(w_n)$$

2) given a leftmost derivation in G:

$$v_0 \Rightarrow v_1 \Rightarrow \ldots \Rightarrow v_n \quad , v_i \in (T \cup N)^*$$

there exists exactly one leftmost derivation in G':

$$w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n$$

such that for each i, $h(w_i) = v_i$.

The following algorithm can be used to construct G':

Algorithm 4.2.1 Given an LL(k) grammar $G = (N,T,P,S)$, construct an equivalent strong LL(k) grammar $G' = (N',T,P',S')$.

```
(initialize)
S' := [S,{ε}]; N' := {[S,{ε}]}; P' := ∅;
(main loop)
repeat
 for each [X,L] ∈ P'
  do
   for each production in P with
            lefthand-side X
    do
     (let p = (X --> w_0 X_1 w_1 ... X_n w_n)
      where w_i ∈ T*, X_i ∈ N,
      and let L_i = FIRST_k(w_i X_{i+1} ... X_n w_n · L))
     P' := P' ∪ ([X,L] --> w_0 [X_1,L_1] w_1 ... [X_n,L_n] w_n);
     N' := N' ∪ {[X_1,L_1],...,[X_n,L_n]}
until no new nonterminals are added to N';
```

G' may be a substantially larger grammar than G. We have used this algorithm on an LL(1) grammar for PASCAL, resulting in an equivalent strong LL(1) grammar with five times as many nonterminals and productions. A strong ALL(k) grammar can provide an implementation of G' by representing the L-sets as attributes of the respective

nonterminals, and computing these attributes during the parse. The size of the original grammar is preserved, at the expense of some extra computation at parse-time. The linear parse time is still preserved, however, since this extra computation is only required for prediction steps, and merely increases the time to process a prediction by a constant.

Given an LL(k) grammar $G = (N,T,P,S)$, the following strong ALL(k) grammar is equivalent:

Example 4.2.2

Nonterminals

X L: inherited for all $X \in N$

Terminals

a for all $a \in T$

Attribute Value Sets

L: $2^{(T^{*k})}$

Productions

1) Z --> S

Z.L <-- {ε}

for each production $p_i = (X \text{ --> } w_0 X_1 \ldots X_n w_n) \in P$
where $w_i \in T^*$, $X_i \in N$:

2_i) X --> w_0 X_1 w_1 ... X_n w_n

for all $u \in T^{*k}$:

$d(X'',u'')$: $(u \in FIRST_k(w_0 X_1 \ldots X_n w_n \cdot X.L))$

for $1 \leq i \leq n$:

$X_i.L$ <-- $FIRST_k(w_i X_{i+1} \ldots X_n w_n \cdot X.L)$

The implementation of an ALL(1) grammar of this form is particularly efficient. The L attributes can be represented by bit vectors, with each bit representing a terminal symbol. The attribute evaluation functions in productions 2_i will be constant if $w_i X_{i+1} \ldots X_n w_n \not\Rightarrow^* \varepsilon$. In any case, $FIRST_1(w_i X_{i+1} \ldots X_n w_n)$ can be precomputed, and

the concatenation with X.L can be implemented as a bitwise-or operation. Some space can be sacrificed for time by representing the d-predicates by the usual full LL(1) parsing table, and, at parse time, using the evaluated X-symbol to access the appropriate row.

4.3 Disambiguating Without Attributes

There are certain programming language constructs for which LL(1) grammars do not exist. The classic example is the <u>if</u>-<u>then</u>-<u>else</u> statement where the <u>else</u> is to be associated with the closest unmatched <u>then</u>. The following ambiguous grammar illustrates the problem:

1) Stmt --> <u>if</u> Bool-expr <u>then</u> Stmt Elsepart
2) Stmt --> Other-stmts
3) Elsepart --> <u>else</u> Stmt
4) Elsepart --> ε

The specific LL(1) conflict arises from the Elsepart productions:

$FIRST_1(\underline{else}) \cap FIRST_1(\epsilon \cdot FOLLOW_1(Elsepart)) = \{\underline{else}\}$

To enforce the rule that each else is to be associated with the closest unmatched then, it is simply necessary to predict the production "Elsepart --> else" whenever else appears as the lookahead symbol. This can be accomplished with a disambiguating rule that does not need any attributes. Consider the following ALL(1) grammar:

1) Stmt --> if Bool-expr then Stmt Elsepart
2) Stmt --> Other-stmts
3) Elsepart --> else Stmt
   d(Elsepart,else): true
4) Elsepart --> ϵ
   d(Elsepart,else): false

In general, it is undecidable whether the application of this technique will leave the language unchanged. However, there are classes of grammars, investigated by Aho, Johnson, and Ullman ([AJU75]), where it can be proven that this form of disambiguating does not alter the defined language. The following definitions are from their paper.

