COMBINING PARSING AND EVALUATION FOR ATTRIBUTED GRAMMARS

by

Bruce Ramon Rowland

Computer Sciences Technical Report #308
November 1977

COMBINING PARSING AND EVALUATION
FOR ATTRIBUTED GRAMMARS

BY

BRUCE RAMON ROWLAND

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

©Copyright 1977 by Bruce Ramon Rowland

COMBINING PARSING AND EVALUATION FOR ATTRIBUTED GRAMMARS

by Bruce Ramon Rowland

Under supervision of Assistant Professor Charles N. Fischer

ABSTRACT

Attributed grammars, developed by Knuth, permit the formal specification of context-sensitive syntax and semantics within the framework of context-free grammars. This thesis explores techniques to combine parsing with attributed grammar evaluation. The methods of Lewis, Rosenkrantz, and Stearns for evaluating the attributes of symbols during a parse of a string are expanded to include both S- and L-attributed grammars as special cases. An extension that involves the temporary retention of subtrees and delays their evaluation during a parse allows evaluation of all non-circular attributed grammars with LR(k) context-free grammars. An attributed pushdown processor using left-corner parsing techniques is described to perform efficient translations specified by attributed grammars. It serves as a model for single-pass compilation that formalizes and generalizes the use of a semantic stack to encompass forward references. The applicability of the processor in a practical translator writing system is considered, and the salient table construction and run-time properties are established.

Acknowledgements

I am greatly indebted to a number of people and resources that were of significant benefit to me at the University of wisconsin and aided in the completion of my thesis.

Of the people, my advisor Charles Fischer provided both the motivation for the ideas and the guidance toward their solutions. Fellow students, Donn Milton and Stephen Skedzeleski gave helpful feed-back during discussions of many of the problems I encountered. My readers, Marvin Solomon and Raphael Finkel were crucial influences on the final organization and comprehensibility of my dissertation.

The resources that supported my studies were a fellowship from the Wisconsin Alumni Research Foundation and assistantships from the Computer Sciences Department and the Academic Computing Center. Other useful resources existed on the Center's Univac 1110 and included Simula 67 for algorithm testing and Text for the dissemination of my ideas.

Above all, I'm indebted to Phyllis for her encouragement and the patience she had to put up with it all.

Table of Contents

Chapter 1
Introduction 1

Chapter 2
Attribute Availability 32

Chapter 3
The Attributed Pushdown Processor 58

Table of Contents

List of Algorithms

List of Figures

Chapter 1

Introduction

Automated Translation Techniques

Interest in translator writing systems has taken many forms ever since compilers were first introduced. Compiler-compilers [FG68], for example, have much to offer the compiler writer in savings of time and reduction of errors. Formal language specification techniques used in compiler-compilers add to the theory of formal languages. Automated compiler construction is often easier than conventional compiler writing because of the amount of tedious work that is transfered to a computer. Those parts that can be automated may be completed early in a compiler project, so more effort can be directed to other less formalized areas, for example optimization, run-time support, and user interfaces. The ease and speed of constructor use paired with language testing encourages design modification. Given compiler constructors, design modification can change the outward appearance of a language or its internal semantics, or it can aid in making the recognizer more efficient without altering the actual language itself.

The automatic generation of language recognizers also aids the process of compiler bootstrapping and compiler portability [MHW70]. The more a compiler is driven by tables, the simpler it will be to transport between computers and to implement on a new machine. A first draft compiler can then be used so that ensuing compilers are written in their own language and made self-compiling.

Much of the early emphasis on translator writing systems centered on syntax analysis. Use of Backus-Naur Form (BNF) notation became widespread as a result of this work and is now commonly used to express context-free aspects of programming language syntax. BNF became the meta-language for syntax analyzer constructors that produced tables or programs (parsers) to accept sentences of the expressed language [DeR69; Knu65; Flo63; Knu71; AJ74]. Programs were also developed to automatically construct lexical analyzers to recognize the tokens of a language [JPAR68; Bak73]. The token meta-language normally consists of regular expressions that can be transformed into finite automata to perform source program scanning. The early constructors worked well in producing recognizers for well-formed programs in the language but often terminated prematurely on an ill-formed program. More recent research has considered the problems of constructing recognizers with error recovery [GR73] and error correction [Iro63; AP72; FMQ77].

The problems associated with the construction of automated semantic analyzers have been more difficult. Formal notations have been developed in which to express various aspects of semantics. Marcotty, Ledgard, and Bochmann in their "Sampler of Formal Definitions" [MLB77] attempt to analyze the strengths and weaknesses of four different techniques for formal language definition: W-grammars, production systems, Vienna Definition Language, and attributed grammars. These methods are compared mainly with respect to completeness, simplicity, and clarity. Their overview supports the claim that formal definition methods lay an important theoretical and practical groundwork in computer sciences, and they encourage further study. The practicality or ease of the formation of language translator constructors based on these methods, however, was not considered. Of the four methods, not all are equally applicable to the area of compiler implementation. Attributed grammars are in fact well suited for implementation; current research considers the problems of practical implementation [Boc76; JW75; KW76].

The advantages of formal language specification are very attractive. The development of a uniform and well-understood theory of compilation would certainly be an asset to both compiler writers (and ultimately compiler users) and to the theory of computation. Such a theory, as it evolves, should provide a notation needed for the communication, teaching, study, and enhancement of compilation techniques. A formal theory can also provide the necessary foundation for work in establishing language and translator properties, showing language and translator equivalence, and providing meaningful comparisons.

A formal semantic specification technique is necessary if automatic translator generators are to become a reality. Languages then can be designed and defined in an unambiguous and communicable medium. Implementations of the language will follow from the specification rather than serve to define the language as has often been the practice. Those details to be left open to the implementation can then be clearly demarcated.

This research develops techniques that enable practical implementation of attributed grammar specifications for language translation. Emphasis is placed on a single pass model of the compilation process capable of dealing with forward references. An important quality of the model is that it formalizes semantic compilation techniques that are now commonly in use.

Background: Attributed Grammars

Of the many attempts at language definition, attributed grammars [Knu68a] are flexible and powerful enough to readily provide definitions of most programming languages, while at the same time they provide a natural, concise and formal means of specification. As a result, they can be adapted directly to various types of translation schemes.

Attributed grammars are an extension to context-free grammars. A context-free syntax is desirable because of the efficient, automatically generated parsers that are commonly available. However, context-free syntax often represents only a superset of legal strings because much unacceptable program syntax cannot be eliminated without context-sensitive specifications.

A context-free language is a set of terminal strings derivable from a context-free grammar. A context-free grammar (cfg) G = (N,T,P,S) is a four-tuple with the following restrictions and terminology [HU69]:

1) N is a finite set of non-terminal symbols.
2) T is a finite set of terminal symbols such that T and N are disjoint.
3) P is a finite set of productions (or generative rules). Each member of P is of the form:

   A ::= B1 B2 ... Bn

   with finite $n \geq 0$, where $A \in N$ and each $Bi \in (N \cup T)$. A is the left-hand side and each Bi is a member of the right-hand side of the production.
4) S is a distinguished member of N, the goal symbol.

The language generated by G = (N,T,P,S) is denoted L(G). Terminal strings are generated in G by applying productions to non-terminals in strings generated from S. Productions are appled by contextually replacing a member A of N by the right component of a production with A as its left component. Notationally

$$X\ A\ Y ==> X\ B1\ B2\ \ldots\ Bn\ Y$$

denotes the application of (A, B1 B2 ... Bn) to X A Y where X,Y ∈ (N u T)*. Thus X A Y <u>derives</u> X B1 B2 ... Bn Y. The reflexive and transitive closure of ==> is denoted by =*=>. Formally L(G) may be defined as

$$\{\ X \mid S=*=>X,\ X \in T^*\ \}.$$

Much theory is available from the study of properties of context-free languages and grammars and their recognizers [HU69; AU73]. Automatically generated syntax recognizers rely heavily on such theories [AU73; FG68; Fel66].

Attributed grammars were introduced by Knuth as an enhancement of context-free grammars to satisfy two goals. The first goal was to obtain syntactic specification where context-free grammars are either very unwieldy or totally incapable. The second goal was the specification of contextually dependent semantic relationships between elements of a cfg. Instead of language recognizers, such grammars can be used as transducers to generate output such as a translation to an intermediate target language. An attributed grammar can represent a translation as an attribute value of the root node of a syntax tree. Attributed grammars can also be used to define procedures to produce labelled trees for use in code optimization algorithms [SU72; AN76; PAN76].

Attributes are associated with the symbols in the grammar (terminal or non-terminal) and can take on values from different, possibly infinite sets. Each grammar symbol has a fixed number of attributes; each attribute is either synthetic or inherited. Synthetic attributes of a symbol derive their values from attributes of symbols that are immediate descendants of the symbol in some syntax tree. Information flows up a tree to synthetic attributes. Inherited attributes of a grammar symbol derive their values from direct ancestor and sibling nodes in a syntax tree, permitting information flow down a tree. Inherited information may indirectly have an effect on synthetic attributes and synthesized information may eventually be passed to other subtrees via inherited attributes.

Attributed grammars include sets of attribute evaluation rules associated with each production in a context-free grammar. Each rule contains a function and is used to define the computation of an attribute value of a symbol in the production in terms of other attribute values of symbols in the same production. The rule for an attribute of the left-hand side symbol of a production defines a synthetic attribute, and a rule for an attribute of a symbol on the right-hand side of a production defines an inherited attribute. Attributed grammars pass contextual information either between parent and offspring nodes or between siblings in a syntax tree.

More formally, an *attributed grammar* is a context-free grammar $G = (N,T,P,S)$ in which

1) there is a finite set of attribute names and domains $A = \{(a,Da),\ldots\}$ where Da is the (possibly infinite) domain of the attribute a.
2) for each symbol $X \in (N \cup T)$ there are two disjoint subsets of A. $I(X)$ is the set of *inherited* attributes of X and $S(X)$ is the set of *synthetic* attributes of X.
3) for each member $X0::=X1\ldots Xn \in P$, there exists an indexed set of *attribute evaluation rules* for each element of

   $$S(X0) \cup I(X1) \cup \ldots \cup I(Xn)$$

   containing functions whose arguments occur in

   $$S(Xi) \cup I(Xi) \quad \forall\, i \in 0,\ldots,n$$

Notational conventions used in attribute evaluation rules are as follows (the production involved is implicit in the use of a function):

let B.a denote the attribute named "a" for symbol B in the grammar.

let B(j).a denote the attribute "a" of the j-th occurrence of B in a particular production. Occurrences are counted from the left after concatenating the left and right sides of a production. The first occurrence is indicated by j=0. The subscript is dropped for convenience when B occurs only once.

let B(j).a := Fi,k(<args>); represent the rule evaluating the attribute "a" of the j-th occurrence of B in production i. The subscript k indexes the function within the production. <args> includes other attribute occurrence references from that production.

For example:

```
<HEAD> ::= <HEAD>  <TAIL>
   <HEAD>(1).Position := <HEAD>(0).Position                   (1)
   <HEAD>(0).Length := <HEAD>(1).Length + <TAIL>.Length
   <TAIL>.Position := <HEAD>(0).Position + <HEAD>(1).Length
```

In this example, the Position attribute is inherited, and the Length attribute is synthetic.

Chirica and Martin's [CM76] restrictions on attribute grammars (which results in no loss of power) requires the attribute occurrences in a production to be divided into two disjoint classes.

The <u>recipients</u> are the targets of the associated attribute evaluation rules. For a given production i:

A ::= B1 ... Bn

The set of recipient attribute occurrences in i is

$\mathrm{Rec}(i) = \{(a,X) \mid X=A \text{ and } (a,Da) \in S(A) \text{ or}$
$X=Bj \text{ and } (a,Da) \in I(Bj),\ 1 \leq j \leq n\}.$

The <u>donors</u> are the attributes that appear as arguments of attribute evaluation functions. The set of donor attribute occurrences in i is

$\mathrm{Don}(i) = \{(a,X) \mid X=A \text{ and } (a,Da) \in I(A) \text{ or}$
$X=Bj \text{ and } (a,Da) \in S(Bj),\ 1 \leq j \leq n\}.$

For purposes of translation and compilation, an extension to Knuth's definition of attributed grammars is desired. Terminal symbols may have only synthetic attributes. The values of the attributes are determined solely by the appearance of the token in the input string. Terminal attribute evaluation is typically done by a scanner or lexical analyzer. For instance, the synthetic attributes of a constant might be its type and its value, those of an identifier, its name and its hash value. In the same vein, the grammar's goal symbol may have inherited attributes. These attributes are of necessity constants. They are useful to set or reset translation options for the evaluator or to initialize attributes such as a symbol table.

The Use of Attributed Grammars

Many language defintions (eg. PASCAL [HW73]) state that name scoping and some type checking is to be accomplished in syntax analysis but do not specify how; such context-sensitive analysis is certainly out of the realm of cfg parsers. Consider the following (typical) productions of a statement oriented language that contains reals, integers and Booleans:

1) <FACTOR> ::= <FACTOR> * <PRIMARY>
2) <FACTOR> ::= <PRIMARY> (2)
3) <PRIMARY>::= <ID>

The operand type of a <PRIMARY> that was found to be an <ID> is a function of the name of the <ID>, a synthetic attribute, and the environment of the <PRIMARY>, an inherited attribute. If the name is not declared in the environment or could never have been assigned a value within the environment, an attribute denoting illegal use could be set "true" (if the language desires it), as is often done in an ad-hoc manner in conventional compilers. In a multiplication, the type of the resulting <FACTOR> is clearly determined by the synthesized types of its constituent <PRIMARY> and <FACTOR>. Again an attribute denoting illegal use could be set if the types of the two operands are incompatible. Thus attributed grammars can solve name scoping and type determination syntactically in a natural, concise and complete fashion.

Translation may occur in various fashions with attributed grammars. One such scheme is to have a "code sequence" attribute for the start symbol of the grammar (call it <PROGRAM>) be the final translation of the program. If the following productions are added to (2)

```
4) <TERM>    ::= <TERM> + <FACTOR>
5) <TERM>    ::= <FACTOR>                         (2´)
6) <PRIMARY>::= ( <EXPRESSION> )
```

as well as those for an assignment statement, "code sequence" could be a synthetic attribute of <EXPRESSION>, <TERM>, <FACTOR>, and <PRIMARY> that would be built by insertion and concatenation functions. In production (4), for example, the code sequence of the second occurrence of <TERM> could be followed by that of the <FACTOR>, and code for addition concatenated to its end depending (possibly) on temporary locations (again synthetic attributes) and the result transmitted to the left-hand side <TERM>.

A particularly attractive application of attributed grammars in syntactic analysis is presented by Milton

[Mil77]. He defines attributed parsers in which contextual predicates are defined for each production in the grammar. For a given production to apply at some point in the parse, the predicate (which is defined in terms of attributes local to the production) must yield a true value. Milton describes another predicate used to disambiguate context-free grammars. If the look-ahead function for a particular parsing scheme fails to differentiate between production choices, a disambiguating predicate on the root of the productions and the look-ahead symbols is used to make the choice. He modifies several parsing algorithms to perform attributed parsing and shows how an underlying context-free grammar can be made much smaller by including attributes in many parsing decisions.

Other methods have been suggested to extend context-free syntax checking. Property grammars [SL69] allow synthetic information flow to occur when productions are recognized in a bottom-up parse. Decisions can be made about the legality of offspring according to the properties (attributes) of the offspring. Property grammars can always be replaced by purely synthetically attributed grammars.

Indexed grammars [Aho68] generate a class of languages properly located between context-free and context-sensitive languages. In this scheme, inherited indexes are passed to offspring as a production is applied to an indexed non-terminal. The indices control the generative capabilities of the offspring. Recognition procedures for indexed grammars using an SLR(1) skeleton are discussed by Solomon [Sol77].

W-grammars, introduced by Van-Wijngaarden [WMPK69], are a means of specifying context-sensitive syntax information (as well as semantic information). They consist of two levels of grammar: metaproductions and hyperrules. The metaproductions are context-free rules for generating protonotions. Hyperrules are templates from which a potentially infinite number of context-free syntax rules can be derived. The unbounded number of program syntax productions gives W-grammars their context-sensitive power. Again W-grammars are generative and not well suited to automatic recognition techniques.

Previous work on semantic analysis has fallen into two categories. String-to-string mappings occur in the realm of the first category, syntax-directed translation schemes (SDTS). In this class are syntax-directed transducers [LS68], generalized syntax-directed translations [AU71b], pushdown assemblers [AU69b] and others [AU69a; AU71a]. Each scheme is string-oriented and is based on pushdown automata recognizers.

The second category, semantic specification systems, generally extend or replace the conventional syntax schemes as a framework on which to base semantics. In W-grammars, the source program and input file are analysed together to produce an execution sequence, a technique not normally used in compilation strategies. Vienna Definition Language (VDL) [Weg72] consists of tree-building predicate functions. VDL semantics is based upon an interpreter that acts upon the VDL syntax tree. While Marcotty, Ledgard and Bochmann [MLB76] find that production systems and attributed grammars do not encompass semantics as completely as the other methods, this is in large measure due to the fact that these definition techniques were designed to be used in translators. The orientation of attributed grammars and production systems makes them well suited for compiler construction.

Problems with Attributed Grammars

There are several problems inherent in evaluating the attributes associated with the symbols of a syntax tree. First, the resulting definitions must be non-circular. A circular attributed grammar is one whose language contains a syntax tree with an attribute that depends functionally upon its own value. Knuth has given a circularity test for attributed grammars [Knu68a], and others have given algorithms that reject all circular attributed grammars (as well as some other grammars that do not fit their evaluation schemes) [Boc76; KW76]. Circular grammars are not of interest and are eliminated from all evaluation schemes.

Another problem in attribute evaluation is finding an order in which to evaluate the attributes of a sentence. One group of methods, known as tree-walk evaluators, relies on the prior completion of the syntactic analysis of a sentence. The first work in this area was done by Isu Fang [Fan72], a student of Knuth. His proposal was to take a syntax tree, start at the root and traverse the tree in a depth-first search evaluating those attributes ready to be evaluated. Those attributes that cannot yet be evaluated are delayed until all the attributes they are dependent upon are evaluated. Full evaluation may require several trips to each node in the syntax tree. Fang´s approach is non-deterministic because the number of visits to each node of the syntax tree as well as the order of attribute evaluation depend upon the sentence itself.

Later work by Jazayeri [JW75] and Bochmann [Boc76] produced evaluators that again require the entire syntax tree but rely on a prior dependency analysis of the attributes. They determine the number of passes over any syntax tree necessary to evaluate all the attributes in any sentence of a language being processed. For each pass, the algorithms identify which attributes are ready to be evaluated. Bochmann considered only left-to-right preorder passes. Jazayeri´s method is more general in that he considers alternating left-to-right and right-to-left passes. This scheme accepts more attributed grammars than

Bochmann's. Neither accepts all non-circular attributed grammars, but they can be considered deterministic in that the same order of attribute evaluation is applied to any sentence generated by an acceptable grammar.

Kennedy and Warren [KW76] describe an evaluation method for attributed grammars that frees itself from strictly left-to-right (or right-to-left) passes over a syntax tree. They return to a tree-walk evaluator that starts at a root and visits nodes carrying down inherited information or bringing back up synthesized information. The symbol visitation order is predetermined for each production and is dependent only upon the immediate offspring of a node; thus the entire subtree below an offspring is not considered. A node is marked with the state of the evaluation of the attributes of its root and immediate offspring. The combination of the state of the node and the evaluated inherited attributes triggers an action sequence that further evaluates recipient attributes whose donors are now available. They present an algorithm that compiles attributed grammars into the necessary tree-walk evaluators for all attributed grammars that are absolutely non-circular, a subset of the non-circular grammars. Warren claims that this approach can be extended to all non-circular grammars at the cost of increased subtree analysis.

