1988

Semantic specification using tree manipulation languages

Randall Paul Meyer
College of William & Mary - Arts & Sciences

Semantic specification using tree manipulation languages

Meyer, Randall Paul, Ph.D.

The College of William and Mary, 1988

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

SEMANTIC SPECIFICATION USING TREE MANIPULATION LANGUAGES

A Dissertation
Presented to
The Faculty of the Department of Computer Science
The College of William and Mary in Virginia

In Partial Fulfillment
Of the Requirements for the Degree of
Doctor of Philosophy

by
Randall P. Meyer
1988

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Randall P. Meyer

Randall P. Meyer

Approved, May 1988

Robert E. Noonan

Robert E. Noonan

Stefan Feyock

Stefan Feyock

Larry J. Morell

Larry J. Morell

William L. Byn

William L. Bynum

Carl W. Vermeulen

Carl W. Vermeulen
Department of Biology

Contents

ACKNOWLEDGEMENTS

I express my appreciation to Professor Robert E. Noonan who served as my disseration advisor. I thank Professor Larry J. Morell for countless hours of discussion and critical review of this manuscript. Professors William Bynum, Stephan Feyock, and Carl Vermeulen, and Louis Slothouber and Fred Stevens deserve recognition for their comments, ideas, and support.

List of Figures

ABSTRACT

Software tools are used to generate compilers automatically from formal descriptions of languages. Methods for specifying the syntax of languages are well-established and well-understood; however, methods for formal semantic specification are not. The method most commonly used for semantic specification is an attribute grammar.

This thesis examines an alternative method of semantic specification. TreeSem is defined as a Tree Manipulation Language applicable to semantic specification. A TreeSem program is easier to read and to write than a corresponding attribute grammar specification.

Algorithms for translation of a TreeSem program into an equivalent attribute grammar specification, and for translation of an attribute grammar specification into an equivalent TreeSem program are presented. Proof of correctness of the algorithms is discussed. The dual translations show the theoretical "specification power" of TreeSem to be the same as that of attribute grammars. Also, since both translations are provided, the compiler writer is free to choose the semantic specification method he wishes to use. The appropriate translation can be applied to implement the compiler using the more efficiently interpreted method, as research continues to improve the executable efficiency of either method.

Semantic Specification Using Tree Manipulation Languages

Chapter 1

Introduction

A compiler translates a program from its original source language into an equivalent program in another target language. The overall translation task is complicated, so it is customary to divide the compiler into phases, each of which performs a small portion of the overall translation process. The lexical analysis and syntax analysis phases (the front end) read the source program text and convert it into an internal representation. They are concerned primarily with the syntactic structure of the input program. The semantic analyzer and intermediate code generator (the middle end) extract the meaning of the program from its internal representation and select statements from an intermediate language that have the same effect as this determined meaning. The code optimizer and code generator (the back end) perform the task of writing an efficient representation of the intermediate code in the target language.[1]

Automatic compiler construction is the process of generating a compiler (interpreter) for a language from a formal specification of its syntax and semantics. The availability of tools to aid in automatic compiler construction varies widely depending on the phase of compilation being considered. Whereas techniques for parsing and lexical analysis are well-understood and in common use, cur-

[1] Aho, Sethi, and Ullman, page 20.

rent methods for semantic processing and code generation and optimization are generally inadequate or inefficient. This research is concerned with defining a formalism for semantic specification that will be more expressive than existing methods while remaining transformable into an efficient semantic processor for the language.

1.1 Grammars and Parsing

The syntax of a language is specified using a *Context Free Grammar*[2] (CFG), which is a four-tuple $(\mathbf{N}, \mathbf{T}, \mathbf{S}, \mathbf{P})$. $\mathbf{T}$ is the set of terminal symbols in the language. $\mathbf{N}$ is a set of nonterminal symbols employed in the grammar, such that $\mathbf{N} \cap \mathbf{T} = \emptyset$. $\mathbf{V} = \mathbf{T} \cup \mathbf{N}$ is the vocabulary of the grammar. One nonterminal symbol is distinguished as the start symbol $\mathbf{S}$. $\mathbf{P}$ is a set of productions of the form $\mathbf{X}_0 \rightarrow \mathbf{X}_1\mathbf{X}_2 \ldots \mathbf{X}_n$, where $\mathbf{X}_0 \in \mathbf{N}$ and $\mathbf{X}_1 \ldots \mathbf{X}_n \in \mathbf{V}$. For a given CFG $\mathcal{G}$, strings in the language described by the CFG, $\mathbf{L}(\mathcal{G})$, are generated from the start symbol by replacing nonterminals in a *working string* with right hand sides of productions whose left hand sides match the symbol being replaced. These replacements are done until only terminal symbols remain in the working string. The sequence of replacements of nonterminals by right hand sides of productions is referred to as a *derivation* of the terminal string. In a leftmost derivation, the leftmost nonterminal is always chosen for replacement; and similarly in a rightmost derivation the rightmost nonterminal symbol is replaced.

Parsers determine whether a given string is an element of a particular language. Many parsing methods exist. **LR** parsers attempt to find a rightmost derivation of the input string, based on a grammar describing the language. **LL** and recursive descent parsers attempt a leftmost derivation of the input

[2] Slanted type will be used throughout this dissertation whenever a new term is being introduced. Subsequent uses of the term will appear in normal type.

string.[3] Particular forms of grammars can be translated into parsers by hand in a straightforward manner, but for large grammars, this translation is tiresome and error-prone.

An alternative to hand-coded parsers is a parser generator, several of which are in widespread use (YACC,[4] PARGEN[5]). A parser generator automatically translates a grammatical description of a language into a parser that recognizes strings in that language. Although the resulting parsers are sufficiently fast and compact in size as to be practically useful, research continues in an attempt to optimize both the parser generators and the resulting parsers.

Whereas the syntactic analysis of a language can currently be handled by automatically generated parsers, the semantic phase of translation is usually hand-coded. The compiler writer utilizing a parser generator specifies the semantics of particular strings in the language by associating semantic actions with productions in the grammar. Depending on the particular parser generator, these semantic actions are written in one of several common programming languages. The effects of these semantic actions (particularly with respect to the scope of allowable assignments and references) are unrestricted. As productions are recognized (or reduced) during parsing of the input string, the associated semantic action is executed. The compiler writer must fully understand the parsing strategy in order to effectively supply the correct semantic actions. Figure 1.1 illustrates the use of a parser generator in automatic compiler generation.

Several formalisms for describing language semantics have been developed, including denotational semantics, axiomatic semantics, and attribute grammars.[6]

[3] Aho, Sethi, and Ullman, pages 40–48.

[4] *The UNIX^{TM} System User's Manual.*

[5] Noonan and Collins.

[6] Pagan.

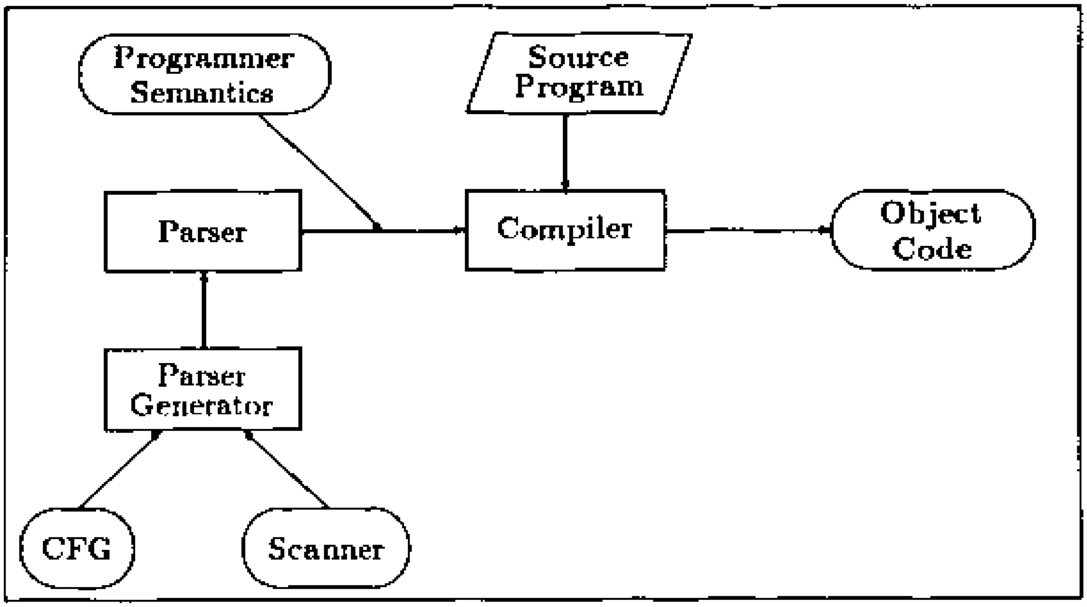


Figure 1.1: Automated Compiler Generation

Of these, *attribute grammars* (AG) are currently the most popular mechanism for describing the semantics in such a way that an evaluator for the language can be automatically generated from the description. The systems **HLP**,[7] **GAG**,[8] and **LINGUIST-86**[9] all employ attribute grammars as their method of semantic specification.

1.2 Attribute Grammars

Attribute grammars were originally developed by Knuth,[10] who extended earlier work by Irons.[11] An attribute grammar is a reduced CFG augmented with attributes associated with the symbols of the grammar and functions (semantic rules or evaluation rules) which assign values to these attributes. An AG is

[7] Raiha and Saarinen.

[8] Kastens, Hutt, and Zimmermann.

[9] Farrow and Yellin.

[10] Knuth, 1968.

[11] Irons, 1961 and Irons, 1963.

a seven-tuple **(N, T, S, P, A, R, C)**. **N**, **T**, **S**, and **P** are defined as for a CFG. We assume the underlying CFG is reduced, i.e., it contains no useless productions. **A** is a finite set of attribute symbols or names and associated ***types*** or set of values they may take on. **R** is the set of semantic rules. These will be defined below. **C** is a set of semantic conditions the attribute values must satisfy in a syntactically correct sentence of the language. For our purposes it will suffice to note that such conditions can be replaced by a distinguished Boolean attribute symbol associated with each nonterminal symbol, and they will not be further discussed. The AG of Figure 1.2[12] will be used to illustrate attribute grammar terminology.