Definition 4.3.1 A grammar G is in Class 1 if the following conditions hold:

1) G is not left recursive.
2) Let wAx be a left sentential form, and let A --> z_1 and A --> z_2 be productions of G such that a string beginning with the terminal "a" can be derived from both z_1x and z_2x. Then either:

   z_1x ==>* ay implies z_2x ==>* ay

   or, z_2x ==>* ay implies z_1x ==>* ay

   for all terminal strings y.

Condition 2 suggests that there exists a choice of A-productions which, if consistently made on lookahead "a", will not alter the defined language. Unfortunately, it can be shown that membership in Class 1 is undecidable. Class 1, however, properly contains a class of grammars which still includes all LL(1) grammars, as well as the if-then-else grammar presented above, and for which membership can be decided.

Definition 4.3.2 Given a grammar G and a nonterminal A, define:

$F(A) = \{X \mid X \in N \cup T$, and there is a left sentential form $wAyXz$ where $y \Rightarrow^* \epsilon \}$

Definition 4.3.3 A grammar G is in <u>Class 2</u> if whenever $A \rightarrow z_1$ and $A \rightarrow z_2$ are distinct productions, the following conditions hold:

1) $FIRST_1(z_1) \cap FIRST_1(z_2) = \emptyset$.
2) At most one of z_1 and z_2 is or derives ϵ.
3) Suppose $z_1 \Rightarrow^* \epsilon$ and $z_2 \Rightarrow^* ax$ for some terminal $a \in FOLLOW_1(A)$. Then for all $X \in F(A)$ such that $X \neq A$, $X \not\Rightarrow^* ay$ for any string y.

When condition 3 applies, the production $A \rightarrow z_2$ is consistently predicted on lookahead "a". It is easy to show that the <u>if</u>-<u>then</u>-<u>else</u> grammar is in Class 2. The Stmt productions trivially satisfy all of the conditions. For the Elsepart productions we need to show that since <u>else</u> $\in FOLLOW_1$(Elsepart), there are no members of F(Elsepart), other than Elsepart, which derive a string beginning with <u>else</u>. However, Elsepart is the only member of F(Elsepart), so condition 3 is satisfied.

We can now provide the ALL(1) version of the relevant result from [AJU75].

Theorem 4.3.4 Every Class 1 (Class 2) grammar G has an equivalent Class 1 (Class 2) grammar G' in which every production is either of the form $A \rightarrow ax$ $(a \in T,\ x \in (N \cup T)^*)$, or $A \rightarrow \epsilon$. An ALL(1) grammar G" can be formed such that $L(G") = L(G')$ by providing for each production $A \rightarrow ax$ the disambiguating rule:

$d(A,a)$: <u>true</u>

and for each production $A \rightarrow \epsilon$ the disambiguating rules:

$d(A,a)$: <u>false</u>

for all $a \in FIRST_1(A)$.

Proof: The result follows immediately from condition 2 of the definition of Class 1.

CHAPTER 5
CONCLUSIONS

5.1 Attributed Parsing and Compiler Construction

In a recent paper, Marcotty, Ledgard, and Bochmann provide a critical discussion of several formal definition systems ([MLB76]). It is curious that they specifically omit as one of their criteria the utility of a definition in the "automatic implementation of compilers." However, it has been amply demonstrated that attributed grammars constitute a convenient method for organizing translations ([Fan72],[LRS74],[Bra76]). This is partially due to the fact that attributed grammars are not a completely formal specification tool. The formal methods of, for example, van Wijngaarden grammars ([Wij75]) or production systems ([Led72]), entail a very low-level specification that is characteristic of formal systems. Attributed grammars, on the other hand, do not constrain the structure of the attribute evaluation rules or contextual predicates. Only the functionality of the rules and predicates, and the paths of information flow within a derivation tree, are explicitly specified. Thus attributed grammars provide a mechanism for modularizing a translation, without

restricting the implementation of particular elements of the translation.