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

The top-down parsing methods like LL(k) recognize nodes of a syntax tree in a different manner. All parents are recognized before their offspring, and each offspring is found before its siblings to the right. The LL(k) parse performs a depth-first left-to-right walk through a syntax tree. Any attributed grammar that is LL(k) and can be evaluated by a single depth first left-to-right walk for any sentence in the language can be evaluated during an LL(k) parse. Details of a stack machine that performs the evaluation are presented in Lewis, *et al* [LRS74]. The machine maintains a single attributed state and a stack of attributed semantic nodes. The machine state represents the most recently recognized grammar symbol and its attributes, and the top stack node represents the grammar symbols to the left of the state symbol in the most recently predicted production as well as the left-hand side of that production. As another symbol is recognized, its recipient attributes' donors must be a part of the current top stack node and the machine state and can be computed. When a predicted production is completely recognized, the top stack node is popped and the attributes associated with the left-hand side are transfered to the machine state. Structure recognized and utilized prior to the current sentential form is deleted as the parse progresses. The class of attributed grammars allowed by this method is termed <u>L-attributed</u> and is the same class described by Bochmann as evaluable in a single pass through the parse tree [Boc76].

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

> An attributed grammar G = (N,T,P,S) is <u>S-attributed</u> if all attributes are synthetic.
>
> An attributed grammar G = (N,T,P,S) is <u>L-attributed</u> if for each production in P of the form:
>
> A ::= X B Y
>
> where $X,Y \in (N \cup T)^*$,
>
> 1) the synthetic attributes of A are only dependent upon the inherited attributes of A and arbitrary attributes of the symbol B and symbols in X and Y, and

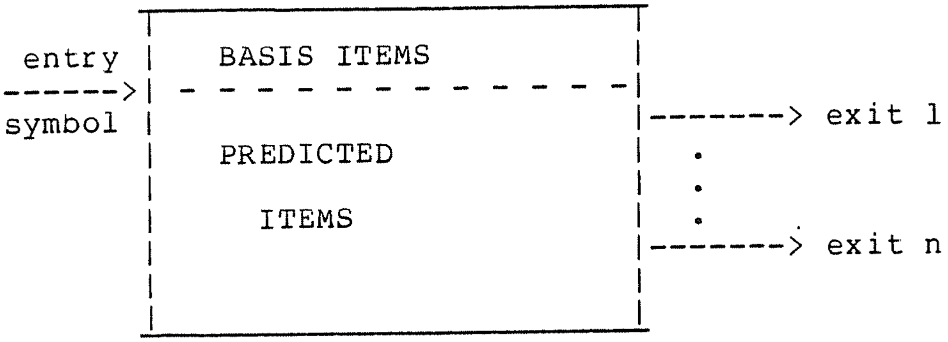
2) the inherited attributes of B are only dependent upon the inherited attributes of the symbol A and arbitrary attributes of the symbols in X.

These two classes of attributed grammars, S-attributed and L-attributed, do not include many desirable properties of full attributed grammars that prove useful in the translation of common programming languages. Forward references, for example, cannot be handled in either class without undue complication. A forward reference can be defined as the use of a name of an object in a portion of code before the definition of that object has occurred. Forward referencing is very common in most programming languages; it occurs in forward branching GO TO statements, procedure declarations after their invocations, and PL/I type declarations (which may occur anywhere in a given scope). To satisfy forward references with an L-attributed grammar, all object references and definitions must be accumulated synthetically until no more are syntactically allowed to occur. At this point, the gathered references must be updated according to the definitions. With unrestricted attributed grammars only the definition list (acting as the traditional symbol table) need be collected and can become an inherited attribute to any node needing it. The function updating a single specific kind of reference from the definition list is far simpler than that applied to reference types in general.

Relaxing the ordering restrictions required by the L-attributed class would thus allow elegant solutions to many forward referencing problems. The LL(k) parse restriction is often found unwieldy as well. The LR(1) family of languages properly contains the LL(k) languages for all k and is found more natural by many.

The Configuration Set and GLC Parsing

LR(k) parsing is based on pushdown states that keep track of every possible production to which the next input symbol could belong. The states, sets of parse configurations called configuration sets, form a parse graph connected by symbol transitions. Each configuration set other than the start state is partially characterized by an entry symbol, that symbol in the grammar (terminal or non-terminal) that was recognized and caused the transition into the state. Each parse configuration is an item which consists of a production and a configuration symbol (".") that is kept between that portion of a production already recognized and that yet predicted. The basis of this set contains the productions that could have generated the entry symbol; it includes completed and incomplete items. The configuration set and types of items are illustrated in figure 1.1. For a full development on the construction of configurations sets for LR(k) parsing, see [AU73].



configuration set

[LHS ::= . RHS] (predicted item)

[LHS ::= RHS .] (completed item)

[LHS ::= R1 ... Ri . Ri+1 ... Rj] (incomplete item)

configuration items in three possible forms

Figure 1.1 Configuration Set and Items

A hybrid parsing scheme has advantages of both the top-down and bottom-up schemes that are useful in attribute evaluation. For parse-time evaluation, monitoring the evaluation of attributes within partially parsed production instances necessitates a parsing technique based on configuration sets. The generalized left-corner (GLC) parsing technique described by Demers [Dem77] is as powerful as LR and fits each of these needs. The left-corner parsing scheme actually generalizes both the bottom-up and top-down parsing techniques. The following definitions leading to a GLC parser are adapted from Demers [Dem77].

A *recognition rule grammar* based on a cfg $G'=(N,T,P',S)$ is a cfg $G = (N \cup \hat{N},T,P,S)$ where

a) $\hat{N} = \{\hat{i} \mid 1 \leq i \leq |P'|\}$ is the set of *recognition symbols*.
   N and $\hat{N}$ are disjoint.

b) P contains $\{\hat{i} ::= e \mid \hat{i} \in \hat{N}\}$ and
   for each i, exactly one production of the form

$$A ::= X \ \hat{i} \ \ Y$$

   for the i-th production: $A ::= X\ Y \ \in P'$ with
   $X,Y \in (N \cup T)^*$
   e represents the empty string
   X is termed the *left corner* and
   Y is termed the *trailing part* of production i.

The GLC configuration sets differ slightly from those of LR. In particular, the configuration symbol does not move beyond the recognition symbol since the production instance is fully recognized at that point and the trailing part symbols are predicted. In GLC parsing, as in LR, a parse stack is used to keep track of the incompletely recognized items. A previous state is re-entered when it becomes the top stack node, and recognition at that level of the syntax tree is continued. In GLC configuration sets that represent predicted symbol states, there is an item denoting the prediction of A, [::=.A], that represents the independence of this item from the predicting production. A state representing a prediction of symbol A is denoted Q(A).

A <u>generalized</u> <u>left-corner</u> <u>parser</u> with k-symbol lookahead for a recognition rule grammar $G = (N \cup \hat{N}, T, P, S)$ is a triple M = (States,Action,Goto) where

a) <u>States</u> is a finite set of states containing at least distinguished states Q(S) and Q(A) for each A that occurs in a trailing part of some element of P.

b) the function <u>Action</u>: States X T^k -->
{error,pop,shift} U $\{\text{announce } i \mid \hat{i} \in \hat{N}\}$
Action describes how the stack is to be manipulated in a given configuration.

c) the function <u>Goto</u>: States X (N u T) --> States.
Goto(Q,X) = closure({items I|I=scan(I′) and X
follows . in I′ ∈ Q})
<u>scan</u>(I) is the item that results from item I by moving the configuration symbol past the symbol to its right, but never beyond a recognition symbol, and is undefined otherwise.
The <u>closure</u> of a set of items Q is the smallest set of items containing Q and such that if [A::=X.B Y] or [::=.B] is in closure(Q) then

$[B ::= \underline{.} Z]$ is in closure(Q) for each $B ::= Z$ in P (where $X,Y,Z \in (N \cup T)^*$).

A GLC parser starts as a top-down parser in a state Q(S) to predict the goal symbol S. It then recognizes left-corners (which may be empty) in a bottom-up fashion. After left corner recognition, the parser predicts the symbols in the trailing part as it parses in a top-down fashion. A formal algorithm is given by Demers [Dem77] and a revised algorithm is presented here in chapter 4.

Single-Pass Compilations

In a single-pass compilation, all lexical, syntactic, and semantic analysis of a source program is done on the same scan of the input stream. Multipass schemes generally perform several scans over the source and/or restructured representations of it. There are several arguments that tend to favor a single-pass compilation scheme rather than the multipass scheme.

The multipass scheme is potentially slow and expensive, since much intermediate storage is needed and many mass storage I/O requests are required to find and update information. (Such overhead may be less obvious in systems with virtual storage.) Single-pass compilations are an attempt to overcome this problem; some languages, e.g. PASCAL and SAIL, have been designed with a single pass (excluding most optimization) in mind. Fang concludes [Fan72] that while his multipass scheme seems fine for language testing, it is inadequately slow for production use. His FOLDS implementation, while capable of creating a translator for SIMULA 67, can only handle SIMULA programs of up to 100 lines due to space restrictions.

With attributed grammars in particular, a single-pass translation scheme is able to detect non-context-free syntax and semantic errors at an earlier point than a multipass scheme. There is a distinct advantage in detecting errors as soon as possible. If error detection is delayed several passes, a compiler most likely does a large amount of preparation for code generation and perhaps optimization that may be fruitless due to the error(s) recognized. Late error recognition also hinders error messages based on source code lines and any chance of error correction [FMQ77; Iro63].

When attributes are desired for syntactic analysis in methods analogous to Milton's [Mil77], it is necessary to evaluate attributes on-the-fly with the parse. His work supports combining syntax analysis with attribute evaluation in a single-pass compilation.

There are programming languages and computer environments for which single-pass compilations are impractical. Multipass compilations are often necessitated by small computer memories that require most compiler tables and intermediate results to reside on mass storage. The mass storage is often most easily managed through the use of sequential files. Each subsequent reference to the intermediate results requires another file traversal and thus another compiler pass. Such languages can still benefit in a reduction in the number of analysis passes required by making use of the techniques developed here. In some cases, the structure of the language itself leads to the necessity of many compiler passes. Languages that are difficult to compile are perhaps more difficult for a user to understand and read as well.

The Semantic Stack

The theory of single pass compilation often refers to the notion of a semantic stack that is maintained and operated upon by a semantic analyzer [Gri71; WW66; LRS76]. Nodes on a semantic stack typically contain descriptors of syntactic entities recognized by the parsing unit. A descriptor (the semantic content of a node) is used eventually in the evaluation of the semantics or translation of the sentence as a whole. Since syntactic entities are found during syntactic analysis by a parser, the semantic stack includes enhancements to the significant nodes of a parse stack. A semantic stack is an attempt to compact the information known so far about a syntax tree and its leaves and retain it in a more manageable and accessible form.

Normally, only the root of a fully recognized subtree is retained to represent the structure and semantics associated with the subtree.

A natural way to envision semantic stack nodes is as attributed grammar symbols. Attributed grammars provide the semantic functions describing how the attributes of a semantic stack node are to be evaluated. With proper restrictions placed upon an attributed grammar, various parsing/semantic strategies can be implemented. Both the L- and S-attributed evaluators are capable of evaluating restricted attributed semantic stacks. With a GLC parser-based evaluator, advantages of both bottom-up and top-down semantic evaluation are available.

The problem with current methods of dealing with a semantic stack is that in practice, more information is needed to evaluate attributes or make semantic decisions than is available. A common recourse is to create auxiliary tables and treat them as global variable attributes, or to provide links through the translated output that are to be filled in when a semantic attribute is eventually evaluated. While such "fixes" solve the problem at hand, they also tend make the compiler obscure and are certainly less amenable to efforts in proving assertions about the translator or language or to automating semantic evaluation.

To use more general attributed grammars in a single pass compilation, the concept of the semantic stack can be extended. The difficulty with an attributed semantic stack is that a particular attribute may not be evaluable when a node is removed from the stack when a reduction is made in a bottom-up parse or a match is made in a top-down parse. The stack in the bottom-up translation could be a stack of trees, a semantic forest stack. The roots of the forest correspond exactly to the recognized portion of the right sentential form, modelling the current position of a parse. A reduction simply connects the top nodes (subtrees) of the stack as offspring of the new root which replaces those nodes (subtrees) as the top of the stack. This technique is in a sense more primitive since it simply rebuilds the full syntax tree (just what the stack was used to avoid), but it does offer a structure through which all attributes of a attributed grammar could be evaluated. If recipient attributes are not immediately evaluable when a node is recognized, a visit by an evaluator will be made at a later time when the necessary donors become available.

A compromise between the full semantic tree (which grows out of the forest stack) and the usual semantic stack is one in which subtrees only occur under semantic nodes that are not fully evaluated. Once a subtree´s root is fully evaluated, the subtree below the root can be effectively discarded. If a node is not fully evaluated, those offspring with the needed donors must be retained. Each offspring will either be a fully evaluated node in the sense above (without any subtree) or another partially unevaluated subtree. In bottom-up parsing, the modified stack thus allows a full range of possibilities simplifying to the simple semantic stack in the case of S-attributed grammars and potentially expanding to the full syntax tree in the case of very complex attribute grammars.

Attribute Evaluation

The scheme for attribute evaluation developed in this thesis is based on a semantic forest stack built and maintained by a GLC parse-time evaluator. A single-pass evaluator can recognize sentence structure and evaluate all attributes simultaneously. When cases exist in which an evaluator must retain subtrees and return to them to complete evaluation, the single-pass criterion appears violated. However, subtree visits are only made to selected portions of the sentence, and the attributes evaluated in the visits are used in the remainder of the single pass.

A modified semantic forest stack as described offers a vehicle to evaluators that will be shown to have three important properties:

(1) It contains a structure in which attributed grammars can be implemented elegantly and practically.

(2) It is built on a sound theoretical framework (stacks, trees, and well-studied parsing techniques) that is essential to establishing its properties.

(3) The implementation is space-efficient because only as much structure as is absolutely needed is used in attribute evaluation.

This dissertation addresses the significant problems that occur in attempting to make the modified stack a workable solution. Chapter 2 defines and investigates the notion of left corner attribute availability. Boolean availability vectors are used to determine at which points in a parse particular attributes are ready to be evaluated. In chapter 3, a machine (the attributed pushdown processor) is developed to construct and manage a semantic forest during a parse. The use of attribute availability in constructing action sequences for the processor is formalized in chapter 4. Methods are necessary to permit eventual evaluation of those attributes not immediately evaluable with the parse. The unevaluated subtree problem is the topic of chapter 5. Finally, grammar classes that work well with the modified stack (i.e. keep subtree retention to a minimum) are identified.

Chapter 2

Attribute Availability

To avoid delaying all semantic evaluation until syntactic analysis is completed, it is necessary to investigate how and when semantic evaluation can proceed in step with a parse. Several special cases were brought to light by Lewis, Rosenkrantz and Stearns [LRS74]. They showed that certain attributed grammmars (L- and S-attributed) can be evaluated by visiting nodes of a syntax tree in the same order that they are recognized by a particular parsing algorithm.

A solution to alleviate the limitations of the two methods of Lewis, et al mentioned above is one that encompasses the power of both. It is desirable to have this solution be flexible enough to extend naturally to more general attributed grammar classes (like those handled by Kennedy and Warren). An evaluator that allows both L-attributed and LR(k) grammars would be attractive as a starting point, though it would not cleanly solve the forward reference problem.

Availability of Attributes During a Parse

An essential objective of a parse-time evaluator for attributed grammars is to evaluate semantic attributes as soon as possible during the parse. To know when specific attribute evaluation can take place, a formal concept of attribute availability is necessary. The availability of an attribute reflects whether it is ready to be evaluated at a given point in an evaluation scheme. An attribute is available for evaluation when its evaluation rule is known and the donor attributes referenced by the function are evaluated. In the rest of this chapter, the concept of LC-availability is considered -- the point in an incomplete GLC(k) parse at which any given attribute for an instance of a specific grammar symbol is available. The evaluation points are identified by augmenting parse states to express attribute availability. The augmented states are termed availability-extended parse states. The GLC parsing mechanism can then be extended to perform attribute evalutions shown possible in the states.

During a parse only an incomplete description of a derivation is formed at any particular time. With an incomplete structure, the attributes of some nodes might be recognized as available for evaluation while many others are not. Dealing with incomplete syntax trees forces attribute donors to be considered unavailable until their corresponding grammar symbols are recognized and incorporated into the (incomplete) structure. At each intermediate step in the process of syntax tree construction, the tree becomes more complete and more attributes may become available.

Attribute availabilities for a given state of a parse can be determined before parsing and the availability of the attributes for a given symbol can be used in choosing parse state-to-state transitions. In this manner, available attribute lists can be maintained for each item of each state. Because different subtrees can result in different synthetic attributes being available in the root of a subtree, each potential combination of evaluated attributes for each grammar symbol must be considered in parse state transitions.

> A _transition_ _pair_ (A,v) is an element of (N u T) x (0,1)* where $|v|$ is equal to the number of synthetic attributes of A. A transition pair, rather than the symbol alone, is used in the selection of the next state of the parse.

The synthetic attributes are considered in transitions because they characterize the structure generated by a grammar symbol.

It is sufficient to consider all combinations of synthetic attributes available in the transition pairs of a given symbol. However, for each transition possibility, a distinct successor state exists. The successor states are syntactically equivalent but they differ semantically due to the variance in attributes available for use as donors. This approach could lead to a serious combinatorial explosion in the number of availability-extended parse states needed. Fortunately not all possibilities occur in general, and it can be determined by an iterative analysis just which attribute availabilities can in fact occur during translation.

In parse states that indicate the recognition of an instance of a production, the availability of donor attributes can be used to determine available recipient attributes. Recipient attribute availabilities of completed items are used to determine legitimate transition pairs for the left-hand side of the item. The availability computations are iterative because each time a computation is complete, new recipient attribute availabilities may be discovered. Newly discovered availabilities require at least partial recomputation of the availability-extended parse graph, since different and new transitions are possible. The iterations must halt, since for each pass, recipient attribute availabilities for each completed item are monotonically non-decreasing and there exist only a finite number of possibilities.

For each production and its associated attribute evaluation rules, a Boolean dependency matrix can be formed.

> Define MDi to be the dependency matrix for production i of size |Rec(i)| X |Don(i)|. A row in the matrix represents the dependence of a recipient attribute upon its donors. MDi is defined as follows:

$$MDi(m,n) = \begin{cases} 1 & \text{if the m-th recipient attribute has} \\ & \text{the n-th donor attribute as a} \\ & \text{argument in the associated defining} \\ & \text{function Fi,m.} \\ 0 & \text{otherwise.} \end{cases}$$

The dependency matrix is used in the calculation of specific attribute availability for a production configuration item. It represents the dependency graph [Knu68] with ones representing a directed arc in the graph between two attributes.

Using the standard concept of an item of a configuration set [AU73], with the notation:

$$[\ A ::= B1 \ldots Bj\ \underline{.}\ Bj+1 \ldots Bn\ ]$$

the notion of item-wise attribute availability can be developed. In a parse state, two Boolean availability vectors are associated with each item. The vectors have a position for each attribute in either Don(i) or Rec(i) for the production i on which the item is based.