The following definitions are introduced in order facilitate the definition of **R**. define With respect to a grammar symbol X ∈ V, A(X) denotes the set of attribute symbols associated with the symbol X. (Some authors prohibit attributes associated with terminal symbols.) For the AG in Figure 1.2, we have:

$$\begin{aligned} \mathrm{A(t)} &= \mathrm{A(u)} = \mathrm{A(W)} = \emptyset \\ \mathrm{A(Y)} &= \{\mathrm{a,b}\} \\ \mathrm{A(X)} &= \{\mathrm{c,d}\} \end{aligned}$$

A production p: $X_0 \rightarrow X_1X_2\ldots X_n$ is said to have the *attribute occurrence* $a(X_i^p)$ if $a \in A(X_i)$. The notation "a(X)" may also be written as "X.a". The type of an attribute occurrence a(X) is the same as that of the associated attribute symbol A(X). *A(p)* denotes the set of all attribute occurrences of p, and is simply the union of all the attribute occurrences of all the symbols in production p.

[12] Adapted from Kastens, figure 8.

$$
\boxed{
\begin{array}{rcll}
\mathrm{N} & = & \{\mathrm{W}, \mathrm{Y}, \mathrm{X}\} & \\
\mathrm{T} & = & \{\mathrm{t}, \mathrm{u}\} & \\
\mathrm{V} & = & \mathrm{N} \cup \mathrm{T} = \{\mathrm{W}, \mathrm{X}, \mathrm{Y}, \mathrm{t}, \mathrm{u}\} & \\
\mathrm{S} & = & \mathrm{W} & \\
\mathrm{P} & = & \{\mathrm{W} \rightarrow \mathrm{Y} & (0) \\
 & & \ \ \mathrm{Y} \rightarrow \mathrm{X}\ \mathrm{Y} & (1) \\
 & & \ \ \mathrm{Y} \rightarrow \mathrm{t} & (2) \\
 & & \ \ \mathrm{X} \rightarrow \mathrm{u}\} & (3) \\
\mathrm{A} & = & \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}\} & \\
\mathrm{R}(0) & = & \{\mathrm{Y.b} \ \leftarrow 17;\} & \\
\mathrm{R}(1) & = & \{\mathrm{Y}_1\mathrm{.a} \leftarrow \mathrm{X.c}; & \\
 & & \ \ \mathrm{X.d} \ \leftarrow g(\mathrm{Y}_2\mathrm{.a}); & \\
 & & \ \ \mathrm{Y}_2\mathrm{.b} \leftarrow \mathrm{Y}_1\mathrm{.b};\} & \\
\mathrm{R}(2) & = & \{\mathrm{Y.a} \ \leftarrow \mathrm{Y.b};\} & \\
\mathrm{R}(3) & = & \{\mathrm{X.c} \ \leftarrow \mathrm{X.d};\} &
\end{array}
}
$$

Figure 1.2: An Example Attribute Grammar

With respect to Figure 1.2, we have the following A(p):

$$\begin{aligned}
A(0) &= \{Y.a, Y.b\} \\
A(1) &= \{Y_1.a, Y_1.b, X.c, X.d, Y_2.a, Y_2.b\} \\
A(2) &= \{Y.a, Y.b\} \\
A(3) &= \{X.c, X.d\}
\end{aligned}$$

The set of semantic rules associated with production p is denoted *R(p)*. These semantic rules have the form

$$X_i.a \leftarrow f(X_j.b, \ldots, X_n.c),$$

where X_i and $X_j \ldots X_n$ are symbols of grammar rule p. Thus the value of an attribute occurrence $X_i.a$ is defined in terms of the values of other attribute occurrences in the same production p. The semantic rule that assigns a value to $X_i.a$ for production p is denoted by $f^{p}_{X_i.a}$. The $f^{p}_{X_i.a}$ for the AG of Figure 1.2 are:

$f^{0}_{Y.b}$	$=$	$Y.b \leftarrow 17;$	inheriting
$f^{1}_{Y_1.a}$	$=$	$Y_1.a \leftarrow X.c;$	synthesizing
$f^{1}_{X.d}$	$=$	$X.d \leftarrow g(Y_2.a);$	inheriting
$f^{1}_{Y_2.b}$	$=$	$Y_2.b \leftarrow Y_1.b;$	inheriting
$f^{2}_{Y.a}$	$=$	$Y.a \leftarrow Y.b;$	synthesizing
$f^{3}_{X.c}$	$=$	$X.c \leftarrow X.d;$	synthesizing

If a semantic function defines the value of an attribute occurrence $X_0.a$ of the production's left hand side symbol, it is a synthesizing function and the attribute "a" is termed a *synthesized* attribute. Otherwise the function is an inheriting one, and the attribute is an *inherited* attribute. *AI* denotes the set of all inherited attributes and *AS* denotes the set of all synthesized attributes. Correspondingly, *AI(X)* is the set of inherited attributes of symbol X and *AS(X)* the set of synthesized ones. Referring to Figure 1.2:

$$\begin{array}{rcl} \mathrm{AS} & = & \{a, c\} \\ \mathrm{AI} & = & \{b, d\} \end{array}$$

$$\mathrm{AS(X)}=\{c\} \quad \mathrm{AS(Y)}=\{a\} \quad \mathrm{AS(W)}=\mathrm{AS(t)}=\mathrm{AS(u)}=\emptyset$$

$$\mathrm{AI(X)}=\{d\} \quad \mathrm{AI(Y)}=\{b\} \quad \mathrm{AI(W)}=\mathrm{AI(t)}=\mathrm{AI(u)}=\emptyset$$

A *semantic tree* (derivation tree) is a parse tree where each node $X \in V$ is annotated with attributes $\{a \mid a \in A(X)\}$. In general, synthesized attributes serve to carry information upward from the leaves of a derivation tree towards the root, and inherited attributes move information towards the leaves. An AG is termed *complete* iff

1. There is exactly one semantic rule defining the value of any attribute occurrence.

2. Using the above method of determining AI(X) and AS(X), $\forall X \in V : ((\mathrm{AI}(X) \cap \mathrm{AS}(X) = \emptyset) \wedge (\mathrm{AI}(X) \cup \mathrm{AS}(X) = A(X))$.

A complete AG is *well-defined* iff the values of all attributes of any derivation tree S corresponding to a sentence in $L(\mathcal{G})$ are effectively computable. Only well-defined AGs are useful as specifications to automatic compiler generators. If an AG is not well-defined, it is not possible to automatically construct a compiler that is guaranteed to terminate and assign the proper values to the attributes of all syntactically legal semantic trees in the language defined by the AG. To check this condition formally, we introduce the notion of dependency. The dependency set $D^p_{X_i.a}$ is the set of attribute occurrences used as arguments by $f^p_{X_i.a}$. The $D^p_{X_i.a}$

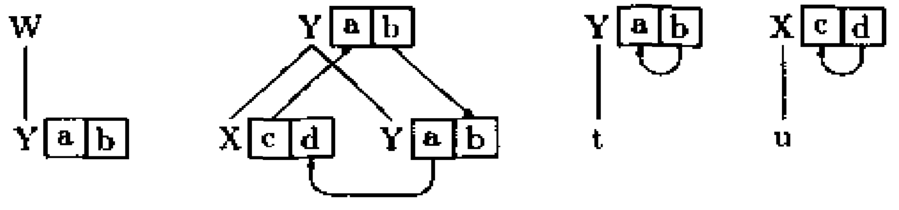
for Figure 1.2 are:

$$\begin{aligned}
D^0_{Y.b} &= \emptyset \\
D^1_{Y_1.a} &= \{X.c\} \\
D^1_{X.d} &= \{Y_2.a\} \\
D^1_{Y_2.b} &= \{Y_1.b\} \\
D^2_{Y.a} &= \{Y.b\} \\
D^3_{X.c} &= \{X.d\}
\end{aligned}$$

The set *of direct attribute dependencies* DDP(p) for a production p is

$$\{(X_j.b, X_i.a) \mid X_i \in p \wedge X_j.b \in D^p_{X_i.a}\}.$$

DDP(p) implicitly defines a graph; the nodes in this graph correspond to the attribute occurrences of symbols in production p, and the edges denote attribute dependencies. Figure 1.3 shows the DDP(p) and the graphs of DDP(p) for the AG of Figure 1.2. An attribute grammar is *locally acyclic* if the graph of DDP(p) is acyclic for each p ∈ P.



$$\begin{aligned}
\mathrm{DDP}(0) &= \{\} \\
\mathrm{DDP}(1) &= \{(X.c, Y_1.a), (Y_2.a, X.d), (Y_1.b, Y_2.b)\} \\
\mathrm{DDP}(2) &= \{(Y.b, Y.a)\} \\
\mathrm{DDP}(3) &= \{(X.d, X.c)\}
\end{aligned}$$

Figure 1.3: Graphs of DDP(p) for AG of Figure 1.2

If S is a derivation tree corresponding to a sentence in L($\mathcal{G}$), *DT(S)* is defined as the superimposition of DDP(p) for all applications of any p in S. An AG is *well-defined* iff it is complete and the graph DT(S) is acyclic for each parse tree corresponding to a sentence of $\mathbf{L}(\mathcal{G})$. Consider the sentence S = *uut* and the AG of Figure 1.2. The parse tree of S and the (acyclic) graph of DT(S) are shown in Figure 1.4. Note how the inherited attributes pass information towards the root of the tree, while the synthesized ones move information towards the leaves.