This thesis has sought to complement the traditional uses of attributed grammars by examining their utility in syntactic specification and parser construction. For these applications, the semi-formal nature of attributed grammars has also proven valuable. As with translation, attributed grammars provide a mechanism for modularizing a parsing algorithm, a framework on which context-sensitive restrictions can be placed. Context-sensitive restrictions can be modeled by formal systems, but a great deal of cumbersome machinery is required: the relative advantages of attributed grammars result from the lack of constraints placed on the implementation of such context-sensitive restrictions. This semi-formality has facilitated our development of the ALL(k) parsing algorithm. Of course, semi-formality has its drawbacks too, and these were examined in Chapter 3 in connection with testing for the ALL(k) property and determining the efficiency of the ALL(k) parser. Sufficient conditions were developed which made such analyses possible, and it is likely that these will not prove to be overly onerous when the technique is applied to complete languages.

The goal of much current compiler research is to automate the task of compiler construction. This thesis represents a step toward that goal: we have provided a

practical technique for constructing efficient parsers that can process many of the context-sensitive aspects of programming language syntax.

5.2 Directions for Future Research

This thesis has developed the concept of attributed parsing in an extension to the LL(k) parsing algorithm. The reason for the choice of LL(k) was discussed in Chapter 3: a top-down parser that does not back-up can provide, for L-attributed grammars, simultaneous evaluation of all attributes. L-attributed grammars permit both inherited and synthesized attributes, while only synthesized attributes are allowed in the class of grammars for which "on-the-fly" evaluation of attributes is possible during a bottom-up parse. This class, called S-attributed, has been described in [LRS74].

Nevertheless, attributed parsing will increase the power of bottom-up parsers, and we have begun work on the definition of attributed LR(k) grammars and parsers. Attributed parsing should be extendable to the precedence parsing techniques as well. The development of

generalized left-corner (GLC) parsers by Demers ([Dem77]), provides a particularly interesting candidate for attributed parsing. While top-down parsers permit more flexible attribute evaluation than bottom-up parsers, it is known that bottom-up parsing can handle a much larger class of context-free grammars. GLC parsing is a hybrid technique, parsing bottom-up until a production can be predicted, and then parsing top-down. A special recognition symbol is inserted in the righthand-side of each production: the portion of the righthand-side to the left of this symbol is recognized bottom-up, the portion to the right is recognized top-down. GLC-attributed grammars can be defined by applying the S-attributed restriction to the left of each recognition symbol, and the L-attributed restriction to the right. Attribute evaluation for GLC-attributed grammars has been studied by Rowland ([Row77]).

Attributed parsing opens up new possibilities in the field of error detection and correction. Error detection is greatly enhanced: not only are the traditional context-free detection methods still available, but context-sensitive errors can now be discovered by the parser through the use of contextual predicates. When an error has been detected, there is a wealth of information available in the values of the attributes. For example,

in the case of a type compatibility error, the attributes will reflect the types of the offending constructs.

In other work, we have developed methods for the automatic generation of efficient error-correctors for LL(1) parsers ([FMQ77]). The table-driven error-corrector computes insertions and deletions by considering the top symbols on the parsing stack along with the next input symbols. A natural extension is to incorporate into the table the attributes of these symbols. There are a number of difficulties to be overcome before this can be a practical technique. First, the domain of the attributes will usually be too large to maintain a table of manageable size. Second, the correction (insertions, deletions, or combinations thereof) must itself be compatible in a context-sensitive sense with the program being corrected. The resolution of these problems in a time and space efficient manner would constitute a significant contribution to the science of error-correction.

BIBLIOGRAPHY

[AJU75] Aho, A. V., S. C. Johnson, and J. D. Ullman. "Deterministic Parsing of Ambiguous Grammars," Communications of the ACM, Vol. 18, No. 8, Aug. 1975, pp. 441-452.

[AU72] Aho, A. V., and J. D. Ullman. The Theory of Parsing, Translation, and Compiling, Volume 1: Parsing, Prentice-Hall, Englewood Cliffs, N.J., 1972.

[Boc76] Bochmann, G. V. "Semantic Evaluation from Left to Right," Communications of the ACM, Vol. 19, No. 2, Feb. 1976, pp. 55-62.

[Bra76] Branquart, P., J.-P. Cardinael, J. Lewi, J.-P. Delescaille, and M. Vanbegin. An Optimized Translation Process and its Application to ALGOL 68, Springer-Verlag, Berlin, 1976.

[Cho59] Chomsky, N. "On Certain Formal Properties of Grammars," Information and Control, Vol. 2, 1959, pp. 137-167.

[Coo71] Cook, S. A. "The Complexity of Theorem Proving Procedures," Proc. 3rd Annual ACM Symposium on Theory of Computing, 1971, pp. 151-158.

[Dem77] Demers, A. "Generalized Left-Corner Parsing,", Conference Record of the 4th ACM Symposium on Principles of Programming Languages, 1977, pp. 170-182.