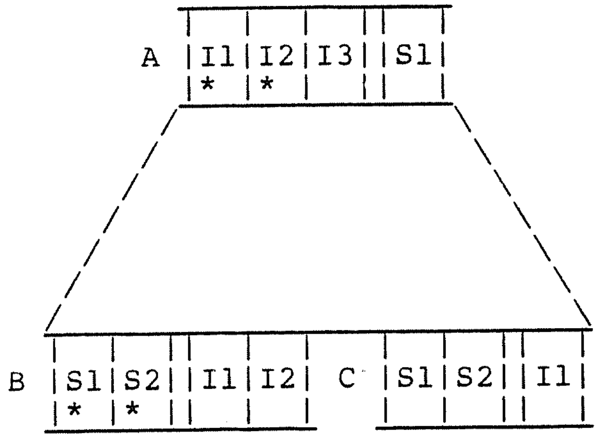
> The Boolean vectors <u>daav(I)</u> (donor attribute availability vector) and <u>raav(I)</u> (recipient attribute availability vector) represent for item I the guaranteed availability of donor attributes and the potential availability of the recipient attributes, respectively.

Algorithms are developed in this chapter to calculate the two vectors at the time of parser generation. Potential availability is only used by an evaluator when it knows exactly which item in the state truly represents the parse configuration. In the case of completed or recognized items, the potential is guaranteed and is used for scheduling attribute evaluation.

A different notation for availability is also useful. In availability-extended items, each availability vector is partitioned to the grammar symbols. Appearing with a symbol is a parenthesized Boolean vector; the daav for a symbol's attributes precedes its raav and the two are separated by a vertical stroke. Due to this construction, inherited attributes are represented before synthetic attributes in the left-hand side of an item and after synthetic attributes in the right-hand side. Attributes brought into a production by a symbol are indicated symbolically before those evaluated at the production. For example, the string:

[A(110|0) ::= B(11|00) $\underline{.}$ C(00|0)]

represents an item whose production is A ::= B C . The symbol C is predicted and the first two of three inherited attributes of A and both synthetic attributes of B are available. This string corresponds to the diagram below in which available donors are marked with an asterisk.



The vector of donor attribute availability for an item I, daav(I), specifies which attributes in the evaluation have already been calculated and are ready to be used to evaluate new attributes. During parser generation, the daav(I) is used in conjunction with MDi (where i is the number of the production used in I) to predetermine which recipient attributes can next be evaluated. Equation (1) describes recipient attribute determination Overscore denotes Boolean complement.

$$\overline{raav}(I) = MDi \times \overline{daav}(I) \tag{1}$$

The equation follows dependency arcs from previously available attributes to find newly available attributes.

As an example:

Suppose for the item I = [A ::= B $\underline{.}$ C]

I(A) = {I1,I2,I3}

S(A) = {S1}

S(B) = {S1,S2}

I(B) = {I1,I2}

S(C) = {S1,S2}

I(C) = {I1}

the associated attribute functions of A ::= B C are:

A.S1 := B.S1 + C.S1;

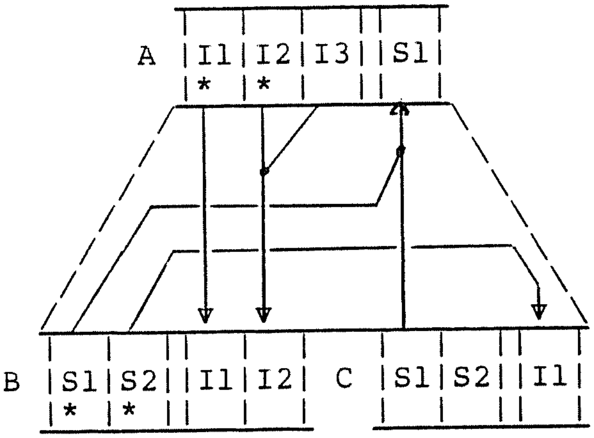
B.I1 := A.I1;

B.I2 := A.I2 + A.I3;

C.I1 := B.S2;

the dependency matrix MDi and graph are:

$$\begin{vmatrix} 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{vmatrix}$$



If it is determined that at this configuration of the parse, the avalability characterization of the item I is:

[A(110|0) ::= B(11|00) . C(00|0)],

then the daav(I) = (1101100). Applying equation (1) gives:

$$\begin{vmatrix} 1 \\ 0 \\ 1 \\ 0 \end{vmatrix} = \begin{vmatrix} 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{vmatrix} \times \begin{vmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{vmatrix}$$

or: raav(I) = (0101). This raav specifies that attribute I1 of B and attribute I1 of C are available for evaluation at this point of the parse under the assumption that I correctly represents the parse configuration.

To find the daav vector for items requires a more detailed look into the relationships among items in a particular configuration set and the relationships between pairs of configuration sets.

When a left corner of a production is recognized in a GLC parse, the synthetic attributes available in the left corner can be used to determine the availability of recipient (inherited) attributes that occur in the trailing part. Some of the trailing part recipient attributes may have donors in the trailing part attributes as well, but no item in any GLC state explicitly considers having parsed beyond the recognition symbol. For this reason additional states are needed in the GLC availability-extended state set. These states each include only a single item (since the current production is known) and contain the configuration symbol ($\underline{.}$) to the right of the recognition symbol. The extra states, called *ghost states* since they cannot directly be entered during a parse, are of the form:

$$\boxed{[\ A ::= X\ \hat{\imath}\ Y\ \underline{.}\ Z\ ]}$$

Ghost states exist for every production, $A ::= X\ \hat{\imath}\ Y\ Z$ in P, and for every combination of $Y \in (N\ u\ T)+$ and $Z \in (N\ u\ T)*$. Those ghost states with the configuration symbol to the right are referred to as *completed states*.

Three special attribute availability vectors take part in the LC-availability calculations.

1) The recipient attribute availability vector associated with a trailing part symbol is a <u>predicted</u> <u>shape</u> for that symbol.
2) The recipient attribute availability vector of the symbol on the left-hand side of a production is an <u>entry</u> <u>shape</u> for the symbol.
3) A <u>propagated-daav</u> for a symbol is the donor attribute availability vector associated with that occurrence of the symbol in any predecessor state.

For the development of the availability algorithms outlined above, several functions and associated notation are necessary.

$$\begin{aligned} \text{Let } CS(X) &= \emptyset \text{ if } X = e \text{ (the empty string), or} \\ &= |S(X)| \text{ if } X \in (N \cup T)\text{, or} \\ &= |S(X1)| + CS(X') \text{ if } X \in (N \cup T)+ \\ &\quad \text{and } X = X1\,X',\ X1 \in (N \cup T). \end{aligned}$$

The function CS returns the number of synthetic attributes for a string of symbols X.

Similarly, CI(X) returns the number of inherited attributes for a string X.

Let raav(I/B,j) and daav(I/B,j) represent the portion of the raav(I) and daav(I) respectively that is associated with the j-th occurrence of the symbol B in the concatenation of the two sides of the production of I. Counting of symbols starts at zero.

Let [0]n and [1]n represent bit vectors of length n containing all 0´s or all 1´s respectively.

Let a dot (.) represent the concatenation operator for bit vectors and <u>or</u> represent an inclusive "or" operation.

For example: [0]3.[1]2 <u>or</u> (10000) = (10011)

The Prediction Step

This first step initializes the predictive states with predicted donor attribute availability vectors. The predictive states are a logical starting point, since they have no entry arcs. For a state Q(A), syntactically identical configuration sets are formed, one for each possible predicted shape found for A. If an item in one of these states has A as its left-hand side symbol, then that A is recognized as referring to the same instance of the predicted trailing part symbol A unless the grammar is left recursive in A. When the left-hand side A of an item is known to be the predicted symbol, its donor availability is identical to the predicted shape of A. Thus, if I is one such item and v´ is the predicted shape of A, then

$$\mathrm{daav}(I) = v'.[0]\mathrm{CS}(X)$$

where the production for I is A ::= X, $X \in (N \cup T)^*$.

An example of the prediction step follows:

Let Q(<stmt>) =

[::= $\underline{.}$ <stmt>]

[<stmt>::=$\underline{.}$<label>$\hat{1}$:<stmt>]

[<stmt>::=$\underline{.}$$\hat{2}$ begin<block>end]

[<stmt>::=$\underline{.}$<var>$\hat{3}$:= <expr>]

[<label>::= $\underline{.}$ identifier $\hat{4}$]

[<var>::= $\underline{.}$ identifier $\hat{5}$]

Let I(<stmt>) = {envir,block#,forw_refs}

and S(<stmt>) = {lab_defs,code,code_len,int_envir}

Let the predicted shapes of <stmt> be:

{(110),(111)}

The prediction step creates the two following availability-extended GLC(0) states:

```
[::= . <stmt>(0000|110)]
[<stmt>(110|0000)::=.<label>(0|0)1:<stmt>(0000|000)]
[<stmt>(110|0000)::=.2 begin<block>(0000|000)end]
[<stmt>(110|0000) ::=.<var>(00|)3 := <expr>(00|00)]
[<label>(0|0)::= . identifier(00|) 4]
[<var>(|00)::= . identifier(00|) 5]
```

and

```
[::= . <stmt>(0000|111)]
[<stmt>(111|0000)::=.<label>(0|0)1:<stmt>(0000|000)]
[<stmt>(111|0000)::=.2 begin<block>(0000|000)end]
[<stmt>(111|0000) ::=.<var>(00|)3 := <expr>(00|00)]
[<label>(0|0)::= . identifier(00|) 4]
[<var>(|00)::= . identifier(00|) 5]
```

As <stmt> is moved from the right-hand side to the left-hand side, its raav is used as a daav. Attributes shown to be evaluable are assumed to be evaluated immediately and are available for use as donors.

The Propagate Step

The successor states to the predictive states and their successors in turn are affected by the daav's of their predecessors. If an attribute is available in a configuration item of one state Q, it is still available in the next state entered by the parse, an availability-extended version of the state Goto(Q,A). Each item in the basis of a successor state exists because it is a result of the scan function applied to an item in its predecessor. Thus availabliliy in the predecessor states propagate to all successors. Each state represents the recognition of one more symbol in the derivation tree (the entry symbol of the state). The availability of the attributes of this entry symbol is significant for each item in the basis. The propagate step is applied recursively to the states reachable from each prediction state. Whenever more than one entry shape is possible, a syntactically similar state is formed for each distinct transition pair.

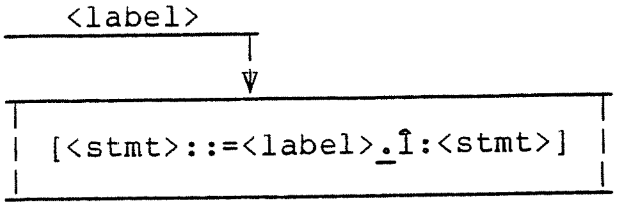
Successor Item daav

Both the propagated-daav and the entry shape are used to determine successor item donor attribute availability vectors. Consider a state Q', the successor to Q over symbol $B \in (N \cup T)$, i.e., $Goto(Q,B) = Q'$. If B has an entry shape v, and the item $I' \in basis(Q')$ is scan(I) for some $I \in Q$, and $I' = [A ::= X\ B\ \underline{.}\ Z]$ then the daav for I' is found as specified in the propagate equation (2).

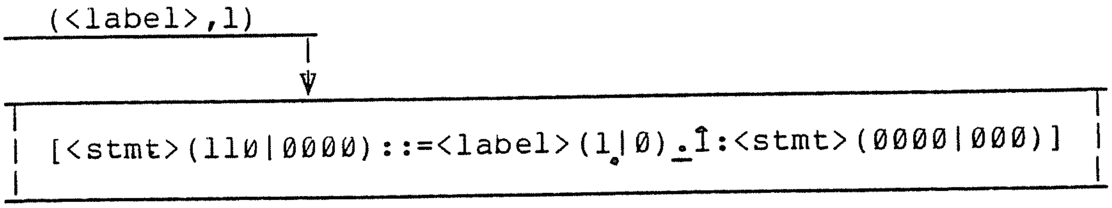
$$daav(I') = ([0](CI(A)+CS(X)) \;.\; v \;.\; [0]CS(Z)) \;\underline{or}\; daav(I) \qquad (2)$$

To continue the previous example:

The successor to Q(<stmt>) over transition <label> is:



Performing the propagate step from the first availability-extended Q(<stmt>) to the availablity-extended state Goto(Q(<stmt>),<label>) with entry shape vector = (1) for <label> gives:



All attributes that were available in the predecessor and those that caused the transition are included in the resulting item.

Entry Symbol Attribute Availability

If the entry symbol to the state is a terminal, then all attributes are synthetic and calculable from the instance of the terminal. Thus the availability vector for a terminal entry symbol x is trivally [1]CS(x). If the entry symbol of a state is a non-terminal A, its entry shapes are found from its occurrences as a left-hand side in completed states.

For the completed item I in each such state, the entry shape: raav(I/A,0) is used.

In the continuing example:

Since identifier is a terminal, all its attributes are synthetic. Let S(identifier) = {name,hash}. The entry shape of identifier is always (11), thus any transition pair with identifier is (identifier,11).

Assume that

I(<var>) = ∅

S(<var>) = {sym_tab,type}

I(<label>) = {address}

S(<label>) = {sym_tab}

and the following completed state exists:

```
                ___________________________________________
                |                                          |
--------------->| [<var>(|11) ::= identifier(11|).4]       |
(identifier,11) |                                          |
                | [<label>(0|1) ::= identifier(11|).5]     |
                |__________________________________________|
```

An entry shape for <var> is raav(I1/<var>,0) = (11) and for <label> is raav(I2/<label>,0) = (1).

Ghost State Items

The algorithm to compute daav's for ghost state items is the same as that for successor states. The Goto function of the GLC machine must be extended to map transitions correctly in ghost states as if the recognition symbols did not exist.

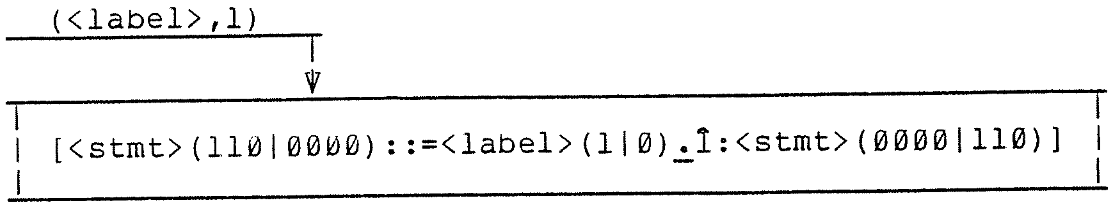
Recipient Attribute Availability Vectors

As previously stated, the raav(I) represents only potential availability for recipient attributes, unless the item I is known to accurately reflect the parse. An item is known to be applicable and termed a <u>recognized item</u> if the configuration marker ($\underline{\centerdot}$) is either to the right of the recognition symbol ($\hat{\imath}$) in the item or to the immediate left of the that symbol. The evaluation equation (1) is usefully applied to recognized items.

$$\overline{\text{raav}}(I) = \text{MDi} \times \overline{\text{daav}}(I) \qquad (1)$$

Each time the evaluation equation is applied to a recognized item, a fresh prediction shape is obtained. If $I = [A ::= X \; \hat{\imath} \; Y \; \underline{\centerdot} \; B \; Z]$, $B \in N$, then raav(I/B,j) is a predicted shape for the j-th occurrence of B in production i.

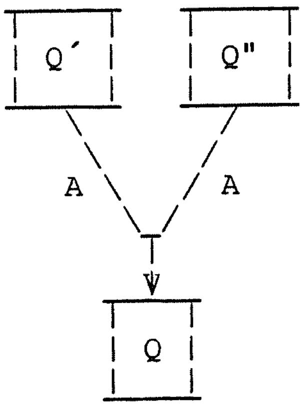
Assume, for example, that the first two inherited attributes of <stmt> are passed identically to its descendent <stmt> in production 1. Then a predicted shape for <stmt> is (110).



Chirica and Martin´s restrictions to attributed grammars are used only to simplify equation (1) and several definitions and theorems to be developed. In order to modify equation (1) for fully general attributed grammars, the set Don(i) must be expanded to include every attribute occurrence in the production. The dependency matrix MDi is likewise enlarged by the size increase in Don(i). An raav is calculated by taking the transitive closure of the enlarged MDi and using an equation similar to (1). Because of the extra complexity of this more general method, Chirica and Martin´s simple restrictions are retained.

Graph Construction

One problem in dealing with graph structures has been ignored in the availability-extended item set algorithms developed so far. As a part of a state graph, each configuration set may have several predecessors, though always over transitions with the same label. The algorithms described work well on the basis of a single predecessor, (treating the graph erroneously as a tree) but fail in the more general case. It may be that one state will be valid as a successor of several states; otherwise new states must be created as valid successors. In the parse graph situation sketched below



it might be the case that the propagate step and application of equation (2) do not yield an identical successor Q for each of Q´ and Q". To determine when multiple semantic copies of a syntactic state are necessary (different attribute availability vectors will apply in each case) the following information is needed: which predecessor to use in the calculations and whether a given state is a valid successor of both of two distinct states. Another method of solution, creating unique successors as soon as possible in the iterations, may never halt. This method is unable to recognize the fact that each of the successors created (from the same base state) may eventually be updated to become equivalent states when the solution converges.

There is a simple way to choose unique predecessors for graph construction. As the GLC(k) machine is created from state zero, it is has a tree structure until a successor state is found to be a duplicate of one already created. A back-loop is a transition to an previously created state. "Back-looping" occurs only in the case of recursion in the grammar (other than left recursion) and from newly created states. Deleting the back-loops leaves a tree (and unique predecessors). Each iteration in the solution then ignores the back-loops allowing the recursive application of the propagate step to eventually halt. When a series of iterations halts because no new shapes can be found, a separate check-up pass corrects any inconsistencies due to ignored back-loops, perhaps creating new states. This process is repeated until the check-up pass determines that no inconsistencies exist. The full process must halt because there are only a finite number of possible availability-extensions for each base syntactic state.

Algorithm 2A details the steps required in extending the GLC configuration sets with availability vectors that were previously outlined. The algorithm consists of a main procedure, Build_LC-Availability_Graph, and several auxillary procedures to extend a GLC(k) parse state graph.

Algorithm 2A: Total graph construction

Input: recognition rule grammar $G=(N \cup \hat{N},T,P,S)$ and the Goto function associated with the GLC(k) recognizer.