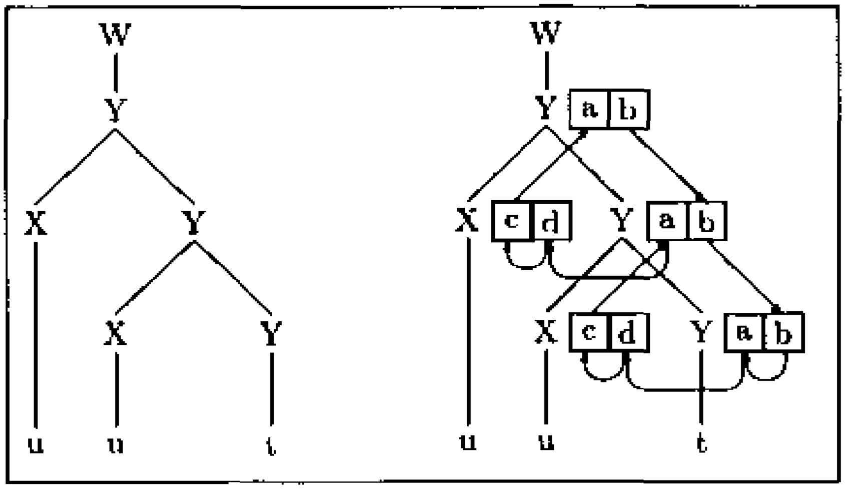


Figure 1.4: Parse Tree and DT(S) for the string "uut" using the grammar of Figure 1.2.

Knuth[13] developed an algorithm which tests whether the graphs of all DT(S) are acyclic for a particular AG. Jazayeri[14,15] has shown that Knuth's algorithm has exponential time complexity with respect to the size of the AG being analyzed, and further that the problem of determining whether all parse trees are acyclic (the circularity test) is inherently exponential. The proof of this complexity involves simulating a linear bounded automaton (lba) using attribute grammars, and thus reducing the circularity test to the lba membership problem, which is known to be exponential.

[13] Knuth, 1968 and Knuth, 1971.

[14] Jazayeri, Ogden, and Rounds.

[15] Jazayeri.

If it is known that an AG is well-defined, then a simple non-deterministic evaluation algorithm (evaluator) can be used to assign values to all of the attributes in the parse tree corresponding to a sentence in the AG. The algorithm walks the tree in a random fashion. Each time a node is visited, all attributes of this node whose semantic rules may be evaluated (those whose arguments are already defined) are assigned values according to the value returned by their semantic functions. The algorithm terminates when all attributes in the parse tree have been defined.[16]

This method of attribute evaluation has two major drawbacks. Because the algorithm is non-deterministic, much time can be wasted visiting nodes whose attributes have all already been defined or whose undefined attributes are still not ready to be evaluated. Another serious drawback is that the circularity test must be applied to the grammar to show that it is well-defined. Since this test has exponential time complexity with respect to the size of the grammar, it is impractical for grammars of the size necessary to define most programming languages.

Because the non-deterministic approaches to attribute evaluation are so inefficient, several researchers have proposed deterministic methods for attribute evaluation.[17,18,19,20] Each of these methods imposes restrictions on the types of attributes or the dependencies between attributes in the AG. Even the most restricted of these methods allows synthesized attributes. Knuth has shown that "synthesized attributes alone are sufficient to define the meaning associated with any derivation tree."[21] His argument is mainly a theoretical one, however, since

[16] Kennedy and Warren, pages 33-34.

[17] Bochmann.

[18] Jazayeri and Walter.

[19] Kastens.

[20] Kennedy and Warren.

[21] Knuth, 1968, page 134.

it involves synthesizing all the information in the tree up to the root, and then applying a single function which specifies the meaning of the tree. Obviously, an AG with a single defining function at the root doing all the work is not practical as a method of semantic specification. It does show, though, that it is possible to define the semantics of any derivation tree using any of the restricted types of AGs.

Deterministic evaluation techniques for subclasses of AGs may be divided into those that rely on a predetermined order for visiting nodes in a derivation tree and those that define a visit sequence uniquely for each AG or each derivation tree. Naturally, the methods that use a predetermined visit sequence are more restrictive than those that do not, but the evaluators are also easier to implement, analyze, and optimize. The most general of these methods is the *Alternating Semantic Evaluator* (ASE).[22] For practical problems (including most programming languages) the restrictions imposed by the ASE do not hinder the AG writer. This research focuses on ASE attribute grammars. Kastens[23] provides a complete characterization of the various classes of AGs.

The Alternating Semantic Evaluator derives its name from the passes it makes over the semantic tree alternating first from left to right and then from right to left. Each pass is a modified preorder traversal of the tree, originally described by Bochmann.[24] The recursive algorithm for a left to right pass performs the following actions.

1. Determine the production $p : X_0 \rightarrow X_1 X_2 \ldots X_n$ that applies at the node.

2. For each node $X_i : 1 \leq i \leq n$, in sequence, beginning with X_1, if X_i is a nonterminal,

[22] Jazayeri and Walter.
[23] Kastens, pages 242–244.
[24] Bochmann.

   (a) Evaluate a maximal subset of $AI(X_i)$ according to the defining functions for these attributes.

   (b) Invoke the algorithm recursively on X_i.

3. Evaluate a maximal subset of $AS(X_0)$ according to the defining functions for these attributes.

For a right to left pass, the only difference is that the nodes $X_1 \ldots X_n$ are treated in the reverse order, node X_n first. Note that each time a node is visited, a subset of its inherited or synthesized attributes is evaluated. Which attributes are evaluated on each pass and the number of passes required for attribute evaluation (if the AG is ASE at all) are established through examination of the AG at compiler generation time.

Before describing the ASE membership algorithm, the binary relation on attributes, β, must be defined. Consider a production $p : X_0 \rightarrow X_1 \ldots X_n$ and attribute occurrences a and $b \in A(p)$.

- For $a \in AS(X_0)$,
  $\beta(a, b) = b \notin AS(X_0)$.

- For $a \in A(X_i), 1 \leq i \leq n$,

  For a left to right pass,
  $\beta(a, b) = b \notin \{AS(X_0) \cup (\bigcup_{k=i}^{n} A(X_k)\}$.

  For right to left pass,
  $\beta(a, b) = b \notin \{AS(X_0) \cup (\bigcup_{k=1}^{i} A(X_k)\}$.

- For $a \in AI(X_0)$,
  $\beta(a, b)$ is not defined.

The ASE membership algorithm determines, independent of any particular parse tree, which attributes can be evaluated during each pass. The first pass is a left to right pass. For each pass, assume initially that all attributes not yet assigned to an earlier pass can be evaluated in this pass. Assign this set of attributes as the value of A_m, where m is the current pass number. For each production $p : X_0 \rightarrow X_1 \ldots X_n$, for each attribute $a \in \{AS(X_0) \cup (\bigcup_{i=1}^{n} AI(X_i))\}$, for each $b \in D^p_{X_i.a}$, if b was not assigned to an earlier pass and $\beta(a, b)$ is false, delete a from A_m. Continue examining each $a \in A_m$ until no further deletions are possible. The attributes remaining in A_m are those that can be evaluated during the mth pass. The algorithm terminates when either

1. no deletions were made during the test of the last pass, in which case the AG is evaluable in m alternating passes and the sets A_m define which attributes will be evaluated on each pass, or

2. A_m and A_{m-1} are both empty, in which case the AG is not ASE evaluable.

The AG of Figure 1.2 is evaluable in two alternating passes.

$$A_1 = \{Y.b\} \text{ and } A_2 = \{Y.a, X.c, X.d\}.$$

Kastens[25] provides an example of a non-ASE attribute grammar.

Note that once we have determined that an AG is ASE, we have a deterministic method for evaluating the attributes in any semantic tree corresponding to a string in the language of the AG. The membership algorithm will detect circularities in the AG, and does so much more efficiently than the exponential algorithm of Knuth (although it also rejects some AGs that are non-circular).

The elimination of the circularity test and the deterministic nature of the resulting attribution algorithm, make ASE attribute grammars a useful method

[25] Kastens, pages 242-244.

for automatically including semantics in a compiler. ASE attribute grammars, however, are often not space efficient and the AG constructs do not always afford a natural description of the semantics of a language. In particular, the author of an AG specification is burdened with the task of introducing attributes and attribution rules that simply copy an attribute value from one tree location to another. *Tree Manipulation Languages* (TMLs) are introduced in the next section as a natural alternative to AGs.

1.3 Tree Manipulation Languages

Tree Manipulation Languages are designed to operate on trees, providing operations to construct, transform, traverse, and annotate them. Current research involves the use of TMLs in syntax directed editing and as a parser generator interface. Input to a TML interpreter consists of a TML program, the tree(s) to be operated on, and the grammar used to construct the tree. Analysis of the grammar and TML program allows the TML interpreter to attain a high level of efficiency. It is able to determine exactly which portions of a tree will be referenced, and can therefore allocate storage for and visit only critical portions of the tree, saving both time and space as actual tree manipulations are carried out. It may also be possible for the TML interpreter to determine which actions of the program can be done in parallel. All of these optimizations result from the constrained domain of TMLs. We will show that in spite of their limited domain, TMLs are a natural and powerful medium for expressing language semantics.

TMLs are procedural, whereas AGs are declarative. Programmers are more accustomed to writing in procedural languages. Therefore, they are likely to find TMLs more natural to use than AGs. Morell[26] notes that "by returning explicit flow-of-control to the programmer, {a TML . . . } enhances verification and ex-

[26] Morell, page 15.

pression of the programmer's intent." Strict AGs require a single assignment to each attribute. TMLs allow multiple assignments to the same attribute. TMLs also provide notations that allow references to non-local attributes. Recall that AGs restrict attribute references to those occurring in the same production as the attribute occurrence being defined. Common programming language semantic actions, such as symbol table construction and typing of expressions become simpler to specify when multiple assignments and remote references are available to the writer of the semantic specification. An example showing symbol table construction using TMLs appears in Chapter 3 after we have described TreeSem, the particular TML used in this dissertation.