[Ear68] Earley, J. "An Efficient Context-free Parsing Algorithm," Ph.D. Thesis, Carnegie-Mellon University, 1968.

[Ear74] Earley, J. "Ambiguity and Precedence in Syntax Description," Acta Informatica, Vol. 4, 1975, pp. 183-192.

[Fan72] Fang, I. "FOLDS, A Declarative Formal Language Definition System," Ph.D. Thesis, Stanford Univ., 1972.

[FMQ77] Fischer, C. N., D. R. Milton, and S. B. Quiring. "An Efficient Insertion-Only Error-Corrector for LL(1) Parsers," Conference Record of the 4th ACM Symposium on Principles of Programming Languages, 1977, pp. 97-103.

[Fos68] Foster, J. M. "A Syntax Improving Program," Computer Journal, Vol. 11, No. 1, May 1968, pp. 31-34.

[HU69] Hopcroft, J. E., and J. D. Ullman. Formal Languages and their Relation to Automata, Addison-Wesley, Reading, Mass., 1969.

[Iro61] Irons, E. T. "A Syntax-Directed Compiler for ALGOL 60," Communications of the ACM, Vol. 4, No. 1, Jan. 1961, pp. 51-55.

[Iro63] Irons, E. T. "Towards More Versatile Mechanical Translators," Proceedings of Symposia in Applied Mathematics, Vol. 15, American Mathematical Society, Providence, R.I., 1963, pp. 41-50.

[KF63] Katz, J. L., and J. A. Fodor. "The Structure of a Semantic Theory," Language, Vol. 39, 1963, pp. 170-210.

[Knu68] Knuth, D. E. "Semantics of Context-free Languages," Mathematical Systems Theory, Vol. 2, No. 2, June 1968, pp. 127-146.

[Knu71] Knuth, D. E. "Top-down Syntax Analysis," Acta Informatica, Vol. 1, No. 2, 1971, pp. 79-110.

[Lan73] Langmaack, H. "On Correct Procedure Parameter Transmission in Higher Programming Languages," Acta Informatica, Vol. 2, 1973, pp. 110-142.

[Led69] Ledgard, H. F. "A Formal System for Defining the Syntax of Computer Languages," Ph.D. Thesis, MAC-TR-60, Project MAC, Mass. Inst. of Tech., 1969.

[LS68] Lewis, P. M., II, and R. E. Stearns, "Syntax-Directed Transduction," Journal of the ACM, Vol. 15, No. 3, July 1968, pp. 465-488.

[LRS74] Lewis, P. M., II, D. J. Rosenkrantz, and R. E. Stearns. "Attributed Translations," Journal of Computer and System Sciences, Vol. 9, No. 3, Dec. 1974, pp. 279-307.

[LRS76] Lewis, P. M., II, D. J. Rosenkrantz, and R. E. Stearns. Compiler Design Theory, Addison-Wesley, Reading, Mass., 1976.

[MLB76] Marcotty, M., H. F. Ledgard, and G. V. Bochmann. "A Sampler of Formal Definitions," ACM Computing Surveys, Vol. 8, No. 2, June 1976, pp. 191-276.

[Nau63] Naur, P., (ed.). "Revised Report on the Algorithmic Language ALGOL 60," Communications of the ACM, Vol. 6, No. 1, Jan. 1963, pp. 1-17.

[Nil71] Nilsson, Nils J. Problem-Solving Methods in Artificial Intelligence, McGraw-Hill, New York, 1971.

[RS70] Rosenkrantz, D. J., and R. E. Stearns. "Properties of Deterministic Top-down Grammars," Information and Control, Vol. 17, No. 3, Oct. 1970, pp. 226-256.

[Row77] Rowland, B. R. "Combining Parsing and Evaluation for Attributed Grammars," Ph.D. Thesis, University of Wisconsin-Madison, 1977.

[Sal73] Salomaa, A. Formal Languages, Academic Press, New York, 1973.

[Tur36] Turing, A. M. "On Computable Numbers with an Application to the Entscheidungsproblem," Proceedings of the London Mathematical Society, Series 2, Vol. 42, pp. 230-265, Vol. 43, pp. 544-546. (Reprinted in Davis, M. The Undecidable, Raven Press, New York, 1965.)

[Wij75] Wijngaarden A. van, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisher. "Revised Report on the Algorithmic Language ALGOL 68," Acta Informatica, Vol. 5, 1975.

[Wil71] Wilner, W. T. "Declarative Semantic Definition," Ph.D. Thesis, Stanford Univ., 1971.