Output: LC-availability graph in form of availability extended GLC(k) states.

```
Initialization:
   let TP = { X $\in$ N | X occurs in a trailing part of some
        production}.
   Let PS(X), the predicted shapes for X, be initialized by
        PS(S) = {[1]CI(S)}
        PS(A) = {} for other A $\in$ N.
   Let ES(X), the entry shapes for X, be initialized by
        ES(a) = {[1]CS(a)} for a $\in$ T
        ES(A) = {} for A $\in$ N.

Build_LC-Availability_Graph:
  repeat
    repeat
      Prediction_Step;
      Recognition_Step
    until (no new members are added to any ES(X) or PS(X));
    Check_Back_Loop
  until (no new states are created in Check_Back_Loop);
end
```

```
Prediction_Step:
  for each X ∈ TP do
    for each v ∈ PS(X) do
      create state Q(X,v);
      if X is not left recursive
      then for each item in Q(X,v)
        if LHS of item = X
        then daav(item) := v.[0]CS(RHS of item)
        else daav(item) := [0](CI(X)+CS(RHS of item));
      for each exit symbol A from Q(X,v) do
        Propagate(Q(X,v),A);
end


Propagate(Q,A):
  if Goto(Q,A) is not a back-loop
  then for each v ∈ ES(A)
    create new state for Goto(Q,A) call it Q´
    for each basis item I´ in Q´ of form: [L::=X A.Y]
      find I ∈ Q such that scan(I) = I´;
      daav(I´):=([0](CI(L)+CS(X)) .v. [0]CS(Y)) or daav(I);
    for each exit B from Q´ do
      Propagate(Q´,B);
end
```

```
Recognition_Step:
  for each state Q
    for each recognized item I in Q
      compute raav(I);
      if A ∈ TP follows . in I
      then PS(A) := PS(A) union raav of predicted A
      else if I is completed with left-hand side B
        then ES(B) := ES(B) union raav of left-hand side B
end

Check_Back_Loop:
  for each back-loop in the current graph
    from Q over transition pair (A,v)
    create new state Goto(Q,A) call it Q´;
    for each basis item I´ in Q´ of form: [L::=X A.Y]
      find I ∈ Q such that scan(I) = I´;
      daav(I´):=([0](CI(L)+CS(X)) .v. [0]CS(Y)) or daav(I);
    if Q" (equivalent to Q´) exists
    then Goto(Q,(A,v)) := Q";
         eliminate Q´;
end
```

The algorithm presented is used to construct for any recognition rule grammar an availability-extended GLC configuration set known as an <u>LC-availability graph</u> (LCAG).

The attribute availability vectors within extended states can be used to locate attributes that have been evaluated and those that are ready to be evaluated in nodes recognized during a GLC parse.

Theorem 2.1

1) Extended versions of all GLC states are created in algorithm 2A.
2) All possible extensions fo GLC states are created by algorithm 2A.

Theorem 2.1 and the correctness of the algorithm for construction of the LC-availability graph can be demonstrated by outlining a proof with the following observations. The underlying GLC parser correctness is shown by Demers [Dem77]. The important observation is that states are created by the prediction and propagate steps in the same order they would occur in any particular parse. In 2A, however, they are all generated in parallel. An induction argument on the number of transitions taken in a GLC parse can be used to show that the states entered by the parse of a given string are availability-extended in at most the same number of iterations of Build_LC-availability_Graph. The same argument can be used to determine that the predicted shapes and expected shapes sets include the attribute availabilities that occur in any particular parse.

The availability graphs alone do not detail how a parse is to procede, how items are to be chosen, nor how and when attributes are to be evaluated. A stack machine is necessary to perform these tasks based on instructions generated from the information content of the graph.

Chapter 3

The Attributed Pushdown Processor

A processor is needed to provide a left-corner parse of an input string and carry out the attribute evaluations shown possible in the LC-availability graph. The pushdown processor (PP) described by Aho and Ullman [AU71a] generates syntax trees from strings and is guided by an LR parse. The pushdown processor contains a semantic stack with pointers to previously constructed subtrees, so the "semantic forest stack model" of chapter 1 can naturally be implemented through a modification of this processor.

A modified processor presented in this chapter, the attributed pushdown processor (APP), is designed to maintain the semantic forest stack and evaluate attributes as it performs a GLC parse. The look-ahead symbols and the top stack node trigger a sequence of actions that manipulate the forest. Recognized items in certain stack states may signal further actions that evaluate and transfer attributes and consider the disposition of subtrees in the forest. The action sequence plans are determined from the left-corner availability graph (LCAG) for the grammar. The GLC(k) parser underlying the LCAG actually guides the movement through the graph.

As an attributed derivation tree is being constructed, the paths of information flow in the tree are connected. In a one-pass translator, this flow directly follows the construction so that it is not indefinitely blocked by incomplete structure. Information flow in an attributed derivation tree occurs between two nodes if an attribute of one node has a donor in the other node. Due to the definition of attributed grammars, information flow is local to nodes that occur as the result of a single production instance in a derivation tree. The concept of information flow can be extended naturally to include the transitive closure of the above defintion, and information flow graphs (attribute dependency graphs) can be constructed in derivation trees [Knu68].

Because the retention of subtrees in the semantic forest and their subsequent evaluation introduces another level of complexity to parse-time evaluators, it is important to distinguish the class of attributed grammars in which subtrees are never required to be saved. After this class is identified in the next section of this chapter, the mechanisms that allow more general grammars to be evaluated will be elaborated.

LC-attributed Grammars

The S-attributed and L-attributed evaluators are capable of evaluating exactly those grammars in which dependency graph arcs closely follow the movement of the associated parser. These methods rely on (1) knowing the exact parse configuration, and (2) having the ability to evaluate all attributes whenever a reduction is recognized. The price that traditionally has been paid for these abilities has been either severe limitations on the allowable underlying cfg´s or sharp restrictions on acceptable attributed grammars. The APP represents a compromise made to gain power, and thus semantic expressibity, in exchange for delays in attribute evaluations due to the uncertainty of the configuration of some states of the parse. It encompasses the particular combinations of attributed context-free grammars mentioned above. This gain in power is made through the use of the configuration sets of the GLC(κ) parse machine.

The GLC parser of the APP recognizes syntax tree nodes in two fashions. While a parse employs bottom-up recognition, all synthetic attributes can be evaluated in the natural way. After a production is announced, both inherited and synthetic attributes can be used in the trailing part of the production. An LC-attributed grammar can be defined in a manner analogous to that of L-attributed and S-attributed grammars.

An attributed recognition rule grammar $G=(N \cup \hat{N},T,P,S)$ is <u>LC-attributed</u> iff for each production of the form:

$$L ::= X\ A\ Y\ \hat{\imath}\ Z\ B\ W$$

where $X,Y,Z,W \in (N \cup T)^*$, $L \in N$, and $A,B \in (N \cup T \cup e)$

1) A has no inherited attributes.
2) The donors of an inherited attribute of B are inherited attributes of L (if L is not left recursive) or arbitrary attributes of the symbols in X, A, Y, and Z.
3) Donors of synthetic attributes of L are inherited attributes of L and arbitrary attributes of X, A, Y, Z, B, or W.

If a language G is known to be GLC(k), then the restriction involving left recursion in L is unnecessary since the recognition symbol will never occur to the extreme left in a left recursive production. In this case, a left recursive symbol must exist in a left corner and cannot have inherited attributes.

In the discussion of classes of attributed grammars, a distinction is made between the types of context-free grammars and the restrictions on attribute evaluation rules. A classification involving both is termed an attributed grammar, context-free grammar pair. Several properties of LC-attributed grammars are contained in theorem 3.1 and its corollary.

Theorem 3.1

a) Every LC-attributed grammar is L-attributed and

b) every S-attributed grammar is LC-attributed.

Proof:

Both containment claims are straightforward. For (a) none of the restrictions 1, 2 or 3 violate the restrictions in the definition of L-attributed in chapter 1. For (b) LC-attributed requires no restrictions on the use of synthetic attributes. I

Corollary 3.2

No LC-attributed grammar is circular.

Proof:

All LC-attributed grammars are L-attributed and L-attributed grammars are non-circular [LRS74]. I

Translation Stack Nodes

The purpose of the attributed pushdown processor is to perform syntactic analysis and semantic anaysis at the same time. It does so by keeping a single compile-time stack that is a combined syntax and semantic stack. To avoid confusion and emphasize both functions, the stack is refered to as the translation stack. Nodes on the translation stack contain LCAG states and represent both the "state of the parse" and a grammar symbol in a sentential form or an entire production. The state of the parse is represented by the underlying configuration set, and the grammar symbol is the entry symbol or predicted symbol of that set. Translation stack nodes have attribute value vectors associated with them of length equal to the number of attributes of the grammar symbol or production they represent. As will be described later in this chapter, each node may also point to its subtrees (see figure 3.1).

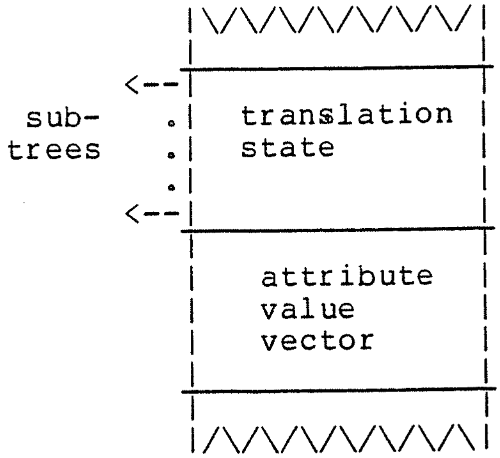


Figure 3.1 Translation Stack Node

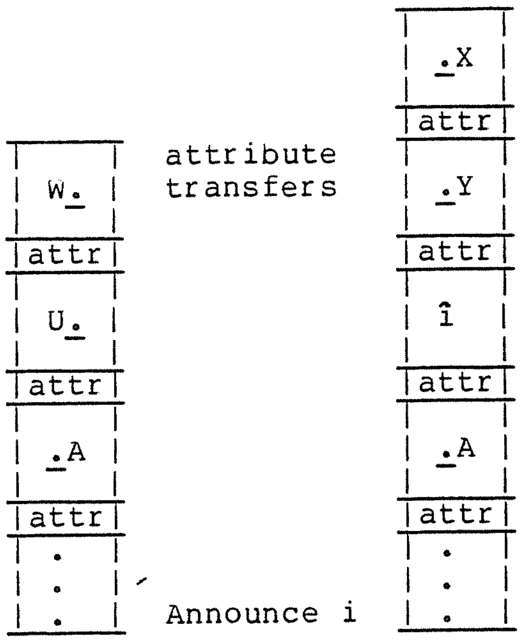
The APP is only restricted in that it uses some type of GLC(k) parse. Both the length of look-ahead used and the means of choosing the look-ahead functions are independent of the semantic aspects of the processor. The parser must only be able to choose which completed item in a configuration set correctly represents the right-hand side that is being formed when it signals production recognition.

Manipulation of the Translation Stack

The item sets of the GLC(k) machine either refer to a scan just completed or a prediction just made. The states can be used, in a natural way, as translation stack nodes that encompass both syntax and attributed semantics. A _scan item set_ is a state entered as a result of bottom-up recognition. A node on the translation stack marked with such a state contains a buffer for the synthetic attributes of the left-corner symbol recognized. A _predictive item set_, denoted Q(A) for a prediction of A, is a state stacked during a top-down prediction. The corresponding predicted node buffers the inherited attributes of symbol A. When a GLC(k) parser determines that predictive and scan item sets refer to the same instance of a given symbol, the inherited and synthetic attributes can finally be merged together. This decision is made in a GLC parse when a _pop_ action is indicated.

Predictive item sets are stacked as a result of an _announce_ action. A prediction occurs for each symbol in the trailing part of the production. In a translation, one other node will be stacked prior to the predictive nodes. The extra node represents, in effect, the left-hand side of the recognized production. Since the entire production is known, this node may be used to hold all attribute information known for that instance of the production in the derivation tree; it can be termed the <u>production node</u>. Before pushing the production node, a GLC parser pops the nodes that represent the left corner of the production. Like the typical bottom-up semantic stack manipulator, the synthetic attributes of those popped nodes are transferred to the production node that replaces them. When the left-hand side of the production refers to the same symbol instance as the most recent predicted symbol, inherited attributes can be copied directly from the predicted node to the left-hand side portion of the production node.

For example, with the production A ::= U W ↑ X Y, the following sequence illustrates the announce action. A predicted node is denoted by $\underline{.}X$ and a scanned node by $X\underline{.}$.



The node marked with $\hat{\imath}$ represents the production node for i.

With availability-extended GLC(k) item set states in APP translation stack nodes, the standard algorithm of Demers [Dem77] is modified. A state with a recognized item marks the top stack node at the time an announce action occurs. When a predictive node is popped, a transition in the LCAG occurs over the popped symbol and its raav from the production state into a ghost state. The ghost state updates the production translation stack node. At the time a production translation stack node reaches the top of the stack, it is popped and a transition is taken from the state of the node now on top of the stack over the left-hand side of the production (considering its available attributes). This treatment of the left-hand side of productions differs from Demers´ algorithm since, in parsing, the announce action immediately pushes the state that results from having seen the predicted symbol. The push is delayed here because at this early point, the recipient attributes available in the root cannot in general be known to characterize the new state. The replacement of the availability-extended state representing the production is required in order to keep track of the intermediate changes in the availabilities of the attributes of the trailing part and the root.

Both predictive and production nodes contain inherited attribute information associated with the predicted symbols. The predictive nodes need inherited information in order to pass it on to descendants. A production node requires a collection of all attributes since the predictive nodes will be popped when recognized.

Two problems are associated with the GLC translator model outlined above. It performs well when the underlying attributed grammar is LC-attributed, because all recipient attributes are available in symbols occurring to the left of the configuration symbol in recognized items. This availablity allows the direct and immediate computation of each attribute´s value as a symbol is recognized in the parse. It is desirable (and possible) though to allow more general attributed grammars.

One problem exists at the discovery of a new production instance in a derivation tree; it is not necessarily known now this production instance attaches to the last predicted non-terminal, although it must be an offspring (see below).

```
          A    last prediction
          .
          . ?
          .
          B
         /|\   discovered
        / | \   production i
       . . . .
```

It is the unknown structure between B, the left-hand side of production i, and its ancestor A that inhibits the transfer of inherited information available in A to B and B's offspring in grammars that are not LC-attributed.

The second problem is associated with the handling of forward references. Since the parse procedes from left to right, some retention of subtrees will be necessary to satisfy attributes that are not evaluable when their nodes would normally be popped. Maintaining attribute availablility vectors within the states of the evaluator and using the semantic forest structure enable the attributed pushdown processor to overcome these difficulties.

Scheduling Attribute Function Application

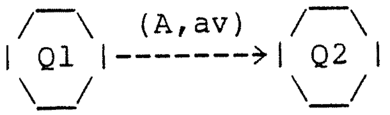
A GLC parse has an action function that maps states and input symbols into actions to be taken on the stack. The actions include *shift*, *announce* i and *pop*. The APP driven by the GLC parse extends the action function to include steps necessary for attribute evaluation. The announce action, as pointed out previously, can evaluate attributes marked as available in the recognized item of the LCAG state. The action consolidates synthetic attribute information from the left-corner symbols about to be popped into the production translation stack node. Inherited information available in the first predicted node is evaluated as that node is put on the stack.

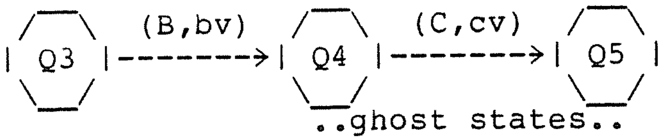
For example, the raav(I) for a recognized item

$$I = [L(110|00) ::= P(11|111)\ \underline{.}\hat{\imath}\ R(0|01)\ T(000|0)]$$

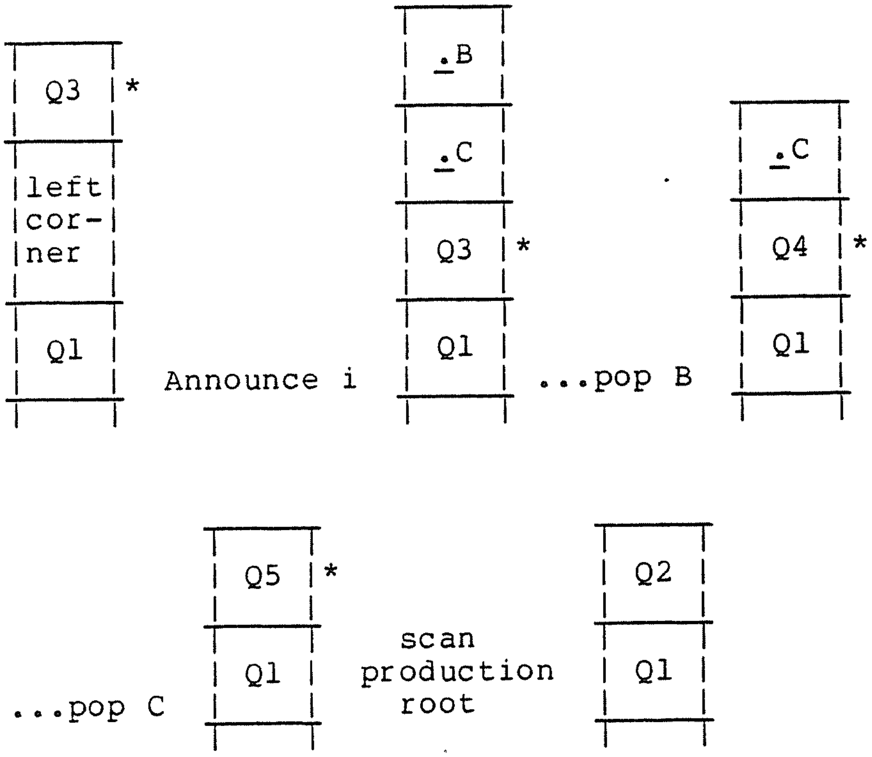
that characterizes a production node, shows that the second (but not the first) inherited attribute of R is ready to be evaluated at this point. With the pop action both inherited and synthetic attributes of the predicted and production stack nodes are merged and then transferred to the production node that placed the predicted node on the stack.

For example, assume B and C are the trailing part symbols of production i, and A is the left hand side. Q3 is the state that causes $\hat{\imath}$ to be announced; the signifigant portion of the graph is:





The sequence of changes occurring in the translation stack is pictured below. (The production node is marked with *.)



Scheduling Subtree Elimination or Retention

The APP machine must make the choice at each reduction to act either as an Aho and Ullman pushdown processor and construct the syntax tree or as a simple stack machine and pop the left corner nodes. In general a combination could occur. It is simpler to consider a machine that builds the syntax tree and then prunes those branches no longer needed. The decision regarding which branches are to be removed is made at machine generation time independent of any particular parse. A popped translation stack node is eliminated if all its attributes are evaluated. For example, from the raav(I) for the item

I = [L(110|00)::= A(11|111) B(100|01) $\underline{\cdot}\hat{\imath}$ C(00|1)],

it can be determined that the subtree of A is fully evaluated and no longer needed, but there are two synthetic attributes of B not yet evaluated. While A can be discarded, B's subtree must be saved. The attributes of a subtree will always have been evaluated by the processor if the LC-availability graph shows them to be available.

APP Evaluation

The machine for semantic attribute evaluation thus far described omits one phase of attribute evaluation. Those subtrees that were retained during a reduction require further evaluation to obtain all necessary synthetic attributes. It is difficult to fit this phase of attribute evaluation into typical parse-time evaluators. To avoid the problem an attempt can be made to find attribute grammar, context-free grammar pairs that never require subtrees to be retained. One such pair is (LC-attributed,GLC(k)), as will be shown. A more general solution utilizing retention will be developed in chapter 5.

Chapter 4

A Formal Model of the APP

This chapter details the specific inter-relationships between the LC-availability graph and the attributed pushdown processor. The APP is formalized as a machine, and it is shown how the machine is constructed from a GLC(k) state graph and the LC-availability graph that extends it. Properties of the APP will be established through theorems based on availability notions and GLC parsing. The formal evaluation model developed in this chapter does not include the retention of subtrees and their eventual evaluation. This version, the <u>simplified</u> APP, is capable of producing translations of all LC-attributed GLC(k) grammars (which includes all S-attributed LR(k) and L-attributed LL(k) grammars).

The attributed pushdown processor is a machine that acts in a manner very similar to a generalized left-corner parser but operates with a stack of translation nodes as described in chapter 3. One major difference is that the Goto (or next state) function operates on transition pairs and thus considers the state of the evaluation of attributes. A second important distinction in the operation of the APP is that it occasionally reaches below the top stack node to update the state of a previously stacked node. The depth of updating is bounded by the length of the longest trailing part in the set of productions. The stack qualities of the APP are not really violated since the machine can provide temporary storage for the nodes above the one being updated, and replace them after the update.