1.4 Related Work

Current research in the area of semantic specification for automatic compiler generators focuses primarily on methods to increase the time and space efficiency of attribute grammars. An obvious strategy used to save space during attribute evaluation is to use pointers to large attributes rather than maintaining separate copies of them for each attribute occurrence.[27] The systems GAG and LINGUIST-86 compare the lifetimes of the attributes in order to allocate storage for them as either a global variable or as a global stack.[28] Raiha is able to replace certain chains of local attribute references with upward remote references, thus eliminating the storage required for the intermediate attributes in the chain.[29] The time required for attribute evaluation can be descreased by identifying passes or portions of subtrees where no significant computation will be done, and skipping the pass or visit to the subtree. Affix grammars, which resemble attribute grammars, have also been investigated as a method of

[27] Bochmann, page 01.

[28] Farrow and Yellin, page 396.

[29] Raiha and Tarhio.

semantic specification. Other formalisms for semantic specification appear more useful for proving properties of programs.

As mentioned in Section 1.3, current TML research includes the use of TMLs in syntax directed editing[30] and as a parser generator interface. A primary focus of the latter research is on developing an optimal evaluation strategy for a TML.[31] Tree transformation rules have also been investigated as an extension to conventional attribute grammars.[32]

1.5 Research Goals

Automatic compiler generation tools allow a compiler to be automatically generated from a specification of the language. Although these compilers are said to be generated "automatically" from the specification, the specification itself must still be written by hand. Most of the current research in the area of semantic specification is concerned with improving the efficiency of the compiler produced, and does little to address the complexities of writing a semantic description of a language.

The goal of this research is to develop a specification language that retains the efficiency characteristics and specification power of existing methods, yet is easier to use and understand. TMLs appear to be readily adaptable to this task.

1.6 Remainder of Thesis

The remaining chapters of this thesis are devoted to TreeSem, a Tree Manipulation Language designed to be used as a semantic specification language. The following topics are addressed:

[30] Donzeau-Gouge, Huet, Kahn, and Lang.
[31] Morell.
[32] Alblas.

1. The syntax and semantics of TreeSem are described in detail.

2. Examples show how TreeSem is used to describe language semantics. A comparison of semantic specifications using conventional attribute grammars and using TreeSem illustrates the advantages of using TreeSem.

3. Translations from a TreeSem specification into an equivalent ASE attribute grammar specification and from an attribute grammar to TreeSem are presented.

4. Proof is given that these translations yield equivalent specifications. The dual translations demonstrate that TreeSem has the same specification power as ASE attribute grammars (and by Knuth's argument, all classes of AGs).

5. Enhancements to TreeSem and conclusions drawn from this research are presented.

Chapter 2

Description of TreeSem

A TreeSem program consists of three main parts: DECLS (attribute type declarations), GRAMMAR and TRAVSEQ (traversal sequence). DECLS contains the type declarations for the attributes of the input tree. The GRAMMAR provides the underlying grammar for the derivation tree input to the TreeSem interpreter. TRAVSEQ specifies the actions to be applied to the derivation tree. An LR grammar describing the complete syntax of TreeSem is given in Appendix A. The following sections detail the syntax and semantics of TreeSem.

2.1 Attribute Declaration Section

$$\begin{array}{lcl}
\text{DECLS} & \rightarrow & \\
 & \mid & \text{attribute types DECL\_LIST} \\
\text{DECL\_LIST} & \rightarrow & \text{DECL} \\
 & \mid & \text{DECL\_LIST DECL} \\
\text{DECL} & \rightarrow & \langle \text{id\_sym} \rangle = \langle \text{id\_sym} \rangle \; ;
\end{array}$$

The DECLS section consists of the keywords "attribute types" followed by a sequence of type declarations for the attribute names appearing in the input derivation tree. Each declaration consists of an attribute name followed by its type. For example,

```
attribute types
  length      =  integer;
  spelling    =  string;
```

declares the attribute "length" to be of type *integer*, and the attribute "spelling" to be of type *string*. Some applications of TreeSem may not require a DECLS section, and it may be empty.

2.2 Grammar Section

```
GRAMMAR    →  grammar PRODLIST end_grammar
PRODLIST   →  PROD
           |  PRODLIST PROD
PROD       →  LHS →RHSLIST ⟨end_of_line⟩
LHS        →  ⟨id_sym⟩
RHSLIST    →
           |  ⟨id_sym⟩ RHSLIST
```

The GRAMMAR section of a TreeSem program is a BNF grammar describing the syntax underlying the input tree, delimited by the keywords "grammar" and "end_grammar." This grammar is not constrained to be the actual syntax of the language being translated. It may describe the abstract syntax of the language or any other syntax the TreeSem programmer wishes. This feature will become especially useful when research proceeds to the point that a TML program can be analyzed to determine what portions of the input are referenced, thus directing parse tree construction accordingly.

Even if the grammar were the actual grammar of the source language, the GRAMMAR section is still not redundant. In an attribute grammar, rules are associated with each production in the underlying context free grammar, and each grammatical production must appear in the attribute grammar. As will be explained in the next section, a TreeSem program does not require all productions in the underlying grammar to be mentioned explicitly, yet algorithms that translate or interpret TreeSem programs must be aware of the entire grammar underlying the tree.

2.3 Traversal Section

The operations that are to be applied to the input tree described by the DECLS and GRAMMAR sections of a program are specified in the TRAVSEQ portion of the program. This specification determines the control flow and attribute assignments that are to occur. Attribute grammars specify only attribute assignments, allowing the evaluator to determine their order of execution.

2.3.1 Control Flow

The control flow statements are of two types: those that determine the order in which the nodes in the input tree are visited and those that specify when an attribute assignment is executed.

$$\begin{array}{lcl} \text{TRAVSEQ} & \rightarrow & \text{TRAVREV} \\ & | & \text{TRAVSEQ TRAVREV} \\ \text{TRAVREV} & \rightarrow & \text{TRAVERSAL} \\ & | & \text{REVERSAL} \end{array}$$

These TRAVERSALs and REVERSALs are executed sequentially.

2.3.1.1 TRAVERSAL

```
TRAVERSAL  →  traverse ORDER GUARD_LIST end_traverse
ORDER      →  inorder
           |  preorder
           |  postorder
```

A TRAVERSAL groups the statements that are to be executed during a single pass over the input tree, in a manner analagous to a begin/end block. ORDER specifies the traversal path to be taken as the program executes. ORDER may be specified as either "inorder," "preorder," or "postorder," corresponding to the usual tree traversals. The only statements currently allowed inside a traversal are guarded statements, and these specify exactly when each assignment is to be executed, so the three traversal orders are essentially equivalent. This will become evident after guarded statements have been introduced. While ORDER currently has no effect, it is included so that statements without explicit execution-time control may be easily added in the future.

2.3.1.2 REVERSAL

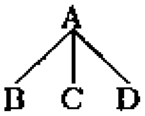
```
REVERSAL  →  reverse ;
```

A REVERSAL has the effect of reversing the order in which child nodes are visited during traversals. Initially, traversals visit the children of a node in a left to right order. After a "reverse" statement, traversals will visit the children in a right to left order, until the next REVERSAL occurs, and the initial order is resumed. An alternative way to think of a REVERSAL is that it rotates the tree about its "trunk" 180 degrees. The only problem with this conceptualization is that *guards* (explained next) always match the original orientation of the tree.

2.3.1.3 GUARD

$$\begin{array}{lcl} \text{GUARD\_LIST} & \rightarrow & \text{GUARD\_STMT} \\ & | & \text{GUARD\_LIST GUARD\_STMT} \\ \text{GUARD\_STMT} & \rightarrow & \text{GUARD} \Rightarrow \text{SEM\_LIST} \\ \text{GUARD} & \rightarrow & \text{SQUARE\_TREE} \\ \text{SQUARE\_TREE} & \rightarrow & [\ \text{IDLIST}\ ] \end{array}$$

A TreeSem GUARD represents a subtree which is matched against the input tree to determine which actions to apply. The notation used for trees is similar to that of LISP. A tree consists of a sequence of identifiers enclosed in square braces and separated by commas. (Examination of the complete syntax for an IDLIST shows that it is more general than this, but TreeSem restricts the form of an IDLIST that may be used as a guard.) The root of the tree is the first element of the list and the remaining elements of the list are the children from left to right. Each guard in a TreeSem program must correspond to one of the productions given in the grammar section. For example, the guard

[A, B, C, D] represents the subtree 

and corresponds to the production **A →B C D**. During a traversal, as each node is visited, the immediate subtree with that node as the root is compared with the GUARDs of the GUARD_LIST contained in that traversal. If a match is found, the SEM_LIST following that guard is applied. If the same guard appears more than once in a GUARD_LIST, the effect is the same as if there were a single guard followed by the concatenation of the SEM_LISTs from each guard. Thus, a GUARD_LIST containing

$$\begin{array}{lcl} \text{GUARD}_1 & \Rightarrow & \text{SEM\_LIST}_1 \\ \text{GUARD}_1 & \Rightarrow & \text{SEM\_LIST}_2 \end{array}$$

is equivalent to the GUARD_LIST

$$\begin{aligned} \text{GUARD}_1 \;\Rightarrow\; & \text{SEM\_LIST}_1 \\ & \text{SEM\_LIST}_2. \end{aligned}$$

The translations and proofs of the following chapters assume (without loss of generality) that all the guards of a single GUARD_LIST are unique.