Translation and Notation

Formally, each node on the translation stack is a 4-tuple. The four components of a translation stack node (abbreviated TSN) for an attributed recognition rule grammar G = (N u $\hat{N}$,T,P,S) are:

(state) an availability-extended state of the LCAG(G),

(symb) a symbol from N u $\hat{N}$ u T,

(attr) a vector of attribute values from the union of all attribute domains,

(link) an integer denoting the distance down the stack from a node to its parent production node.

A *configuration* of an APP is a pair (Tstack,Input) where Tstack is a stack of translation stack nodes and Input is an attributed string from T*. Tstack represents the combined syntactic and semantic translation stack, and Input holds the unscanned portion of the sentence to be translated. For the APP algorithms, the following notation and functions are useful.

let POP(Node) = Node, and assume it has the side effect of removing Node from the translation stack. POP is only applied to the top of the stack.

let PUSH(Node) have the side effect of placing Node on top of the translation stack.

let $k{:}x$ = the first k symbols in $x \in T^*$ if x contains at least k symbols, otherwise x.

let TSN[n] represent the n-th node from the bottom of the translation stack, starting at zero.

let TSN[n].<f> represent the <f>-field of TSN[n].

let V1&V2 be the concatenated value vector resulting from V1 and V2.

let Ø denote an undefined attribute value.

let [Ø]n represent n occurrences of Ø.

let V1 <u>merge</u> V2 be the component-wise merging of value vectors V1 and V2. Both V1 and V2 must be the same length. If two components have identical values (or are both undefined), the merged result is that value (or undefined). If exactly one component is undefined, the result is the defined component. Otherwise, an error results.

let Q(A,v) = the availability-extended version of predicted state Q(A) with predicted shape v.

let PREaav(I) = raav(I/B,j) where the j-th occurrence of B in I directly follows the configuration symbol or the configuration, recognition symbol pair. It is not defined if I is completed.

let LHSaav(I) = raav(I/L,0) where L is the left-hand side of the production of I.

let Goto(Q,(s,v)) be the availability-extended next state function for the LCAG. The argument Q is a state and (s,v) is a transition pair.

let C(X) = CI(X) + CS(X), the total number of attributes in string X.

let SI be the initial value vector of the inherited attributes of the goal symbol S.

The APP translator starts in a configuration (Z,w) where w is the sentence to be translated and

$$Z = (Q', \hat{0}, [\emptyset]C(S), 0) \quad (Q(S), S, SI \& [\emptyset]CS(S), 1).$$

In the two nodes of Z (top appearing to the right) Q' is the initial state of the GLC(k) machine which automatically announces the augmented production zero: <sentence>::=$\hat{0}$ S. Thus its symbol field is $\hat{0}$ and it has an empty value vector. The symbol field in $\hat{N}$ identifies this node as a production node. A production node assumes the special role of maintaining attribute value vectors for the entire production similar in manner to its counterpart in Lewis, Rosenkrantz, and Stern´s L-attributed evaluator [LRS74]. As an automatic result of the action "announce 0," the predicted state, Q(S) = closure({[::= $\underline{.}$ S]}), is entered. Its corresponding translation stack node contains the state Q(S), the symbol S, an attribute value vector SI&[0]CS(S) and a link of 1 referencing the production node just below it.

The processor then procedes to change configurations according to algorithm 4A below until an error is encountered or the terminal configuration (Z´,e) is reached. Z´ will contain only the bottom node of Z with an attribute value vector relecting the completely parsed goal symbol. Due to the basic similarity of the APP with a GLC parser, the following algorithm is a generalization of algorithm 3.4 of Demers [Dem77].

Algorithm 4A: APP Translation

1) Let the initial configuration be (Z,w) as defined above.
2) (repeat this step until it halts indicating error or the terminal configuration (Z´,e) is entered). Let (TSN[0]...TSN[m],x) be the current configuration and u = k:x.

Perform whichever of the following steps applies to the configuration.

a) Action(TSN[m].state,u) = shift. Perform the extended shift step, algorithm 4B.

b) Action(TSN[m].state,u) = announce i. Perform the extended announce i step, algorithm 4C.

c) Action(TSN[m].state,u) = pop. Perform the extended pop step, algorithm 4D.

d) Action(TSN[m].state,u) = error. Halt.

I

The extended pop, shift, and announce i steps are explained in detail after an overview of the actions they perform. Each time part of the step relies on a function based on the LCAG(G), the computation of that function is explained.

The Action function of algorithm 4A is that of the left-corner parser for G: the triple (States,Action,Goto). Thus the driver of the APP is the underlying GLC parser, and attribute transmission is accomplished during APP actions.

Synthetic attribute evaluation takes place in two instances. When a shift consumes a token, its attributes are accessed in lexical analysis and incorporated into the translation stack. When a production node reaches the top of the translation stack and is to be popped, all synthetic attributes of the root are evaluated.

Inherited attribute evaluation occurs when a predicted non-terminal reaches the top of the translation stack. A function EvalInh is applied to it to compute inherited information from its root and left context.

Attribute values vector are transferred to the production node in several cases. Synthetic information from the left corner is inserted when an announce action occurs. Because a pop signals subtree recognition, synthetic and inherited information of a predicted symbol is entered into the vector of its production node as it is popped. when a production node is placed upon the node that predicted its left-hand side, a function GetInh retrieves its inherited attributes.

An example LC-attributed grammar is presented in figure 4.1, and its LCAG appears in figure 4.2. Illustrations will depict the translation of a sentence of the language as each action is detailed.

```
EG = (N,T,P,A)

N = {A,B}   T = {x,y,z}

S(A) = S(B) = {typ,val}

S(x) = S(y) = S(z) = {val}

I(B) = {op}

P includes:


A ::= x y 1 B z

    B.op := x.val + y.val;

    A.val := B.val + y.val + z.val;

    A.typ := B.typ;

A ::= x y 2 z B

    B.op := x.val * y.val;

    A.val := B.val + z.val;

    A.typ := B.typ;

B ::= x 3 y

    B.val := if B.op > 0 then x.val + y.val

                         else x.val - y.val;

    B.typ := 1;

B ::= x 4 z

    B.val := (sign(B.op) * x.val) / z.val;

    b.typ := 2;
```

Figure 4.1 LC-attributed Grammar : EG

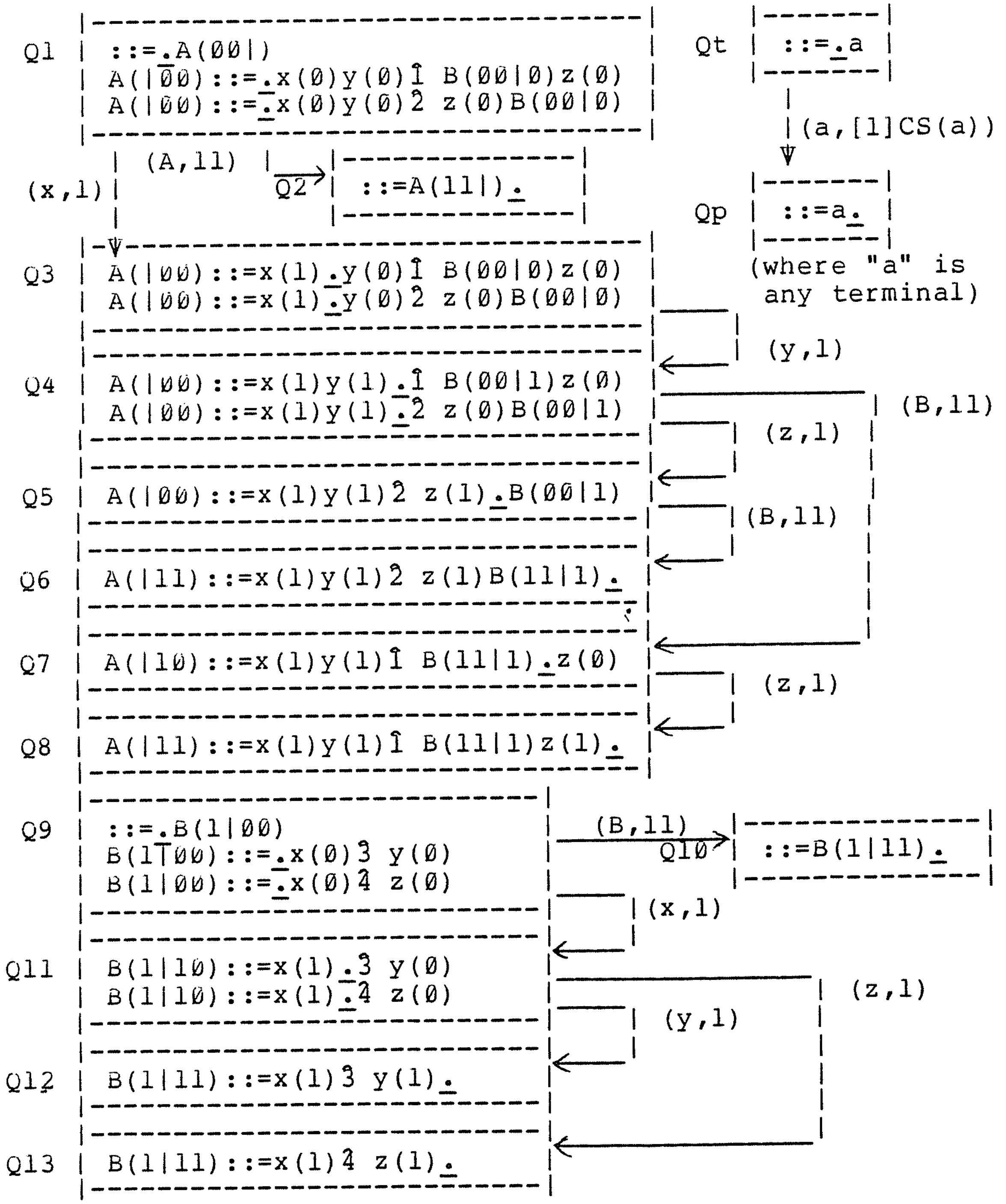


Figure 4.2 LCAG(EG)

The Extended Shift Action

The shift action is used to remove a terminal from the input string and then enter a new state. This action alone affects the Input section of configurations of a translation.

Algorithm 4B: Extended Shift

```
Assume (TSN[0]...TSN[m],a Input´) is the current con-
figuration and Vt is the synthetic attribute vector
determined for "a" in lexical analysis
1) Input := Input´
2) Next := Goto(TSN[m].state,(a,[1]CS(a)))
   PUSH((Next,a,Vt,0))   I
```

The following sentence will be translated using the attributed grammar EG in the ensuing examples:

x(3)y(2)x(4)y(2)z(5)

The parenthesized number following each terminal represents its attribute value. The translation stack is illustrated with its bottom to the left. Attribute values (within brackets) are partitioned into symbol portions by ":" and within symbols by "."; a hyphen "-" identifies missing values. Within symbols, inherited attribute values procede synthetic attribute values.

The initial shifts of the translation are presented below in sequences of configurations.

Tstack	Input
(Q1,A[-.-],1)	x(3)y(2)x(4)y(2)z(5)
* shift x(3) *	
(Q1,A[-.-],1)(Q3,x[3],0)	y(2)x(4)y(2)z(5)
* shift y(2) *	
(Q1,A[-.-],1)(Q3,x[3],0)(Q4,y[2],0)	x(4)y(2)z(5)

The Extended Announce Action

When Action(TSN[m].state,k:x) = announce i, enough information exists in the parse configuration to signal the recognition of a production instance. At this point, the parsing stategy turns from bottom-up to top-down. A production node is created to represent the recognized item and serve as a repository for its evaluated attributes. The effect of announce i is to remove the left corner of production i from the stack, absorb its attributes, and predict the production´s trailing part. Because the predicted shape of trailing part symbols is not known until they reach the top of the stack, the state field is not set until that time. The production node state will not reflect the recognition of the left-hand side until it reaches the top of the stack as a result of a pop step. It is necessary to keep ghost state information at the production node, so that available attribute information correctly reflects the parsed offspring.

```
Algorithm 4C:  Extended Announce i
   for production i = L ::= R1 ... Rj î Rj+1 ... Rn
   Assume current configuration is (TSN[0]...TSN[m],x).
   "Top" is always the index of the top node on the
   translation stack.
   let I be the item from TSN[m].state that caused  i  to
   be announced.
     1) Buffer := e (the empty vector)
        create node TEMP := (TSN[m].state,î,Buffer,0)
     2) (remove left corner, gather attributes)
        for k from 1 to j do
        begin LCS := POP(TSN[Top])
              TEMP.attr := LCS.attr & TEMP.attr
        end
     3) (stack the production node,
         copy inherited information)
        PUSH(TEMP)
        GetInh(TSN[Top])
     4) (push all but first node of trailing part)
        for k from n down to (j+2) do
              PUSH((Nil,Rk,[0]C(Rk),n+1-k))
              (Nil is used to represent an
```

```
              undefined state part)
     5) If n>j then
           (push first predicted node, if one exists)
           PUSH((Q(Rj+1,PREaav(I)),[0]C(Rj+1),n-j))
           EvalInh(TSM[Top])
      I
```

The procedure GetInh copies inherited attributes of the left-hand side of a production node from the predicted node that exists below a production node when they are known to reference the same symbol instance. In terms of attribute occurrences, GetInh(Node) is the following set:

{ L(0).a | a is the n-th attribute in I(L),
daav(Item/L,0)(n) = 1, and
L is the left-hand side of Item
which is recognized by Node }

when a true vector element of daav(I/L,0) exists for the root of a production node, the predictive node and production node root refer to the same instance of the symbol. This equivalence is determined in the prediction step of algorithm 2A.

EvalInh evaluates the inherited attributes of a predicted symbol according to the state of the production node that predicts it. The attribute occurrences selected for evaluation by EvalInh(Node) are:

{ B(j).a | a is the n-th attribute in I(B),

raav(Item/B,j)(n) = 1, and
the j-th occurrence of B follows . in
Item which is recognized by Node }

The attribute function rules selected by EvalInh are applied to the production node value vector, and their results are merged into the value vector of the predicted node.

The translation example is continued to illustrate the announce step. Qt is the state used to predict any terminal and Qp is the state entered upon a shift of any terminal.

```
(Q1,A[-.-],1)(Q3,x[3],0)(Q4,y[2],0)                 x(4)y(2)z(5)

* announce 1 *

(Q1,A[-.-],1)(Q4,1̂[-.-:3:2:5.-.-:-],0)(nil,z[-],1)
   (Q9,B[5.-.-],2)                                  x(4)y(2)z(5)

* shift x(4) *

(Q1,A[-.-],1)(Q4,1̂[-.-:3:2:5.-.-:-],0)(nil,z[-],1)
   (Q9,B[5.-.-],2)(Q11,x[4],0)                          y(2)z(5)

* announce 3 *

(Q1,A[-.-],1)(Q4,1̂[-.-:3:2:5.-.-:-],0)(nil,z[-],1)
   (Q9,B[5.-.-],2)(Q11,3̂[5.1.-:4:-],0)(Qt,y[-],1)    y(2)z(5)
```

The Extended Pop Action

The underlying GLC parser signals a pop action when a predicted symbol is finally completely parsed. The two top nodes are actually removed from the translation stack. The first popped node represents the parsed instance of a trailing part symbol and it synthetic attributes. The second node represents the prediction of the same trailing part symbol and its inherited attributes. Since these nodes are about to be discarded and their attributes may yet be used as donors, their attribute value vectors are merged into the production node that made the trailing part prediction. The link field of the predicted node references the production node and is used to locate it. After merging the attribute value vectors, the production node state is altered to identify any new attributes received from the popped nodes. Which new attributes will be available cannot be predetermined in general, making this parse-time update necessary. If a production node becomes the new top stack node, then it is altered to reflect the completed parsing of the production instance.

Algorithm 4D: Extended Pop

```
  Assume current configuration is (TSN[0]...TSN[m],x).
  "Top" is always the index of the top translation stack
  node.
1) (remove parsed instance of symbol)
   Sym := TSN[Top].symb
   TopSt := TSN[Top].state
   TSN[Top-1].attr := TSN[Top-1].attr merge TSN[Top].attr
   POP(TSN[Top])
```

```
2) (update production node attribute value vector)
   Link := TSN[Top].link
   let PROD = TSN[Top-Link]
   (assume production of PROD is L::=R1...Rj î Rj+1...Rn)
   k := Link+j
   Av := [0]C(L.R1...Rk) & TSN[Top].attr & [0]C(Rk+1...Rn)
   PROD.attr := PROD.attr merge Av
   POP(TSN[Top])
3) let TItem be the completed item of TopSt for Sym.
   (update production node state)
   PROD.state := Goto(PROD.state,(Sym,LHSaav(TItem)))
   let PItem be the item in ghost state PROD.state
   (set state of predicted node
    or replace production node)
   If PItem is not a completed item
     then TSN[Top].state := Q(TSN[Top].symb,PREaav(PItem))
        EvalInh(TSN[Top])
     else PROD.attr := EvalSyn(PROD)
        PROD.state:=Goto(TSN[Top-1].state,(L,LHSaav(PItem)))
        PROD.symb := L
I
```

One new function is introduced in the extended pop algorithm 4D. EvalSyn evaluates synthetic attributes of the left-hand side of a production node. EvalSyn(Node) evaluates the following set of attribute occurrences:

{ L(0).a | a is the n-th attribute in S(L),
raav(Item/L,0)(n) = 1, and
L is the left-hand side of Item
which is recognized by Node }

Its resultant value is the updated attribute value vector for the left-hand side.

The translation is completed to demonstrate the effects of the extended pop step.

```
* shift y(2) *
(Q1,A[-.-],1)(Q4,1̂[-.-:3:2:5.-.-:-],0)(nil,z[-],1)
   (Q9,B[5.-.-],2)(Q11,3̂[5.1.-:4:-],0)(Qt,y[-],1)(Qp,y[2],0)
* pop y *
(Q1,A[-.-],1)(Q4,1̂[-.-:3:2:5.-.-:-],0)(nil,z[-],1)
   (Q9,B[5.-.-],2)(Q12,3̂[5.1.6:4:2],0)
* which immediately becomes *
(Q1,A[-.-],1)(Q4,1̂[-.-:3:2:5.-.-:-],0)(nil,z[-],1)
   (Q9,B[5.-.-],2)(Q10,B[5.1.6],0)
* pop B *
(Q1,A[-.-],1)(Q7,1̂[1.-:3:2:5.1.6:-],0)(Qt,z[-],1)
* shift z(5) *
(Q1,A[-.-],1)(Q7,1̂[1.-:3:2:5.1.6:-],0)(Qt,z[-],1)(Qp,z[5],0)
* pop z *
(Q1,A[-.-],1)(Q8,1̂[1.13:3:2:5.1.6:5],0)
* which immediatlely becomes *
(Q1,A[-.-],1)(Q2,A[1.13],0)
* pop A *
Halt state is reached with Z′ = (Q′,A[1.13],0)
```

Simple APP Properties

Lemma 4.1 is instrumental in establishing the result that the APP is capable of performing translations of any LC-attributed grammar without subtree retention. It shows that in any given stacked translation state, all necessary donor attributes will be marked available.

Lemma 4.1

Any LC-attributed recognition rule grammar $G = (N \cup \hat{N}, T, P, S)$ has a LCAG(G) such that for each item I in a state Q that marks a node on the translation stack during a translation, if

$$I = [\ L ::= X \ \underline{.}\ Y\ ]$$

then: (a) $daav(I/A,j) = [1]CS(A)$ if the j-th occurrence of A is in X, (i.e., all synthetic attributes of A have been evaluated),

and: (b) $daav(I/L,0) = [1]CI(L)$. (i.e., all inherited attributes of L have been evaluated)

Proof:

The proof of lemma 4.1 requires a look at translation stack configurations and how they change. An induction argument will show that both (a) and (b) are true for the initial configuration (Z,w) and continue to be true for any number of actions that are applied in the course of a translation.

<u>Base</u> <u>step</u>: number of actions = 0.

The configuration is (Z,w). Consider state Q(S). The items in the state are of the following two forms:

(i) $[\ S ::= \underline{.}\ Y\ ]$

(ii) $[\ B ::= \underline{.}\ Z\ ]$ with $B \neq S$

because no symbols have yet been scanned. Condition (a) clearly holds in both (i) and (ii) since X is empty. In the case of (i), $daav(I/S,0) = [1]CI(S)$

because all inherited attributes of S, the goal symbol, are constant and must be available by definition. Since the item (ii) exists in the closure of {[::=_._S]}, its left-hand side B must occur in a left corner because predictions are not made beyond a recognition symbol in a GLC state. Thus B cannot have inherited attributes and (b) is true vacuously. A similar analysis shows that in general any state Q(A,[1]CI(A)) satisfies both (a) and (b).

<u>Induction</u> <u>step</u>: number of actions = n, assume (a) and (b) hold for number of actions < n.

<u>case</u>: n-th action is a shift.

A shift action places one more state on the translation stack. A transition is taken over the transition pair (a,[1]CS(a)) according to algorithm 4B, step 2. The lemma holds for each item in the state on top of the stack.

Let any basis item in the state entered be illustrated by:

$$I = [\ B ::= X\ a\ \underline{.}\ Y\ ]$$

By equation (2) of chapter 2, daav(I/a,j) = [1]CS(a) where the j-th a is the a of the transition pair, since v = [1]CS(a). For any symbol A in X, daav(I/A,k) = daav(I´/A,k) for scan(I´) = I, again due to equation (2). I´ is from a state already on the stack so daav(I´/A,k) = [1]CS(A) by the induction hypothesis.

For items outside of the basis, (a) is true since the X parts are empty and (b) is true since the left-hand sides have no inherited attributes (because they are in left corners). Thus in general, any stacked state with an entry symbol shape of [1]n satisfies both (a) and (b).

case: n-th action is announce i.

The announce action removes states from the translation stack and pushes the production node and prediction nodes for each trailing part symbol. The lemma holds already for the state of the production node since it is a state that was already on the stack. For every predictive node other than the top, the state field is still undefined, and may be ignored. In the top predicted node, the predicted shape is raav(I/B,k) where the k-th occurrence of B is Rj+1 for the announced item in TopSt. The raav(I) is found by equation (1) of chapter 2. If daav(I/C,n) = [1]CS(C) for all symbols C(n) of the production in which attribute donors of Rj+1 can reside and daav(I/L,0) = [1]CI(L) for the root L, then raav(I/B,k) = [1]CI(B). Since the grammar is LC-attributed, all attribute donors of Rj+1´s attributes must be in the root or in symbols appearing left of the configuration symbol. The donor availability vectors for these symbols must be [1]n by the hypothesis since the item occurs in TopSt. The fact that the grammar is LC-attributed precludes any donors beyond what has been parsed and shown available, so the predicted shape must be [1]CI(Rj+1). As shown, Q(Rj+1,[1]CI(Rj+1)) satisfies the lemma.

case: pop is the n-th action.

The pop action deletes the top two nodes from the translation stack and changes the states of two other nodes. The new top stack node is either a predictive node (TSN[m-2] in step 2 of algorithm 4D) or the completed production node (PROD in algorithm 4D). The states of the new top stack node and scanned production node are updated. For the production node, the transition pair (Sym, LHSaav(TItem)) leads to the next state. The previous state had all donors to the left of Sym available. For the scanned symbol, Sym, the entry shape is taken from the left-hand side symbol of the popped state. The popped state represented a fully parsed production and by hypothesis all donors were available in that production instance. The entry symbol shape LHSaav(TItem) = [1]CS(Sym), so the state satisfies the lemma. For the updated predictive node (if one remains), PREaav(PItem) again must be all 1's (or totally available) as the first predicted node was (by the same argument as in the announce case), so this node´s state satisfies the lemma. If no predictive node remains, then the production node on the stack is replaced. Its new state must satisfy (a) and (b) in a manner similar to the scan case since the LHSaav must be all ones, and the state below it already satifies (a) and (b).

The induction is complete. I

Lemma 4.1 will be used to show that whenever EvalInh or EvalSyn are used in an LC-attributed evaluation, the recipient availability vectors referenced are all 1´s, and thus all attributes in their scopes are evaluable.

Theorem 4.2

Algorithm 4A for the APP constructed for an LC-attributed recognition rule grammar G from LCAG(G) successfully evaluates all attributes of each symbol before it is removed from the translation stack.

Proof:

In order to establish theorem 4.2, two facts must be demonstrated.

1) The daav equations are accurately reflected by the APP actions.

2) The evaluation functions evaluate all attributes of the grammar before they are popped from the stack.

Two equations set bits in the donor attribute availability vectors. The propagate equation ((2) in chapter 2) sets bits in the entry symbol of an item and copies set bits from a previous item. As a node is stacked in the translation (a transition is taken in the graph), evaluated attributes are retained in their respective nodes and the copying is justified. If the entry symbol is a terminal, then the lexical analyzer evaluates all its (synthetic) attributes verifying the use of the shape [1]n. If the symbol is a non-terminal, then EvalSyn evaluates the synthetic attributes marked available in the raav associated with the symbol just prior to the transition. The raav is computed to be all 1′s since all necessary donors are guaranteed available by lemma 4.1.

In the prediction equation, bits are set in the daav of the predicted symbol when it occurs as a left hand side and is not left recursive (directly or indirectly). Such attributes are copied to the production node when announced by GetInh.

The predicted shape is accurately reflected by the APP since EvalInh is applied to each predicted node as it reaches the top of the translation stack. In an

LC-attributed grammar, all the inherited attributes of a predicted symbol B can be evaluated since the raav associated with B shows them all to be available. The attributes are available because the dependency matrix in equation (1) of chapter 2 can reflect donors only in the root and symbols occuring to the left of the predicted symbol. The donor attribute availability vectors were shown to be fully available again by lemma 4.1.

EvalInh is applied to every trailing part symbol as it is stacked providing for the evaluation of all inherited attributes. EvalSyn is applied to every production node when it reaches the top of the translation stack resulting in the evaluation of every synthetic attribute. I

Theorem 4.2 demonstrates an important property of the attributed pushdown processor and LC-attributed grammars. It shows that the LCAG is not needed in its full generality to provide stack states for the APP when G is known to be LC-attributed. Much simpler state computations could be made. There is a simple linear (in size of grammar) algorithm to check for the LC-attributed quality, and it has been shown that the attributes are always ready to be evaluated. The broad usefulness of the LCAG is that it can be used with any LR(k) grammar, any left corners for that grammar, and any attribute sets and functions defined on the grammar. (Many important properties do rely on minimal left corners though.) The extent of this usefulness will be shown in the next chapters.

Chapter 5

Subtree Retention and Delayed Evaluation

For grammars more general than LC-attributed, the LCAG item-wise attribute availabilities still identify donors known to be evaluated and recipients known to be evaluable. If the APP algorithm is used as presented with a more general attributed grammar, EvalSyn and EvalInh can evaluate just those attributes that the associated raav´s show to be available. Those attributes not available when their stack node is popped cannot be evaluated in this simple scheme. The LCAG thus isolates those parts of an arbitrary grammar that are not LC-attributed.

Several techniques are available to handle more general grammars. The problem can be returned to the language designer to rewrite the attribute specifications (perhaps using Knuth´s algorithm to alter inherited attributes into synthetic ones [Knu68a]) so that the grammar is LC-attributed. According to Knuth, this solution is quite awkward; the resulting grammar often tends to be very unreadable, much more complicated, and less convincing. The aim of this research is to make the language designer and implementor's work simpler, but this first solution complicates the task.

A second solution involves layers of attributed grammars, each of which is LC-attributed. The evaluation techniques of the APP can be coupled with the attributed tree transformations of Schulz [Sch76]. Schulz details a theory of n-pass compilations with attributed grammars utilizing tree transformations that is based on Jazayeri's alternating semantic evaluator [JW75]. Because Schulz leaves the particular method of parsing open to choice, the APP evaluator could be used to allow more flexible attributed context-free grammars. Finding the appropriate LC-attributed layers to accomplish a desired translation is a suitable area for further research.

Another possibility is to retain subtrees and use attributed tree evaluators for delayed attribute evaluation during the APP pass. Several advantages are apparent. Subtrees are retained only when necessary and only as long as necessary. Tree evaluators are constructed only for the (presumably smaller) languages of retained subtree root symbols. Attributes of some symbols that could not be evaluated until a second left-to-right pass in other evaluators (pass three in the alternating evaluator) because of earlier missed attributes are evaluable on pass one with retention and delayed evaluation. The fact that multiple passes over the complete syntax tree are unnecessary makes this solution attractive. Depending upon the complexity of the forward references in the language, relatively smaller portions of the derivation tree will in general have to be committed to storage.

The simplified APP presented in chapter 4 may be generalized to deal with translation stack nodes that would normally be discarded during a parse but are not fully evaluated semantically. The semantic forest stack model alluded to in chapter 1 can be implemented by suitable modifications to the extended actions of the processor. The retention of unevaluated subtrees is introduced because it allows an elegant formalization of unresolved semantic translation decisions without resorting to multiple full syntax tree passes. When attributes are evaluated in a retained subtree during the single pass of the APP, that subtree can be released from the semantic forest and its attributes used in the remainder of the translation. In this fashion, the APP is able to overcome the difficulties involved in forward references, left recursion, and other prediction problems.

Plans and Visits

Tree-walk evaluation visits within the semantic forest are applied to fully parsed subtrees to finish evaluation of attributes in these subtrees. The subtrees are pruned as much as possible during evaluation visits. With reasonable language specifications, the subtree storage allocation requirements can be kept at manageable levels. The storage allocation strategy, it appears, is best handled by standard heap management techniques [Knu68b]. For this reason, translation nodes that are part of a retained subtree are said to exist on the <u>translation</u> <u>heap</u>, rather than on the translation stack (although they are found through the translation stack).

There are three new functions that must be added to the generalized version of the APP. The APP must decide (1) when to prune constructed subtrees, (2) when to make a visit to a retained subtree, and (3) when attributes at the visited nodes are ready to be evaluated. The difference in operation between the generalized APP and the simplified APP is an enhanced evaluation mechanism that replaces EvalInh and EvalSyn. The evaluation mechanism is an ordered instruction sequence called a <u>plan</u> which includes the following instruction types:

1) Fp,n -- apply the semantic attribute evaluation rule for production p, recipient attribute n.
2) Visit(k,VS) -- visit either the k-th offspring if k>0 else the parent of the currently visited node with attributes marked in the availability vector VS.

3) Prune(k) -- release the k-th subtree of the visited node.

4) Update(Q) -- replace the state currently marking the heap state by Q.

A visit instruction uses a function PLAN that maps the states and availablity vectors into plans to select an instruction sequence. PLAN can either perform a table lookup to find the sequence, or it can compute the sequence (see algorithm 5D). A visit instruction is initially issued whenever a node is popped from the stack with the POP function in the extended announce (algorithm 4C) or pop (algorithm 4D) steps.

Algorithm 5A: Generalized POP(Node) function

let POP(Node) have the following side effects:

1) remove Node (must be the top) from the translation stack.

2) if Node is associated with a production node N´ that it has replaced as a result of a production node update (after reaching the top of the stack in algorithm 4D, step 3), then let either the node PROD (in the announce step) or node TEMP (in the pop step) reference N´ as an offspring on the heap.

3) schedule the instruction: visit(0,LHSaav(Item)) for execution at the end of the current extended action for Item that is recognized by the state of Node. I

Visits are made to a parent node when one of its offspring is popped. Visits to offspring are made when a plan is executed as a result of a visit to a parent node. The visit algorithm 5B is an extension of Kennedy and Warren's algorithm [KW76].

Algorithm 5B: Visit(k,VS)

k is an integer from zero to the length of the right-hand side of the currently visited production node.

$VS \in (0,1)^*$

the current production node is $(Q,\hat{i},attr,0)$

the visited node is $(Qk,\hat{j},attr\text{-}k,0)$

1) Merge attributes of the symbol shared between productions i and j from attr to attr-k.
2) Execute PLAN(Qk,VS) on the relative.
3) Merge attributes of the shared symbol from attr-k to attr.

Plans carry out the EvalSyn evaluations when right-hand side symbols of a production are popped. The EvalInh evaluations result from a visit to an offspring node that is on top of the stack after an announce step or the pop of a sibling.

States in the Extended Processor

The stack and heap nodes for the generalized APP require availability vectors for items to identify evaluated and evaluable attributes. The synthetic yield, information that identifies the effect of subtree visits, also characterizes production nodes. The synthetic yield is a function that maps sets of inherited attributes of a particular symbol into sets of synthetic attributes of that symbol. The resultant synthetic attributes are those that will become available for evaluation if the symbol's subtree is visited with the input set of evaluated inherited attributes. The synthetic yield can be used to determine if a visit to a subtree is worthwhile or to identify new attributes that will be evaluated (and perhaps cause other attributes within the production instance to become available). Yield information is kept in production nodes for every parsed non-terminal of the production. LCAG states and transitions are to be augmented to maintain this information; details are described later. A stacked production node containing the necessary information is diagrammed as follows:

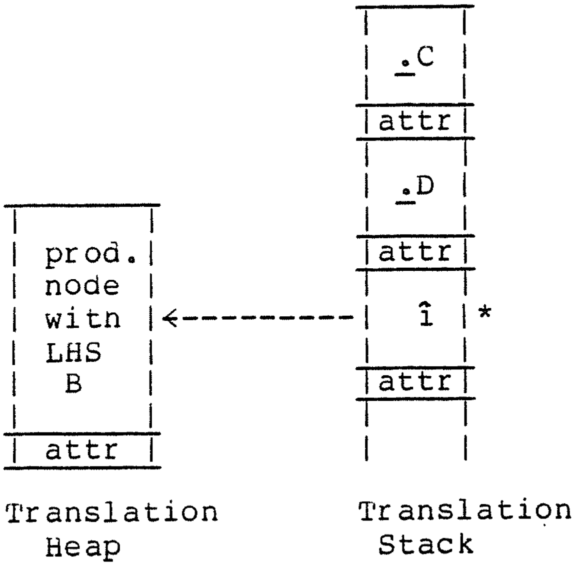
```
                          |                  |
                          |==================|
          <---------------|     î, link      |
                          |------------------|
                        . |    LCAG  state   |
          offspring     . |------------------|
                        . |    offspring     |
                          |      yield       |
          <---------------|   information    |
                          |------------------|
                          | attribute values |
                          |==================|
                          |                  |
```

<u>Subtree Retention</u>

Subtrees are automatically appended to production nodes by the generalized APP when nodes are normally popped in a GLC parse. Left corner nodes are popped during an announce action. Trailing part nodes are popped when a predicted symbol is totally parsed. Subtrees are constructed by linking the nodes on the heap to their parents and offspring. The decision to release an appended node with a Prune instruction is made when no missing attributes are discovered in availability vectors of the recognized item. A missing attribute is identified by a zero in the attribute availability vector. For example, if an offspring B is missing attributes as marked in the item

[A(01|100) ::= B(101|10) $\underline{.}$ î C(00|110) D(0|00)],

then B must be retained. In this case, B's second donor is unavailable. When attributes cannot be evaluated due to missing donors, the local subtrees with missing donors cannot be released. If an offspring is not fully evaluated, it exists as a production node with the attribute values of the production of which it is the left-hand side. A heap node linked to the stack is diagrammed below.



As the parse procedes in an APP translation, more information is accumulated about the syntactic structure of the sentence. As a result, more attribute flow paths are discovered. Eventually all flow paths are completed in any attributed grammar. If the grammar is non-circular, visits can be made following those flow paths to evaluate all of a tree's attributes. If enough information is maintained about retained subtrees, these paths can be followed correctly as soon as they become completely connected.

Scheduling Visits

Visits occur as a result of applying POP to a node. They may also be scheduled in a plan. During execution of a plan, if new attributes are transferred to symbols already parsed (those that appear to the left of a configuration symbol), further evaluation can be planned. Such parsed symbols become candidates for visits because other attributes in the tree may depend upon their values. In particular, some of the synthetic attributes of the parsed symbol may be evaluated after a visit is made to its subtree. If a translation is formed as a synthetic attribute of the grammar's goal symbol, then only the synthetic attributes of the root of a visited tree are *significant* attributes in the translation. *Action symbols* [LRS76] are attributed terminal grammar symbols that are associated with a position in a production right-hand side but do not occur as an input token. They cause attribute dependent semantic routines to be invoked when the symbols to their left are are recognized. If action symbols are used to perform a translation, then every attribute may be considered *significant*. APP plans can be arranged to evaluate all significant attributes in either of these cases. An advantage of the former scheme is that no subtree needs to be visited unless it will yield new synthetic attributes. In either case, a visit is included in a plan when it is found that significant attributes can be evaluated in a previously parsed symbol.

For example, consider the pair of recognized items below, each from distinct states connected by the transition (C,11):

$$[\ A(01|100) ::= B(101|10)\ \underline{\cdot}\ \hat{\imath}\ C(00|110)\ D(0|00)\ ]$$

$$[\ A(01|100) ::= B(101|11)\ \hat{\imath}\ C(11|110)\ \underline{\cdot}\ D(0|11)\ ]$$

The second inherited attribute of B has become available due to the completed parsing of C. This pair of availability-extended items denotes the evaluation state of the subtree before and after parsing C. The raav of the second item, reflecting the completed parsing of C, identifies newly available recipient (inherited) attributes of a right-hand side symbol of the production. At this point, a visit can be made with the (input set) attributes that are newly available. A visit is made to A's leftmost offspring, B, with the second inherited attribute of B (the first being previously evaluated) as illustrated below.

```
                          ________
                         |        |
                         |   .D   |
            ___________  |        |
           |           | |--------|
           | prod.|  visit   |attr|
           | node |  with    |----|
           | with |<=========|  î | *
           | LHS  |  (B,11)  |    |
           |  B   |          |attr|
           |      |          |----|
           |------|          |    |
           | attr |          |    |
           --------

        Translation        Translation
          Heap               Stack
```

As another example, if a completed ghost state containing the item

[E(01|100) ::= E(100|01) î F(111|11) .]

is visited with the first inherited attribute of the root E and

E(1).I1 := E(0).I1 ,

then a visit is made to the first offspring of the root E as well. The attribute evaluation according to the function above leaves the item as follows:

[E(11|100) ::= E(100|11) î F(111|11) .].

The evaluation of the missing attribute in the offspring E may allow another visit to be scheduled.

Determining Subtree Yields

Kennedy and Warren [KW76] used Knuth's original algorithm for circularity [Knu68a] (later modified to correct an error) to precalculate the minimal synthetic yield for each non-terminal symbol in a grammar. When they scheduled a visit to a production node with a particular (visit) set of inherited attributes, the guaranteed yield was used to formulate evaluation plans. The weakness of this method is that more attributes than the minimal set may become available as a result of the visit, but plans will not schedule their evaluation.