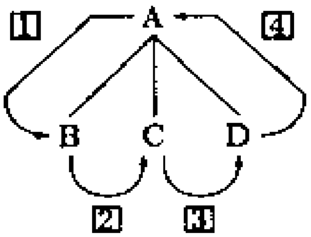
2.3.1.4 SEM_LIST

```
SEM_LIST       →  WHEN_ASGN
               |  SEM_LIST WHEN_ASGN
WHEN_ASGN      →  WHEN ASGN_LIST
WHEN           →  @ ID_SYM_PLUS :
ID_SYM_PLUS    →  ⟨id_sym⟩ ( ⟨number⟩ )
               |  ⟨id_sym⟩
```

A SEM_LIST is a sequence of WHEN_ASGNs. Each WHEN_ASGN begins with a WHEN that indicates the exact point, in the traversal of the subtree described by the preceding guard, that the assignments in the ASGN_LIST following it are to be executed. The ID_SYM_PLUS of a WHEN indicates one of the nodes from the current subtree matched by the preceding guard. The ⟨id_sym⟩ of an ID_SYM_PLUS is the name of the node. The ⟨number⟩ is used to distinguish multiple occurrences of the same node name appearing in a single guard. Node names are numbered sequentially, from left to right, beginning with '1'. If no ⟨number⟩ is specified, it is assumed to be '1'.

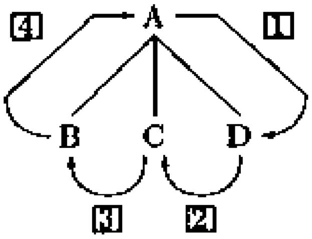
For a guard consisting of n components, there are n points in the traversal of the matched subtree where assignments may be applied. These points correspond to the arcs in the traversal path that connect the nodes of the subtree. Upon encountering the node indicated by ID_SYM_PLUS, the ASGN_LIST is executed. Thus, "@A: B.x ← C.y" may be read as "after encountering node A, execute the assignment B.x ← C.y." Consider, again, the guard [A, B, C, D]. If an even number of REVERSAL statements have previously been executed, the assignment locations and corresponding WHENs are:

point	WHEN
1	@A:
2	@B:
3	@C:
4	@D:



If an odd number of REVERSAL statements have previously been executed, the assignments and WHENs are:

point	WHEN
1	@A:
2	@D:
3	@C:
4	@B:



2.3.2 Attribute Assignments

Attribute assignments are used to assign or change the values of attribute occurrences in the input tree. The value assigned to an attribute is determined as a function of attribute values occurring in the tree. TreeSem expands the scope of allowable references beyond those allowed by attribute grammars. It also allows multiple assignments to the same attribute occurrence.

2.3.2.1 Regular Assignments

ASSIGNMENT → DIRTREE ← TREE ;

The simplest form of attribute assignment occurs when DIRTREE and TREE are both attribute occurrences of symbols in the same production; the production indicated by the preceding GUARD. The effect of such an assignment is to assign the *current* value of the attribute referenced on the right hand side of the production as the value of the attribute on the left hand side of the production. The types of the two attribute occurrences must be compatible. Thus, if the guard in the current context is [A, B, A, C], the assignment

A(2).x ←B.y

assigns the value of the y attribute of the B node as the value of the x attribute of the second A node. All other attribute values are unaffected. For purposes of uniformity, we may assume that an identity function is being applied to the value of the right hand side attribute.

2.3.2.2 Auxillary Functions

```
TREE        →  DIRTREE
            |  FN_CALL
FN_CALL     →  fn_sym ⟨id_sym⟩ OPTARGLIST
OPTARGLIST  →
            |  ( ARGLIST )
ARGLIST     →  TREE
            |  TREE , ARGLIST
```

A FN_CALL invokes an externally defined function to return an attribute value. The arguments of a function consist of a possibly empty list of attribute references and function calls. These functions are assumed to return a single value and may not exhibit any side effects on the values of attributes appearing in the input tree. They may not reference attributes in the input tree other than through their argument list. When a FN_CALL appears on the right hand side of an assignment statement, the *current* values of its arguments are passed to the function and the result returned is assigned as the value of the attribute on the left hand side of the assignment. The function must return a value that is compatible with the type of the left hand side attribute occurrence.

2.3.2.3 Reassignment

TreeSem allows multiple assignments to the same attribute occurrence. In this respect, TreeSem attributes behave in a fashion similar to regular program *variables*. Recall that an attribute grammar requires exactly one assignment to each attribute occurrence if the grammar is to be well-defined. TreeSem is able to allow multiple assignments because the flow of control is explicitly defined by the programmer. In an attribute grammar, the flow of control is determined externally by the evaluator. When a reference is made to a TreeSem attribute, the value returned is the value most recently assigned to that attribute during execution of the TreeSem program. Consistent with the notion of multiple attribute assignments, TreeSem assumes that each attribute occurrence in the input tree has been assigned an initial value consistent with its type. This value may be a useful one, such as the spelling of an identifier determined by a scanner, or it may simply be a default initialization value. Thus all references to attribute values during execution of a TreeSem program will produce a defined result.

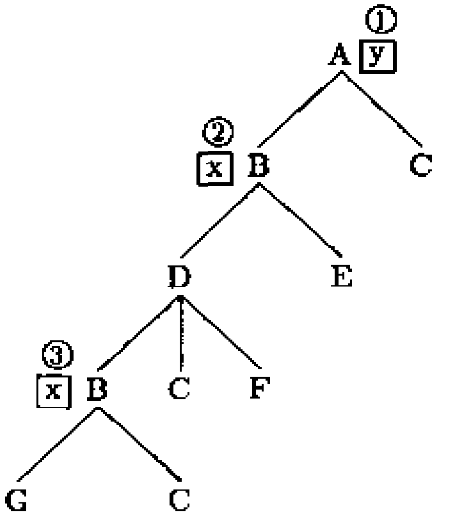
2.3.2.4 Upward Remote Attributes

```
DIRTREE      →  ↑ID_SYM_PLUS . (id_sym)
             |  DOWN_SPEC
             |  ID_SYM_PLUS . (id_sym)
DOWN_SPEC    →  DOWNID < SQUARE_TREE_LIST >
                SQUARE_TREE . (id_sym)
```

The scope of allowable attribute references and assignments is not limited to those of the current production as a traversal proceeds. TreeSem includes notations for attributes occurring either upward or downward from the current position in the tree. This eliminates the need for the programmer to introduce attributes and attribution rules that simply copy an attribute value from one tree location to another. An upward attribute is specified using an uparrow (↑) followed by the node name, a period, and the name of the attribute. An upward attribute specification always refers to the named attribute of the first instance of the node encountered on a path from the current traversal position to the root of the tree. Nodes in the current production are not included in this path. It is the specifier's responsibility to ensure that the named node always occurs on a path from the current traversal position to the root. Figure 2.1 illustrates the notation for upward remote attributes.

2.3.2.5 Downward Remote Attributes

The specification of a downward attribute is a bit more complex, as there is usually more than one downward path from the root node of the current position. Thus the specification for a downward attribute defines the path to be taken to arrive at the desired attribute's node from the current position in the tree.



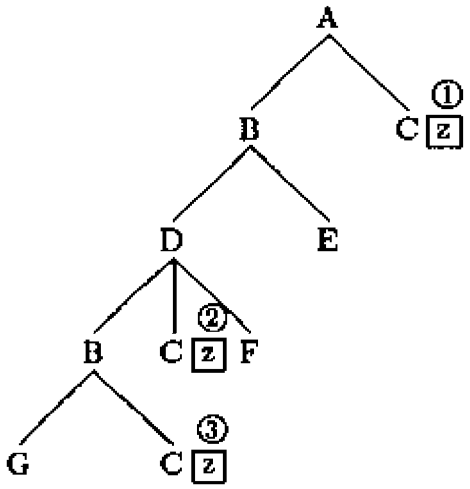
Guard	Notation	Local	Attribute Referenced
[B,G,C]	B.x	*	3
[B,G,C]	↑B.x		2
[B,G,C]	↑A.y		1
[D,B,C,F]	B.x	*	3
[D,B,C,F]	↑B.x		2
[D,B,C,F]	↑A.y		1
[B,D,E]	B.x	*	2
[B,D,E]	↑B.x		illegal
[B,D,E]	↑A.y		1
[A,B,C]	B.x	*	2
[A,B,C]	↑B.x		illegal
[A,B,C]	↑A.y		illegal

Figure 2.1: Upward Remote Reference Notation

The DOWNID is the the name of right hand side (child) symbol of the local production that occurs in the path to the desired attribute. The SQUARE_TREE representations of all of the production applications that will possibly be encountered along the path from the current production to the attribute are contained in the SQUARE_TREE_LIST. Each SQUARE_TREE must contain exactly one "#" annotation preceding the child symbol that is to be followed (along the path) whenever this production is encountered. The first element of a SQUARE_TREE is the root element, so it cannot be the annotated one. Any production occurring in this portion of the specification must occur on some possible path from the current traversal position to the desired attribute at least once. The order of productions in this list is not important. No production may appear more than once. The final production (the one indicated by the SQUARE_TREE in the top-level DOWN_SPEC expansion) must not appear in this list. The productions in the list may be encountered zero or more times along any *particular* path. The SQUARE_TREE portion of the DOWN_SPEC describes the production containing the actual attribute being referenced. The symbol that owns the attribute is marked with a "#". The final element of the specification, the ⟨id_sym⟩, contains the name of the attribute being referenced. As with upward references, it is the programmer's reponsibility to ensure that the desired attribute will always exist on the indicated path. Examples of downward remote attribute specifications appear in Figure 2.2.

2.4 Example Semantic Specifications

The following examples are presented to illustrate the use of TreeSem in specifying the semantics of a language. In each case, the corresponding ASE attribute grammar specification is included, so that the reader already familiar with attribute grammar notation will be able to more easily understand the TML nota-



Guard	Notation	Local	Attribute Ref'd
[B,G,C]	C.z	*	3
[D,B,C,F]	C.z	*	2
[D,B,C,F]	#B <> [B,G,#C].z		3
[B,D,E]	#D <[D,#B,C,F]> [B,G,#C].z		3
[B,D,E]	#D <> [D,B,#C,F].z		2
[A,B,C]	C.z	*	1
[A,B,C]	#B <[B,#D,E]> [D,B,#C,F].z		2
[A,B,C]	#B <[B,#D,E], [D,#B,C,F]> [B,G,#C].z		3
[A,B,C]	#B <[D,#B,C,F], [B,#D,E]> [B,G,#C].z		3

Figure 2.2: Downward Remote Reference Notation

tion, and also so that the two specification methods can be compared. Some of the examples were produced automatically using the translation algorithms presented in the next chapter. The DECLS section is not included in the TreeSem specifications.

This first example is an AG specification for the declarations of a block-structured language. It is adapted from Raiha.[1]

```
attribute types
   error = boolean;
   spelling = string;
   env = env_type;
   upd = env_type;

<program> ::= <block>
rules
     <block>.env := empty;
selur

<block> ::= <declist> <stmtlist>
rules
     <declist>.env := <block>.env;
     <stmtlist>.env := <declist>.upd;
selur

<stmtlist> ::= <stmtlist> <stmt>
rules
     <stmtlist>2.env := <stmtlist>.env;
     <stmt>.env := <stmtlist>.env;
selur

<stmtlist> ::= <stmt>
rules
     <stmt>.env := <stmtlist>.env;
selur

<stmt> ::= <id>
rules
     <stmt>.error := checkuse(<stmt>.env, <id>.spelling);
selur

<stmt> ::= <block>
rules
     <block>.env := <stmt>.env;
```

[1] Raiha and Tarhio, page 84.

```
selur

<declist> ::= <declist> <decl>
rules
     <declist>2.env := <declist>.env;
     <decl>.env := <declist>2.upd;
     <declist>.upd := <decl>.upd;
selur

<declist> ::= <decl>
rules
     <decl>.env := <declist>.env;
     <declist>.upd := <decl>.upd;
selur

<decl> ::= <id> <block>
rules
     <block>.env := procdecl(<decl>.env, <id>.spelling);
     <decl>.upd := procdecl(<decl>.env, <id>.spelling);
selur

<decl> ::= <vardecl>
rules
     <vardecl>.env := <decl>.env;
     <decl>.upd := <vardecl>.upd;
selur

<vardecl> ::= <id> <comma> <vardecl>
rules
     <vardecl>.upd := makevardecl(<vardecl>2.upd, <id>.spelling,
                       <vardecl>2.spelling);
     <vardecl>.spelling := <vardecl>2.spelling;
     <vardecl>2.env := <vardecl>.env;
selur

<vardecl> ::= <id> <colon> <typeid>
rules
     <vardecl>.spelling := <typeid>.spelling;
     <vardecl>.upd      := makevardecl(<vardecl>.env, <id>.spelling,
                           <typeid>.spelling);
selur
```

This is a straight translation of the above AG specification for declarations of a block-structured language into TreeSem syntax.[2]

[2] In order for these examples to be used as input to the translation programs that implement

```
grammar
   vardecl       ::= id colon typeid
   vardecl       ::= id comma vardecl
   decl          ::= vardecl
   decl          ::= id block
   declist       ::= decl
   declist       ::= declist decl
   stmt          ::= block
   stmt          ::= id
   stmtlist      ::= stmt
   stmtlist      ::= stmtlist stmt
   block         ::= declist stmtlist
   program       ::= block
end_grammar

traverse preorder
   [program, block] -->
      @ program :
         block.env := fn empty;

   [block, declist, stmtlist] -->
      @ block :
         declist.env := block.env;
      @ declist :
         stmtlist.env := declist.upd;

   [declist, declist, decl] -->
      @ declist :
         declist(2).env := declist.env;
      @ declist(2) :
         decl.env := declist(2).upd;
      @ decl :
         declist.upd := decl.upd;

   [declist, decl] -->
      @ declist :
         decl.env := declist.env;
      @ decl :
         declist.upd := decl.upd;

   [decl, id, block] -->
      @ id :
         block.env := fn procdecl(decl.env, id.spelling);
      @ block :
         decl.upd := fn procdecl(decl.env, id.spelling);
```

the algorithms of Chapter 3, several syntactic symbols have been replaced with ascii aproximations: ::= replaces →, := replaces ←, --> replaces ⇒.

```
   [decl, vardecl] -->
      @ decl :
         vardecl.env := decl.env;
      @ vardecl :
         decl.upd := vardecl.upd;

   [vardecl, id, comma, vardecl] -->
      @ comma :
         vardecl(2).env := vardecl.env;
      @ vardecl(2) :
         vardecl.upd := fn makevardecl(vardecl(2).upd,
                           id.spelling, vardecl(2).spelling);
         vardecl.spelling := vardecl(2).spelling;

   [vardecl, id, colon, typeid] -->
      @ typeid :
         vardecl.upd := fn makevardecl(vardecl.env,
                           id.spelling, typeid.spelling);
         vardecl.spelling := typeid.spelling;
   [stmtlist, stmtlist, stmt] -->
      @ stmtlist :
         stmtlist(2).env := stmtlist.env;
      @ stmtlist(2) :
         stmt.env := stmtlist.env;

   [stmtlist, stmt] -->
      @ stmtlist :
         stmt.env := stmtlist.env;

   [stmt, id] -->
      @ id :
         stmt.error := fn checkuse(stmt.env, id.spelling);

   [stmt, block] -->
      @ stmt :
         block.env := stmt.env;

end_traverse
```

The next example shows how the remote references of TreeSem are used to eliminate the need for copy rules in the specification. Upward references and assignments are made to the "env" attribute of block, and the types of variables are obtained through a downward reference.

```
grammar
   program      ::= block
   stmtlist      ::= stmtlist stmt
   stmtlist      ::= stmt
   declist       ::= declist decl
   declist       ::= decl
   decl          ::= id block
   decl          ::= vardecl
   vardecl       ::= id comma vardecl
   vardecl       ::= id colon typeid
   block         ::= declist stmtlist
   stmt          ::= id
   stmt          ::= block
end_grammar

traverse preorder

   [program, block] -->
      @ program :
         block.env := fn empty;

   [decl, id, block] -->
      @ decl :
         block.env := fn procdecl(^block.env, id.spelling);
         ^block.env := block.env;

   [vardecl, id, comma, vardecl] -->
      @ vardecl(2) :
         ~block.env := fn makevardecl (^block.env, id.spelling,
                       #vardecl(2)<[vardecl,id,comma,#vardecl(2)]>
                       [vardecl,id,colon,#typeid].spelling);

   [vardecl, id, colon, typeid] -->
      @typeid :
         ~block.env := fn makevardecl (~block.env, id.spelling,
                       typeid.spelling);

   [stmt, id] -->
      @stmt :
         stmt.error := fn checkuse(^block.env, id.spelling);

   [stmt, block] -->
     @stmt :
         block.env := ^block.env;

end_traverse
```

The following example is the result of translating the previous TreeSem specification into an attribute grammar. The attributes decl.blockenv2 and declist.blockenv3 correspond to the decl.upd and declist.upd attributes of the original AG example. The oddly named "p8n3_p9n3_spelling1" attribute, as well as the numerical endings on the attribute names are a result of the translation process, and will be explained in the next chapter.

```
attribute types
   error1 = boolean;
   p8n3_p9n3_spelling1 = string;
   spelling0 = string;
   env1 = env_type;
   env2 = env_type;
   blockenv1 = env_type;
   blockenv2 = env_type;
   blockenv3 = env_type;

<program> ::= <block>
rules
     <block>.env1 := empty;
selur

<block> ::= <declist> <stmtlist>
rules
     <declist>.blockenv1 := <block>.env1;
     <block>.env2 := <declist>.blockenv3;
     <stmtlist>.blockenv1 := <declist>.blockenv3;
selur

<stmtlist> ::= <stmtlist> <stmt>
rules
     <stmtlist>2.blockenv1 := <stmtlist>.blockenv1;
     <stmt>.blockenv1 := <stmtlist>.blockenv1;
selur

<stmtlist> ::= <stmt>
rules
     <stmt>.blockenv1 := <stmtlist>.blockenv1;
selur

<stmt> ::= <id>
rules
     <stmt>.error1 := checkuse(<stmt>.blockenv1, <id>.spelling0);
selur

<stmt> ::= <block>
rules
     <block>.env1 := <stmt>.blockenv1;
selur
```

```
<declist> ::= <declist> <decl>
rules
     <declist>2.blockenv1 := <declist>.blockenv1;
     <declist>.blockenv2 := <declist>2.blockenv3;
     <decl>.blockenv1 := <declist>2.blockenv3;
     <declist>.blockenv3 := <decl>.blockenv2;
selur

<declist> ::= <decl>
rules
     <decl>.blockenv1 := <declist>.blockenv1;
     <declist>.blockenv3 := <decl>.blockenv2;
selur

<decl> ::= <id> <block>
rules
     <block>.env1 := procdecl(<decl>.blockenv1, <id>.spelling0);
     <decl>.blockenv2 := procdecl(<decl>.blockenv1, <id>.spelling0);
selur

<decl> ::= <vardecl>
rules
     <vardecl>.blockenv1 := <decl>.blockenv1;
     <decl>.blockenv2 := <vardecl>.blockenv3;
selur

<vardecl> ::= <id> <comma> <vardecl>
rules
     <vardecl>2.blockenv1 := <vardecl>.blockenv1;
     <vardecl>.blockenv2 := <vardecl>2.blockenv3;
     <vardecl>.blockenv3 := makevardecl(<vardecl>2.blockenv3,
                             <id>.spelling0,
                             <vardecl>2.p8n3_p9n3_spelling1);
     <vardecl>.p8n3_p9n3_spelling1 := <vardecl>2.p8n3_p9n3_spelling1;
selur

<vardecl> ::= <id> <colon> <typeid>
rules
     <vardecl>.blockenv3 := makevardecl(<vardecl>.blockenv1,
                             <id>.spelling0, <typeid>.spelling0);
     <vardecl>.p8n3_p9n3_spelling1 := <typeid>.spelling0;
selur
```

This example is an AG grammar taken from Waite and Goos.[3] The AG

[3] Waite and Goos, page 206.