Rather than simply considering the minimal synthetic yield of a subtree, the total yield may be found. The total yield of a visit to a particular subtree root with a set of inherited attributes of the root is the exact set of synthetic attributes of the root that can be evaluated. A yield matrix of Boolean values can depict the dependency of the synthetic attributes of a non-terminal symbol upon its inherited attributes. It is of size CS(A) by CI(A) for symbol A. The corrected Knuth algorithm may be used to tabulate functions that determine the total yield matrix of the left-hand side of a production based on right-hand side yields. Each symbol in general has a set of distinct yields because each individual yield reflects the dependency graph constructed from a particular set of subtrees. Algorithm 5C, adapted from Knuth [Knu68a], is used to compute the sets of possible total synthetic yield matrices TSY(A) for each attributed grammar symbol A.

Algorithm 5C: Total Synthetic Yield

Input: cfg G = (N,T,P,S) and MDi for each production i in P.

Output: a) relations RYi that associate the yield matrices of right-hand side non-terminals of a production i with a yield matrix for the left-hand side.

b) sets TSY(A) of possible total synthetic yield matrices for each symbol A in N.

1) (initialization) Set RYi to the empty set for each production i in P. For each production i (A::= X) in a grammar, construct a matrix of size CS(A) by CI(A). Set row j, column k to 1 if the j-th synthetic attribute of A depends upon the k-th inherited attribute of A. Include the matrix in TSY(A).

2) As long as there exists a production i: A0::=A1...An $\in$ P and elements Yj $\in$ TSY(Aj) for each Aj $\in$ N, $j \geq 1$, such that (Y1,...,Yn) is not yet included in a member of relation RYi, do the following:

   a) Construct a square matrix MD′i with |Don(i)| + |Rec(i)| rows and columns. Each row and column of MD′i represents an attribute occurrence of the production. Each entry is a Boolean value denoting the direct dependence of the row attribute upon the column attribute. Set the entries of MD′i from the yield matrices Y1,...,Yn and production dependency matrix MDi in the following manner:

  i) If the j-th attribute occurrence in production i is a recipient attribute dependent upon the k-th attribute occurrence, a donor, (as shown in MDi), set MD´i(j,k) = 1.

  ii) If the j-th attribute occurrence in production i is a donor attribute of the m-th right-hand side symbol and is shown in Ym to be dependent upon the k-th attribute occurrence in production i, set MD´i(j,k) = 1.

b) Replace MD´i with its Boolean transitive closure to determine all indirect attribute dependencies.

c) Extract from MD´i the yield matrix of its root, Y0, as the matrix determined by deleting all rows and columns not representing attribute occurrences of the left-hand side. Include ((Y1,...Yn),Y0) in RYi and include Y0 in TSY(A0). I

Circularity in the grammar is indicated if, for some i and j, an MD´i is constructed such that MD´i(j,j) = 1.

The relations RYi are used at LCAG generation time to determine the yield matrix of the left-hand side of any completed ghost state based on the yields of its non-terminal offspring. The synthetic yield of the left-hand side of production i with a right-hand side of all terminals can be found from MDi alone. Since yield information is necessary in visit scheduling, it must accompany each parsed symbol in recognized items. The yield matrix thus becomes a part of each parsed symbol instance and like the raav, it is used in ghost state to ghost state transitions. Yield matrices can be simplified when they are being added to availability-extended states by eliminating rows in which synthetic attributes are already evaluated and eliminating columns in which inherited attributes are evaluated. This simplification potentially leads to fewer possible states and in effect smaller matrices. The state chosen for a production node as a result of an announce action must consider the yields of each left corner symbol.

Members of TSY(A), yield matrices, are used at evaluator-generation time to construct plans. When a subtree with root yield matrix Y is visited with the set I_in of inherited attributes evaluated in the root, a set S_out of synthetic attributes yielded from the tree is found by the equation:

$$\overline{S\_out} = Y \times \overline{I\_in} \qquad (1)$$

Plan Construction

Plans are constructed for a production node state by considering the recognized item of the state and the daav's that will potentially update it. At the termination of the construction of a plan, the next state to be entered by the translation is determined. The process of constructing plans can create new states, which again require new plans. The plan formulation algorithm 5D is a minor modification of Kennedy and Warren´s. It treats the case in which only synthetic attributes are significant for visits.

Algorithm 5D: Creation of Plans

Input: recognized item I in state Q
update to daav(I): VS $\in$ (0,1)*.

Output: PLAN(Q,VS)

1) Let PLAN be an empty sequence of instructions and
daav´(I) = daav(I) or VS∘[0]CS(X)
if I is of form [A::=X.]
= daav(I) or [0](CI(A)+CS(X)).VS.[0]CS(Y)
if I is of form [A::=X.B Y]

2) Compute raav´(I) for daav´(I) by equation (1) of chapter 2. Include in PLAN all evaluation rules Fp,n marked by newly available attributes in raav´(I).

3) Using equation (1) of this chapter for each offspring in turn that appears before the configuration marker with I_in = raav´(I/B,j), find S_out. If S_out for the k-th offspring includes attributes not currently evaluated then include in PLAN the instruction Visit(k, raav´(I/B,j)), where

the k-th offspring is the j-th occurrence of B in the production. Update daav′(I) with the yield S_out of the visit. Repeat (2) if a visit was scheduled in this step.

4) For each symbol B prior to the configuration marker with daav(I/B,j) ≠ daav′(I/B,j) = [1]CS(B), include Prune(k) in PLAN such that the j-th occurrence of B in the production is the k-th symbol in the right-hand side.

5) Transform Q into Q′ by replacing daav(I) with daav′(I). The new raav can be calculated in the regular fashion. If Q is a completed state, include Update(Q′) in PLAN. If B is not completed then Q′ will become its successor in the state graph. I

When all of a tree′s attributes are significant, step 3 is modified to allow visits with empty yields as long as the visit set is not empty, and step 4 delays pruning instructions until all attributes are evaluated.

States need to be created for every combination of daav and right-hand side yield matrices for each completed item. For each such combination, plans are constructed for each potential visit set of attributes. A visit set is possible for a state if a visit is scheduled for a symbol, yield function pair that matches the left-hand side of the state′s production. A state is considered visitable if a visit is scheduled to its root symbol during complete plan construction.

The following algorithm is used to construct a complete set of plans and find all possible visit sets.

Algorithm 5E: Complete plan scheduling

1) Compute TSY(A) for each A in N using algorithm 5C.
2) Enhance the LC-availability graph to include yield matrices with each parsed symbol in recognized items and transitions from recognized item states.
3) Repeat this step until no more plans can be constructed.
    a) If there exists a state Q with a non-completed recognized item and a transition (A,V,Y) out of Q such that PLAN(Q,V) has not been constructed, compute PLAN(Q,V) using algorithm 5D. Replace the state Goto(Q,(A,V)) with the resultant state Q´ of algorithm 5D.
    b) If there exists a completed state Q with left-hand side A with yield matrix Y in its only item, and a visit with visit set V has been scheduled in some plan to a symbol A with yield matrix Y, and PLAN(Q,V) has not been contructed, then compute PLAN(Q,V).

The above algorithm must halt since there are only a finite number of visit set, visitable state possibilities.

After a subtree visit is terminated, the APP continues in its parsing-and-evaluation mode, properly reflecting the total yield of the root of the subtree visited. The resultant state Q´ (determined in the PLAN construction algorithm 5D) reflects such changes and becomes the new successor to non-completed recognized item ghost states. States on the heap are altered to identify the evaluation changes that occur as a result of a visit by an Update instruction in the plan.

Translation Example

To illustrate the generalized APP translation, the grammar EG of chapter 4 is altered to include a forward reference. The inherited attribute "op" of B is dependent upon the synthetic attribute of z rather than y in production 1 (figure 5.1). The LCAG and plans of the grammar EX are presented in figure 5.2. The sequence of configurations with references to the heap in the translation of the same sentence of chapter 4

x(3)y(2)x(4)y(2)z(5)

appears in figure 5.3.

EG = (N,T,P,A)

N = {A,B}   T = {x,y,z}

S(A) = S(B) = {typ,val}

S(x) = S(y) = S(z) = {val}

I(B) = {op}

P includes:

A ::= x y $\hat{1}$ B z

    B.op := x.val + z.val;

    A.val := B.val + y.val + z.val;

    A.typ := B.typ;

A ::= x y $\hat{2}$ z B

    B.op := x.val * z.val;

    A.val := B.val + y.val;

    A.typ := B.typ;

B ::= x $\hat{3}$ y

    B.val := if B.op > 0 then x.val + y.val
                                  else x.val - y.val;

    B.typ := 1;

B ::= x $\hat{4}$ z

    B.val := (sign(B.op) * x.val) / z.val;

    B.typ := 2;

Figure 5.1 Attributed Grammar : EX

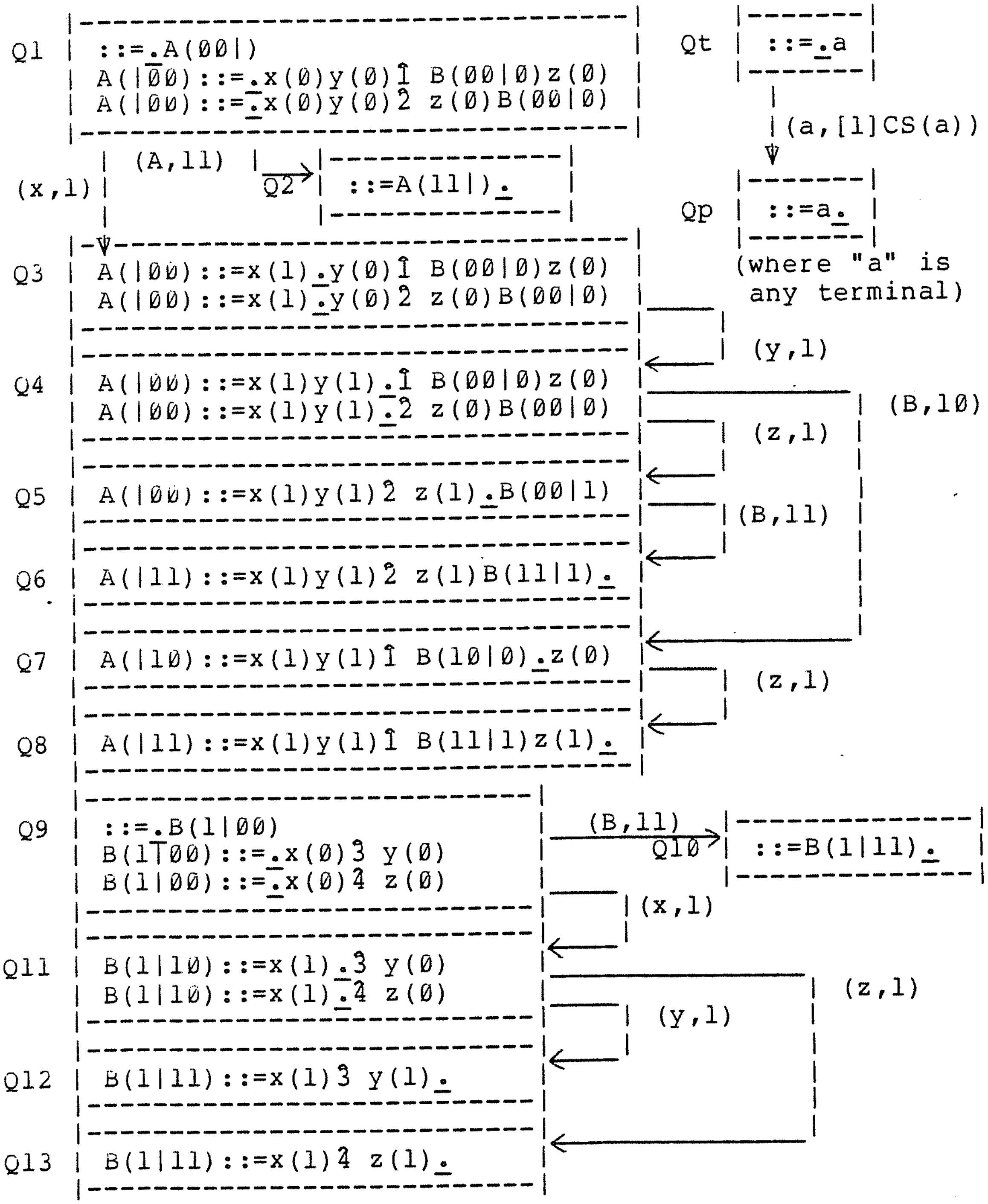
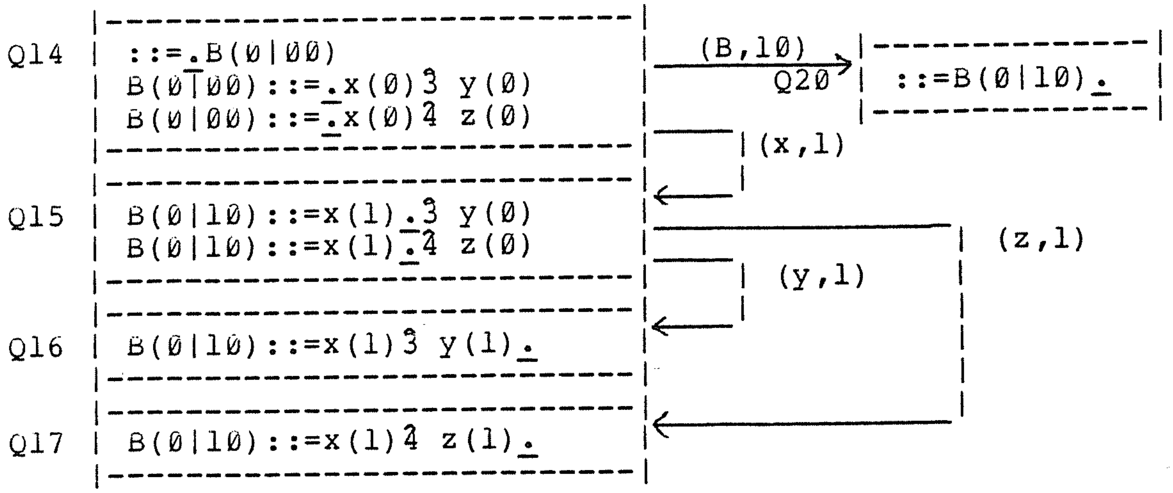


Figure 5.2 Generalized LCAG(EX) with Plans



PLAN(Q4,z(1)) = {F2,1; Prune(1); Prune(2); Prune(3)}

PLAN(Q4,B(10))= {F1,1; Prune(1); Prune(2)}

PLAN(Q5,B(11))= {F2,2; F2,3; Prune(4)}

PLAN(Q7,z(1)) = {F1,1; Visit(3,(1)); F1,2; F1,4; Prune(3)}

PLAN(Q15,y(1))= {F3,2; Prune(1)}

PLAN(Q15,z(1))= {F4,2; Prune(1)}

PLAN(Q16,B(1))= {F3,1; Update(Q12)}

PLAN(Q17,B(1))= {F4,1; Update(Q13)}

Figure 5.2 (continued)

```
   Tstack                                                 Input

(Q1,A[-.-],1)                                  x(3)y(2)x(4)y(2)z(5)

* shift x(3) *

(Q1,A[-.-],1)(Q3,x[3],∅)                           y(2)x(4)y(2)z(5)

* shift y(2) *

(Q1,A[-.-],1)(Q3,x[3],∅)(Q4,y[2],∅)                     x(4)y(2)z(5)

* announce 1 *

(Q1,A[-.-],1)(Q4,1̂[-.-:3:2:-.-.-:-],∅)(nil,z[-],1)
    (Q14,B[-.-.-],2)                                    x(4)y(2)z(5)

* shift x(4) *

(Q1,A[-.-],1)(Q4,1̂[-.-:3:2:-.-.-:-],∅)(nil,z[-],1)
    (Q14,B[-.-.-],2)(Q15,x[4],∅)                             y(2)z(5)

* announce 3 *

(Q1,A[-.-],1)(Q4,1̂[-.-:3:2:-.-.-:-],∅)(nil,z[-],1)
  (Q14,B[-.-.-],2)(Q15,3̂[-.1.-:4:-],∅)(Qt,y[-],1)    y(2)z(5)

* shift y(2) *

(Q1,A[-.-],1)(Q4,1̂[-.-:3:2:-.-.-:-],∅)(nil,z[-],1)
  (Q14,B[-.-.-],2)(Q15,3̂[-.1.-:4:-],∅)(Qt,y[-],1)          z(5)
  (Qp,y[2],∅)
```

Figure 5.3 Translation with EX

```
* pop y *
(Q1,A[-.-],1)(Q4,1̂[-.-:3:2:-.-.-:-],0)(nil,z[-],1)
  (Q14,B[-.-.-],2)(Q16,3̂[-.1.-:4:2],0)                      z(5)
* which associates Q16 with Q20 and becomes *
(Q1,A[-.-],1)(Q4,1̂[-.-:3:2:-.-.-:-],0)(nil,z[-],1)
  (Q14,B[-.-.-],2)(Q20,B[-.1.-],0)                          z(5)
* pop B *
(Q1,A[-.-],1)(Q7,1̂[1.-:3:2:-.1.-:-],0)(Qt,z[-],1)          z(5)
                ↓
    Heap:     (Q16,3̂[-.1.-:4:2],0)
* shift z(5) *
(Q1,A[-.-],1)(Q7,1̂[1.-:3:2:-.1.-:-],0)(Qt,z[-],1)(Qp,z[5],0)
                ↓
    Heap:     (Q16,3̂[-.1.-:4:2],0)
* pop z, PLAN(Q7,z(1)), PLAN(Q16,B(1)) *
(Q1,A[-.-],1)(Q8,1̂[1.13:3:2:8.1.6:5],0)
                ↓
    Heap:     (Q12,3̂[8.1.6:4:2],0)
* which immediately becomes *
(Q1,A[-.-],1)(Q2,A[1.13],0)
* pop A *
  resulting in:  (Q′,A[1.13],0)
```

Figure 5.3 (continued)

Generalized APP Properties

The important properties of the APP are summarized in the following theorems concerning its capabilities and time and space execution requirements.

Lemma 5.1

In an APP translation with a non-circular attributed grammar in which all attributes are considered significant, if, in a heap node, an unevaluated attribute "a" has all direct and indirect donors in stack nodes evaluated, then the next plan executed at the heap node's ancestor on the stack will cause a sequence of plans and visits during which "a" will be evaluated.

Proof:

The proof is by induction on the length of the longest dependecy path from an unevaluted attribute "a" on the heap to any evaluated donor attribute. Dependency path length is well-defined since the attributed grammar is assumed non-circular. Without loss of generality, only attributes of symbols on the right-hand side of productions need be considered since all symbols with attributes on the heap occur as part of a right-hand side.

<u>Base</u> <u>step</u>: greatest path length = 1.

All donor attributes for "a" must occur in symbols of the production in which "a" occurs on the right-hand side. One such donor must exist in the left-hand side symbol (which is on the stack) or "a" would have been previously evaluated. If all donors were in the right-hand side and were evaluated, then the last plan executed at the node would have had "a" scheduled for evaluation in step 2 of algorithm 5D. When a plan evaluates the inherited donor(s) of "a" on the stack, a visit will be scheduled to this node in algorithm 5D, step 3 since all attributes are considered significant. The visit (algorithm 5B) copies any needed donors from the left-hand side symbol on the stack. All other donors must have been previously evaulated, since the longest dependency path length is one. Thus the raav′ in algorithm 5D step 2 will show "a" available for evaluation and schedule it.

Induction step: greatest path length = n, hypothesis holds for length < n.

By the induction hypothesis, all direct donor attributes must have greatest path length to evaluated attributes less than n. Thus by the hypothesis they will have evaluations scheduled in a sequence of visits. All such donors exist in the right-hand symbols or the left-hand side symbol of the production containing "a". When the last direct donor is evaluated by some visit plan, then "a" will also be scheduled for evaluation by step 2 of algorithm 5D in that plan (if the last attribute was in the right-hand side), or a visit will be scheduled by step 3 of algorithm 5D to the node containing "a" from its parent node and the ensuing plan will evaluate "a" (if the last donor attribute evaluated occurs in the left-hand side). I

Theorem 5.2

In a translation with the APP in which all attributes are considered significant, if an unevaluated attribute "a" exists in any node on the translation stack or heap, then either:

(1) that part of the syntax tree structure containing the dependency paths that lead to "a" is not completely recognized, or

(2) attribute "a" will be evaluated by the time the parser consumes another input symbol, or

(3) the attributed grammar is circular.

Proof:

Case i: the state of the node containing "a" does not contain a recognized item.

The structure is not complete, so condition (1) holds trivially.

Case ii: the node containing "a" is a production node on the stack.

If all donors for "a" are evaluated, then the last plan executed at the node (if any) would have evaluated "a". If an unevaluated donor (direct or indirect) occurs in a symbol beyond the configuration symbol, then (1) is satisfied due to the unrecognized sibling. Unless (1) and (3) are true, all inherited donors are evaluated due to successful predictions and offspring visits. Before the next scan step can occur, a plan is executed with the latest pop or announce action. Unless the grammar is circular, by repeated application of Lemma 5.1, all direct and indirect donors of "a" become evaluated by a sequence of plans and visits stemming from that plan. In the return from that visit "a" will be evaluated due to step 2 of algorithm 5D. Thus (2) must hold.

Case iii: the node containing "a" is on the heap.

Unless (1) or (3) is true, Lemma 5.1 shows that "a" will be evaluated when the last ancestor node on any dependency path to "a" is recognized by an announce or pop step. I

The next corollary follows directly from theorem 5.2 and the fact that Demers´ parsing method with minimal left corners recognizes productions with the minimal number of scanned input symbols [Dem77].

Corollary 5.3

The APP evaluates each attribute of a non-circular GLC(k) attributed grammar with minimal left corners at the earliest possible time.

Theorem 5.4

Ignoring the complexity of the evaluation rules and assuming that attribute values can be represented in a bounded space, the APP operates in time and space at most linearly proportional to the length of its input.

Proof:

The linear space result for the APP is based on the observation that the size of any syntax tree for an unambiguous context-free grammar is at most linearly proportional to the length of the sentence it represents [AU73]. The space used in an APP translation stack and heap is bounded above by the size of the syntax tree of the sentence, and the size of each node is bounded by a constant fixed by the size of the attributed grammar. To fix a bound on the size of any node requires the assumption that all attribute values can be represented in a bounded space.

For a given attributed grammar, there is a bound on the number of attributes associated with any one symbol. The GLC parse itself is linear in its time requirements [Dem77]. It must be shown that the number of visits made by the evaluator is also linear. Since (1) no visit is ever made without evaluating at least one attribute, (2) there are at most a linear number of symbols to visit, and (3) there is a constant bound on the number of attributes per symbol, there are at most a linear number of visits made. I

The APP in its full generality is impressive in capability but potentially very unwieldy in both the time required to generate it and the storage required to hold its states. Jayazeri, et al [JOR75] have shown that the worst case execution time of Knuth's circularity algorithm is exponential. The number of parse evaluation states is also potentially exponential. Whether or not attributed grammars used to specify typical programming languages approach the worst case is unknown. Research and practical experience in the use of attributed grammars is needed.

There are reasons to believe that typical programming languages are not difficult to deal with. The construction of SLR(k) parsers for programming language syntax from BNF specifications is frequently undertaken despite exponential worst case time and space bounds [DeR69]. Knuth´s algorithm for circularity and the construction of the generalized LCAG become expensive when many possible yield functions exist for each of the non-terminals in the grammar. In practice, this situation does not appear to arise. A specification would be extremely difficult to comprehend or design if a given input set of inherited attributes to a subtree produced many variations in output sets of synthetic attributes for the tree, dependent always upon the structure of the subtree. Attributes should most likely take on variations in <u>value</u> for corresponding structural variations, rather than undergo variations in <u>availability</u>. Clearly this principle is a useful guideline in language specification design and might be made a requirement. The current trend of limiting forward references in programming languages also aids in the simplification of APP translation since grammars become more nearly LC-attributed and subtrees are less likely to be saved.

Several simplifications to the method of visit plan scheduling are worth describing. One is the previously mentioned method of Kennedy and Warren. Only a single minimal yield function is associated with a particular grammar symbol. Each state then has only a single possible configuration of yields and the plans are simpler to construct. More visits to a particuar subtree may be necessary due to the imperfect specification of the yields, and a smaller class of attributed grammars can be evaluated [Kw76].

A second simplification permits another interesting subset of attributed grammars to be defined: those that are partial order attributed.

> A <u>partial</u> <u>order</u> <u>attributed</u> (PO-attributed) grammar is an attributed context-free grammar G = (N,T,P,S) such that there exists a partial ordering of the right-hand side non-terminals in any production, such that the attributes of one symbol are not donors for any attributes of the symbol itself or of symbols preceding it in the ordering.

PO-attributed grammars are guaranteed non-circular because inherited attributes cannot indirectly depend upon synthetic attributes of the same symbol; any dependency cycle is broken at the top. Grammars can be recognized as members of the class by a simple topological sort [Knu68b] on each of the production right-hand sides (worst case execution time is O(n**2) where n is the length of the longest right-hand side). The recognition algorithm simply looks for symbol-wise dependency cycles of the right-hand side of each production.

When evaluating attributes of PO-attributed grammars, no subtree need be visited twice if its visit is delayed until all its inherited attributes are available. All inherited attributes of an offspring become available by the time its parent and siblings that precede it in the ordering have been visited. Plans in the scheduling algorithm are therefore trivial for PO-attributed grammars and are optimal in the sense of minimal tree traversal path length. L-attributed grammars are a special case of PO-attributed with the partial ordering being the left-to-right position in the right-hand side.

Experience will tell if typical attributed grammars do indeed fit these simplified grammar classes. While the modifications that come with the more restrictive classes are attractive, the general model is an important foundation in the theory of attributed grammar evaluation.

Chapter 6

Significance of the Attributed Pushdown Processor

The attributed pushdown processor makes several significant contributions to the theory of translation. It provides a parse-time attribute evaluator that generalizes all previous evaluation methods. A formal model is established for efficient single-pass compilations that is not hampered by forward references. Inspection of the algorithms involved in APP construction uncovers several attributed grammar specification principles that aid in the simplification of processor construction. Algorithms are developed that can be used to automate the generation of semantic analyzers.

The use of the generalized left-corner parser enables the APP to surpass the power of previous evaluation techniques. The methods of Lewis *et al* [LRS74; LRS76] allow evaluation of two small portions of the specification space of attributed grammars (see figure 6.1). The simplified attributed pushdown processor, because it takes advantage of both LL(k) and LR(k) parsing techniques, covers the additional specifications that fall between the L-attributed and S-attributed processors. Both those techniques are

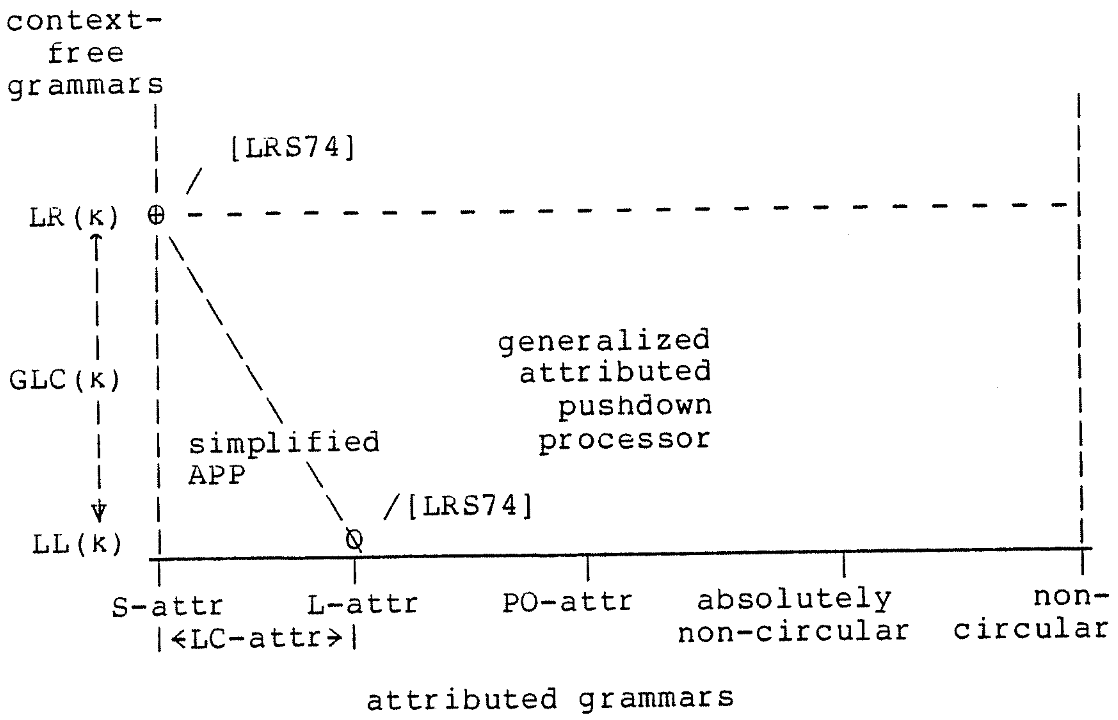


Figure 6.1 Attributed grammar specification space

included as special cases of the APP with no degradation of efficiency. The inclusion of subtree retention and a powerful tree-walk facility for delayed evaluation (modelled after the work of Kennedy and Warren [KW76]) allows enhanced capabilities that now cover all LR(k) non-circular attributed grammars.

The efficiency of the APP is demonstrated in its linear time and space properties and its ability to keep syntax trees well pruned. Attributes are evaluated at the earliest possible point so that they may be used in the remainder of a single-pass compilation for parsing decisions, error recovery and correction, code generation and optimization.

Attributed grammars are a powerful tool for language specification and with the APP they become a powerful tool for translator design and automated implementation. There remain many areas for research in attributed grammar theory and application. Much experience can be gained in the use of attributes grammars to specify current and new programming languages. Several particular applications would benefit from further study:

1) Investigation of the use of the APP and LC-attributed grammars coupled with Milton´s attributed parsers [Mil77].
2) The use of LC-attributed grammars in error recovery and correction. The top-down corrector of Fischer et al [FMQ77] might be expanded to more general parsing techniques.
3) Consideration of details for efficient representations and implementations of code generation and symbol table specification using attributed grammars.
4) Methods for producing layers of LC-attributed grammars, to be parsed and evaluated in separate passes using transformations similar to those of Schulz [Sch76]. In this scheme, subtrees would not be retained, nor would visits be required, but forward references could occur

and be satisfied on distinct passes. This method might provide significant improvements in storage requirements.

The design principles for attributed grammar specification take into consideration the factors that can either increase the number of LCAG states or make the plan algorithm more tedious. A symbol's attribute availabilities are best made independent of the symbol's subtree. Further, plan construction is vastly simplified by knowledge of an offspring attribute evaluation ordering that is consistent for each instance of a production. The subclass of PO-attributed grammars is a usable result following these principles.

An important feature of this model for single-pass compilation is its natural correspondence to common compiler designs. Many of the features and operations are modelled after current translation techniques, which makes the model easy to comprehend and compare to existing compilers. It provides an enhanced formalism in which to describe existing methodology, encompassing ideas such as semantic stacks, semantic routines, forward references, and environments. Finally, it allows the development of automation techniques which free the language designer from many tedious implementation details and allow him to concentrate on formal language specification at high levels.

References

[Aho68] Aho, A. V. "Indexed grammars -- an extension of context-free grammars." Journal ACM 15:4, (1968) 647-671.

[AJ74] Aho, A. V., and S. C. Johnson. "LR-parsing." Computing Surveys 6:2 (1974) 99-124.

[AN76] Amirchahy, M. and D. Néel. "L'optimisation de code pourra-t-elle s'affirmer grâce aux attributs sémantiques." Rapport de Recherche #162, IRIA Laboria, March 1976.

[AU69a] Aho, A. V. and J. D. Ullman. "Properties of syntax directed translations." Computer and Systems Science 3 (1969) 319-334.

[AP72] Aho,A. V. and T. G. Peterson. "A minimum distance error-correcting parser for context-free languages." SIAM Journal on Computing 1:4 (1972) 305-312.

[AU69b] ------. "Syntax directed translations and the Pushdown Assembler." Journal of Computer and Systems Sciences 3:1 (1969) 37-52.

[AU71a] ------. "Characterizations and extensions of pushdown translations." Math Systems Theory 5 (1971) 172-192.

[AU71b] ------. "Translations on a context-free grammar." Information and Control 19 (1971) 439-475.

[AU73] ------. The Theory of Parsing, Translation, and Compiling, 1 & 2. Prentice Hall, Englewood Cliffs, NJ (1973).

[Bak75] Baker, A. C. "A generalized lexical scanner for a translator writing system." Report #UIUCDCS-R-73-596, Dept. of Computer Science, University of Illinois - at Urbana Champaign (1973).

[Boc76] Bochmann, G. V. "Semantic evaluation from left to right." Communications ACM 19:2 (1976) 55-62.

[CM76] Chirica, L. M. and D. F. Martin. "An algebraic formualation of Knuthian semantics." 17th Annual Symposium of Foundations of Computer Science, IEEE CH1133-8, Houston, TX (October 1976) 127-136.

[Dem77] Demers, A. "Generalized left-corner parsing." Conference Record of the Fourth Annual Symposium on Principles of Programming Languages, Los Angeles, CA (Jan. 1977) 170-181.

[DeR69] DeRemer, F. L. "Practical translators for LR(k) languages." Project MAC TR-65, M.I.T., Cambridge, MA (1969).

[Fan72] Fang, I. "FOLDS, a declarative formal language definition system." Technical Report STAN-72-329, Stanford University, (1972).

[Fel66] Feldman, J. A. "A formal semantics for computer languages and its application in a compiler-compiler." Communications ACM 9:1 (1966) 3-9.

[FG68] Feldman, J. A. and D. Gries. "Translator writing systems." Communications ACM 11:2 (1968) 77-113.

[Flo63] Floyd, R. W. "Syntactic analysis and operator precedence." Journal ACM 10 (1963) 316-333.

[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 Fourth Annual Symposium on Principles of Programming Languages, Los Angeles, CA (Jan. 1977) 97-103.

[GR73] Graham, S. L. and S. P. Rhodes. "Practical error recovery in compilers." Conference Record of the ACM Symposium on Principles of Programming Languages, (Oct. 1973) 52-58.

[Gri71] Gries, D. Compiler Construction for Digital Computers. Wiley & Sons, Inc., New York (1971).

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

[HW73] Hoare, C. A. R. and N. Wirth. "An axiomatic definition of the programming language PASCAL." Acta Informatica 2 (1973) 335-355.

[Iro63] Irons, E. T. "An error-correcting parser algorithm." Communications ACM 6:11 (1963) 669-673.

[JW75] Jazayeri, M. and K. G. Walter. "Alternating semantic evaluator." In ACM ´75, Proceedings of the Annual Conference, Minneapolis, MN (Oct 1975) 230-234.

[JOR75] Jazayeri, M., W. F. Ogden and W. C. Rounds. "On the complexity of the circularity test for attribute grammars." Conference Record of the Second Annual ACM Symposium on Principles of Programming Languages, Palo Alto, CA, (Jan. 1975) 119-129.

[JPAR68] Johnson, W. L., J. H. Porter, S. I. Ackley, and D. T. Ross. "Automatic generation of efficient lexical processors using finite state techniques." Communications ACM 11:12 (1968) 805-813.

[Knu65] Knuth, D. E. "On the translation of languages from left to right." Information and Control 8 (1965) 607-639.

[Knu68a] ------. "Semantics of context-free languages." Mathematical Systems Theory 2 (1968) 127-145. Correction appears in Mathematical Systems Theory 5 (1971) 95.

[Knu68b] ------. The Art of Computer Programming, Volume 1. Addison-Wesley, Reading, MA (1968).

[Knu71a] ------. "Examples of formal semantics." Lecture Notes in Mathematics, No. 188, Springer-Verlag, Berlin (1971).

[Knu71b] ------. "Top-down syntax analysis." Acta Informatica 1:2 (1971) 79-110.

[Kw76] Kennedy, K. and S. K. Warren. "Automatic generation of efficient evaluators for attribute grammars." Conference Record of the Third Annual ACM Symposium on Principles of Programming Languages, Atlanta, GA (Jan. 1976) 32-49.

[LRS74] Lewis, P. M., D. J. Rosenkrantz, and R. E. Stearns. "Attributed translations." Journal of Computer and Systems Sciences 9 (1974) 279-307.

[LRS76] ------. Compiler Design Theory. Addison-Wesley, Reading, MA (1976).

[LS68] Lewis, P. M. and R. E. Stearns. "Syntax directed transduction." Journal ACM 15:3 (1968) 465-488.

[MHW70] McKeeman, W. M., J. J. Horning, and D. B. Wortman. A Compiler Generator. Prentice Hall, Englewood Cliffs, NJ (1970).

[Mil77] Milton, D. N. "Syntactic specification and analysis using attributed grammars." Ph.D. thesis, Computer Sciences Dept., University of Wisoconsin, Madison, in preparation (1977).

[MLB76] Marcotty, M., H. F. Ledgard, and G. V. Bochmann. "A sampler of formal definitions." Computing Surveys 8:2 (1976).

[PAN76] Pair, C., M. Amirchahy, and D. Néel. "Preuves de descriptions de traitments de textes par attributs." Rapport de Recherche #163, IRIA Laboria (March 1976).

[Sch76] Schulz, W. A. "Semantic analysis and target language synthesis in a translator." Ph.D. thesis, Computer Sciences Dept., University of Colorado, Boulder (1976).

[SL69] Stearns, R. E. and P. M. Lewis. "Property grammars and table machines." Information and Control 14 (1969) 524-549.

[Sol77] Solomon, M. "Theoretical issues of the implementation of programming languages." Ph.D. thesis, Computer Sciences Dept., Cornell University, Ithaca (1977).

[SU72] Sethi, R. and J. D. Ullman. "The generation of optimal code for arithmetic expressions." Journal ACM 17:4 (1972) 715-728.

[Weg72] Wegner, P. "The Vienna definition language." Computing Surveys 4:1 (1972) 5-63.

[Wil72] Wilner, W. T. "Formal semantic definition using synthesized and inherited attributes." In Formal Semantics of Programming Languages, R. Rustin, ed., Prentice Hall, Englewood Cliffs, NJ (1972) 25-39.

[WMPK69] van Wijngaarden, A., B. J. Mailloux, J. E. Peck, and C. H. A. Koster. "Report on the algorithmic language ALGOL 68." MR 101, Mathematisch Centrum, Amsterdam, The Netherlands (1969).

[WW66] Wirth, N. and H. Weber. "EULER: A gereralization of ALGOL, and its formal definition: Part II." *Communications ACM* 9:2 (1966) 89-100.