specification is evaluable in a minimum of 2 alternating passes.

```
attribute types
   a = integer;
   b = integer;
   c = integer;
   d = integer;
   e = integer;
   f = integer;
   g = integer;

<Z> ::= <X>
rules
   <Z>.b := 1;
selur

<X> ::= <W> <X> <Y>
rules
   <X>.a := <W>.d;
   <X>.e := <Y>.g;
   <X>2.b := <X>.b;
   <W>.d  := <X>2.a;
   <X>.f  := <X>2.e;
selur

<X> ::= s
rules
   <X>.a := <X>.b;
   <X>.e := <X>.b;
selur

<W> ::= t
rules
   <W>.c := <W>.d;
selur

<Y> ::= u
rules
   <Y>.g := <Y>.f;
selur
```

The TreeSem specification below is a translation of the previous AG example. Note that two traversals result, corresponding to the two passes needed to evaluate the AG. The use of the *reverse* statement is also shown. This causes the second traversal to proceed from right to left, rather than left to right, as it visits the child nodes of each production.

```
grammar
   Z            ::= X
   X            ::= W X Y
   X            ::= s
   W            ::= t
   Y            ::= u
end_grammar

traverse preorder

   [Z, X] -->
      @ X :
         Z.b := fn 1;

   [X, W, X, Y] -->
      @ W :
         X(2).b := X.b;
      @ Y :
         X.f := X(2).e;
         X.e := Y.g;

   [X, s] -->
      @ s :
         X.e := X.b;

   [Y, u] -->
      @ u :
         Y.g := Y.f;

end_traverse

reverse;

traverse preorder

   [X, W, X, Y] -->
      @ W :
         X.a := W.d;
      @ X(2) :
         W.d := X(2).a;

   [X, s] -->
      @ s :
         X.a := X.b;

   [W, t] -->
      @ t :
         W.c := W.d;
```

```
end_traverse
```

Chapter 3

Translation Algorithms

This chapter presents a pair of algorithms that will translate a TreeSem program to an equivalent ASE attribute grammar specification, and will translate an ASE attribute grammar specification into an equivalent TreeSem program. The translation algorithms take advantage of the explicitly defined control flow in the TreeSem program and the known traversal strategy of the ASE evaluator to determine a mapping between the TreeSem and AG attribute assignments and references. Statistics on the implementations of these algorithms appear in Appendix B.

3.1 Translation from TreeSem to AG

This section describes the algorithm used to translate a TreeSem program into an equivalent ASE attribute grammar. The algorithm must remove remote attribute specifications and multiple attribute assignments, must assign the proper values to significant attributes, and it must ensure that the resulting AG specification satisfies the ASE restrictions.

3.1.1 Grammar, Declarations, and Reversals

As the grammar section of the TreeSem specification is read, it is stored in such a way as to facilitate the various types of access required by later portions of the translation algorithm. Each production in the grammar is numbered, starting with 0.

A simple list of attribute type declarations is created as these declarations are read. These stored types are used to determine the types of attributes for the AG specification.

A boolean *reversed* indicates whether an odd number of *reverse* statements have previously been encountered in the text of the TreeSem program. It is initialized to *false* and negated by each subsequent *reverse* statement.

3.1.2 Traversals

The TRAVERSALs of the TreeSem program define the actual assignment actions that are to be carried out on the input tree. As pointed out in Chapter 2, all traversal ORDERs are essentially equivalent, so this algorithm assumes the ORDER is always preorder. As each TRAVERSAL is read, the assignments contained in it are stored. At the end of the TRAVERSAL, several steps are performed to effect the translation of the TreeSem assignments into AG assignments.

The overall translation strategy is based on the fact that we can determine, for each possible attribute occurrence in an input tree, the maximum number of assignments to this attribute *before* its owning symbol is visited, *as* it is visited (in the root position), and *after* it is visited, regardless of the symbol's derivation. These assignment counts are used as the basis for establishing proper attribute assignments and references in the resulting AG specification.

3.1.2.1 Remote Attribute Translation

Remote attributes of a TreeSem program are translated to local attributes of the production indicated by the current GUARD. Upward attributes are changed to local attributes of the root symbol of the production. Downward attributes are changed to local attributes of the DOWNID symbol appearing in the downward attribute specification. When remote attributes are changed into local attributes, a new name is constructed for them so they do not conflict with existing attributes. The type of the local attribute is the same as the type of the remote attribute that generated it.

For upward remote attributes, the new attribute name is constructed as a concatenation of the original symbol name and attribute name. For example, if the specification ↑**x.val** appears in the context of the guard **[A,B,C]**, it will be replaced with **A.xval.** We will assume that names created in this manner will not conflict with existing attribute names.

For downward remote attributes, name construction for the new attributes is more complicated. The constructed names represent the path from the owning symbol of the attribute to the actual *goal attribute*, the attribute at the terminal end of the path. To construct an attribute name for a particular owning symbol:

1. Determine all the productions in the SQUARE_TREE_LIST and the final SQUARE_TREE of the downward specification that are *used* in any path from the symbol to the goal attribute, expanding only the marked symbol of each SQUARE_TREE_LIST production, and never expanding the final SQUARE_TREE production. (Although all productions must be used in some path from the point where the specification originally occurs, the translation process creates new assignments and uses the same downward specification in different contexts, so some productions may not be

reachable.)

2. Sort the productions that were *used* by their associated numbers.

3. The attribute name is made up of a sequence of

   p(production number)n(node number)_

   constructions. *Production number* refers to the associated number for each production and *node number* is the position of the marked right hand side symbol in the production, where the left hand side symbol is numbered 0. The initial elements of the sequence are derived from the sorted list obtained in step 2. These are followed by an element derived from the final production in the downward specification.

4. The name is terminated with the goal attribute name from the downward specification.

For example, assume the following CFG and the associated production numbers:

Production Number	Production
1	B → C U
2	F → G
3	A → B
4	C → D
5	D → E
6	G → C V
7	C → D A
8	D → D A

If the following reference were associated with [A,B],

$$\#B < [B, \#C, U][C, \#D][D, \#D, A][C, \#D, A] > [D, \#E].x$$

then the attribute name for the symbol B would be

p1n1_p4n1_p7n1_p8n1_p5n1_x,

the attribute name for the symbol D would be

p8n1_p5n1_x,

and at E the attribute name is simply

x.

Again, we assume that names created in this manner will not conflict with existing attribute names.

Although the method for constructing downward remote attribute names is complex, the resulting names have the advantage that, regardless of where a downward attribute specification initially occurs, the name generated for a particular attribute of any symbol along the path will be the same. This greatly simplifies the task of determining which downward assignments and references are affecting the same attribute in the original specification.

3.1.2.2 Explicit Assignments and Counts

Each assignment appearing in the body of a TRAVERSAL is termed an *explicit* assignment. For each TRAVERSAL, the *explicit assignment list* for each symbol of each grammar production is initially empty. Each explicit assignment, with remote attributes replaced by local ones, is added to the end of explicit assignment list of the symbol indicated by the most recent GUARD and WHEN encountered in the text of the program.

In order to compute the assignment counts mentioned above, it is necessary to categorize each assignment based on whether it is made before or after the traversal visits the node indicated by the lhs of the assignment. Thus, for each assignment, a Boolean BEFORE is determined based on the lhs of the assignment, the most recent WHEN, and the production indicated by the most recent

GUARD as follows:

1. If the symbol indicated by the WHEN is the lhs symbol of the production, BEFORE is *false*.
2. If WHEN indicates a rhs symbol in the production, let *when_index* be the number of symbols in the GUARD preceding the symbol indicated by the WHEN, and let *lhs_index* be the number of symbols in the GUARD preceding the symbol indicated by the lhs of the assignment.
   (a) If *reversed* is *true*, BEFORE = $(when\_index \leq lhs\_index)$.
   (b) If *reversed* is *false*, BEFORE = $(when\_index \geq lhs\_index)$.

For each attribute of each symbol of each production, there is a pair of counts, *assigned_up* and *assigned_down* that indicate the number of assignments made to this attribute during this traversal on an upward path and on a downward path respectively. These counts are set to 0 at the beginning of each traversal. Based on the value BEFORE , each explicit assignment increments one of these counts for the attribute indicated by the lhs of the assignment.

3.1.3 Implicit Assignments

As remote attribute specifications appearing in the explicit assignments are changed to local ones, it is necessary to add *implicit assignments*. These assignments copy attribute values upward or downward in the tree, linking locally generated attributes with their corresponding remote attributes.

3.1.3.1 Downward Reference Propagation

For each downward reference occurring in an explicit assignment, *implicit downward reference* assignments are introduced to copy the value of the referenced attribute upward to the point where the explicit assignment is applied. The value of BEFORE associated with each of these implicit assignments is the same. It is determined using the method of Section 3.1.2.2, considering the DOWN_SPEC symbol of the downward specification as the lhs assignment symbol. An assignment is generated for each production included in the SQUARE_TREE_LIST or the final SQUARE_TREE of the downward attribute specification being referenced, unless a duplicate implicit downward reference assignment with the same BEFORE value already exists for this TRAVERSAL. The lhs of each assignment is an attribute with the same downward specification as the remote attribute, associated with the root symbol of the production. The rhs of the assignment is the same attribute associated with the symbol that was marked for this production in the downward specification of the remote attribute. (Note that the same downward attribute specification will generate different attribute names when associated with different symbols, and that different downward specifications may generate the same attribute name for a given symbol.) Thus, when determining whether duplicate assignments exist, it is necessary to compare the *names* of attributes occurring in the assignments, rather than the specifications of those attributes. Each assignment that is added increments either the *assigned_up* count or *assigned_down* count of the lhs attribute of the assignment, depending on the value of BEFORE. For the grammar of Section 3.1.2.1, the guarded assignment

```
[A,B] ⇒
    @A: B.y ←  #B < [B, #C, U][C, #D][D, #D, A][C, #D, A] > [D, #E].x;
```

will generate the following implicit downward reference assignments for the productions appearing in the downward specification:

B → C U

```
    B.p1n1_p4n1_p7n1_p8n1_p5n1_x ← C.p4n1_p7n1_p8n1_p5n1_x;
C → D
    C.p4n1_p7n1_p8n1_p5n1_x ← D.p8n1_p5n1_x;
C → D A
    C.p4n1_p7n1_p8n1_p5n1_x ← D.p8n1_p5n1_x;
D → E
    D.p8n1_p5n1_x ← E.x;
D → D A
    D.p8n1_p5n1_x ← D(2).p8n1_p5n1_x;
```

Of course, the explicit assignment will have the remote attribute reference replaced with a local one:

```
A → B
    B.y ← B.p1n1_p4n1_p7n1_p8n1_p5n1_x;
```

3.1.3.2 Downward Assignment Propagation

Implicit downward assignment assignments are added for explicit assignments to downward remote attributes in a similar manner to the addition of implicit downward reference assignments. Since implicit downward assignments must move information downward in the tree, the left hand sides and right hand sides of the generated assignments are reversed from those generated for downward references; the lhs of the assignment will be an attribute of the marked symbol of the production, and the rhs will be an attribute of the root symbol of the production. The value of BEFORE associated with each of these implicit assignments is the BEFORE value computed for the explicit assignment containing the downward attribute specification. Considering again the grammar of Section 3.1.2.1, the guarded assignment

```
[A,B] ⇒
    @A: #B < [B, #C, U][C, #D][D, #D, A][C, #D, A] > [D, #E].x ← B.y;
```

will generate the following implicit downward assignment assignments for the productions appearing in the downward specification:

```
B → C U
    C.p4n1_p7n1_p8n1_p5n1_x ← B.p1n1_p4n1_p7n1_p8n1_p5n1_x;
C → D
    D.p8n1_p5n1_x ← C.p4n1_p7n1_p8n1_p5n1_x;
C → D A
    D.p8n1_p5n1_x ← C.p4n1_p7n1_p8n1_p5n1_x;
D → E
    E.x ← D.p8n1_p5n1_x;
D → D A
    D(2).p8n1_p5n1_x ← D.p8n1_p5n1_x;
```

And the explicit assignment becomes:

```
A → B
    B.p1n1_p4n1_p7n1_p8n1_p5n1_x ← B.y; .
```

3.1.3.3 Upward Reference Propagation

When introducing implicit upward reference assignments, the remote attribute specification does not indicate what productions are used in the path from the explicit assignment to the desired remote attribute. Thus the first step in generating implicit upward reference assignments is to determine all the upward paths in any possible derivation tree, from the root of the production guarding the explicit assignment to the owning symbol of the upward referenced attribute.

For each arc in any of these paths, an implicit upward reference assignment is added, unless an identical assignment already exists for this TRAVERSAL. The node of the ar closest to the root of the tree is termed the *root* symbol, and the other node is the *child* symbol. The production used to generated the arc is referred to as simply the *production*. It is possible that more than one arc of a given production could be used in the upward paths. In this case, multiple assignments are generated.

The lhs of the implicit assignment generated for an arc is an attribute with the same specification as the upward remote attribute, but belonging to the child symbol. The rhs of the assignment is an attribute with the same specification as the upward remote attribute, belonging to the root node. Name translation for upward attributes is used to generate the actual names for these attributes. Note that the root symbol of the uppermost arc in each path is the actual upward remote attribute symbol, so the attribute name generated is the local attribute name. The BEFORE value for all implicit upward reference assignments is *false*, so the *assigned_down* count for the lhs attribute is incremented for each assignment generated.

For example, consider the grammar

$$\begin{array}{lcl} A & \rightarrow & B\ C\ B \\ B & \rightarrow & D \\ D & \rightarrow & E\ D \\ D & \rightarrow & E \quad . \end{array}$$

The guarded command,

[D,E] ⇒

  @D: E.y ← ↑A.x;

generates the implicit upward reference assignments:

```
A → B C B
     B.ax ← A.x;
     B(2).ax ← A.x;
B → D
     D.ax ← B.ax;
D → E D
     D(2).ax ← D.ax; .
```

The explicit assignment is changed to

```
D → E
     E.y ← D.ax; .
```

3.1.3.4 Upward Assignment Propagation

Implicit upward assignment assignments are generated in a similar manner to implicit upward reference assignments. The only difference is that the lhs and rhs of assignments are reversed. The BEFORE value for all generated assignments is *false*, and the *assigned_down* counts are incremented. Using the above grammar, the guarded assignment,

```
[D,E] ⇒
    @D: ↑A.y ← E.x;
```

generates the implicit upward assignment assignments:

```
A → B C B
     A.y ← B.ay;
     A.y ← B(2).ay;
B → D
     B.ay ← D.ay;
```

D → E D

  D.ay ← D(2).ay; .

The explicit assignment is changed to

D → E

  D.ay ← E.x; .

Notice that the attribute A.y is assigned twice in the first production, a situation not allowed in AGs. This problem is resolved later in the translation process.

3.1.4 Catchup Assignments

Because of the context-free nature of the grammar underlying a TreeSem input tree, implicit assignments that are really only applicable to some instances of a symbol or production must be applied to all of them. The problem that results from this is that an attempt to reference attribute values that are not properly in place may occur. Examples of this type of problem appear in the following sections. The solution to this problem is to identify the attributes that will be referenced, and to copy the old value of the assigned attribute to the referenced attribute. That way, when the assignment is made, the value of the attribute that was not supposed to be changed will remain the same. Since these types of reference problems result from upward and downward attribute assignments, the assignments added here will be called ***upward catchup*** and ***downward catchup*** assignments.

Another situation that causes reference problems results from an attribute numbering scheme that will be discussed later. Assignments added to correct these problems are termed ***explicit catchup*** assignments.

3.1.4.1 Downward Catchups

Downward catchup assignments are necessary when an implicit downward assignment has been added for an attribute x of a production p, but there is some production q that derives the root symbol of p, yet makes no assignment, with the same BEFORE value as that of the implicit downward assignment, to the x attribute of that symbol. Figure 3.1 shows an example of this situation. The figure shows the flow of information in the parse tree corresponding to the string *DD* after implicit downward assignments have been added for the explicit assignment

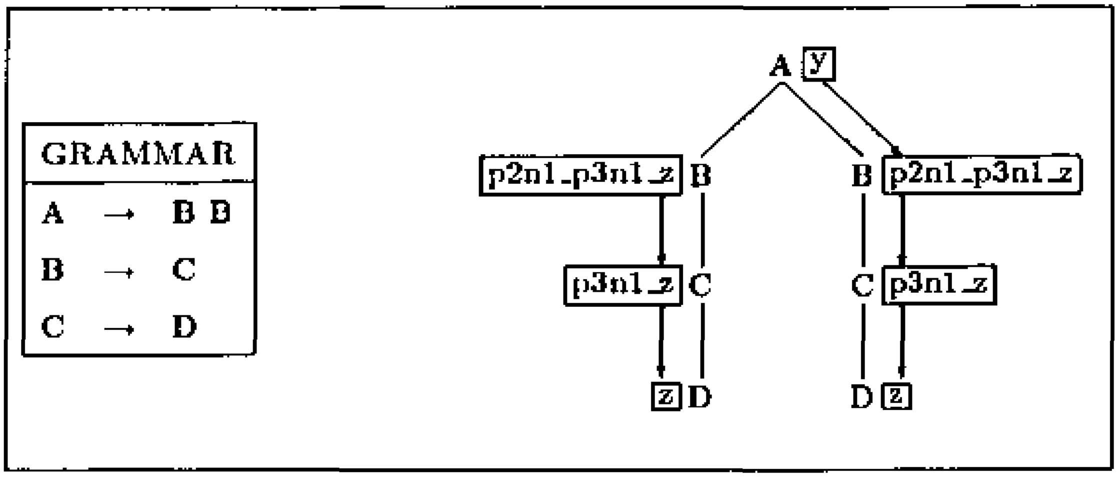


Figure 3.1: Downward Reference Anomaly

[A,B,B] ⇒

@A: #B(2) < B, #C > [C, #D].z ← A.y; ,

using the grammar shown. In this case, p is the production "B →C" and q is "A →B B." The attribute, x, that was added is *B.p2n1_p3n1_z*. There is no assignment to *B(1).p2n1_p3n1_z* in production q, yet it is referenced in production p.

The solution to this problem is to add an assignment in the context of production q that assigns the attribute x the old value of the attribute on the lhs of the original explicit downward assignment. The lhs of this assignment is the previously unassigned attribute. The rhs of this assignment is a downward reference to the attribute indicated by the original explicit downward assignment , but whose downward path specification originates at the symbol owning the "unassigned" x in production q. For our example, this adds the assignment

B(1).p2n1_p3n1_z ← #B(1) < B, #C > [C, #D].z;

in the context of production q. The BEFORE value for this assignment is the same as the BEFORE value for the implicit assignment in production p.

The WHEN symbol associated with this assignment is determined based on the current value of *reversed* and BEFORE . If BEFORE is false, WHEN is the root symbol of production q. Otherwise, if *reversed* is *true*, WHEN is the first rhs symbol of p, and if *reversed* is *false*, WHEN is the last rhs symbol of p. For the example in Figure 3.1, BEFORE is *false* and we will assume *reversed* is *false*, so WHEN is A. In all cases, the assignment is added to the *beginning* of the explicit assignment list for the WHEN symbol, and either *assigned_up* or *assigned_down* is incremented as appropriate.

The added assignment contains a downward reference. This assignment is subjected to downward name translation and implicit downward reference assignments are added. The AG assignments that result in the case of our example are

```
A → B B
        B.p2n1_p3n1_z ← B.p2n1_p3n1_z;
        B(2).p2n1_p3n1_z ← A.y;
B → C
```

C.p3n1_z ← B.p2n1_p3n1_z;

B.p2n1_p3n1_z ← C.p3n1_z;

C → D

D.z ← C.p3n1_z;

C.p3n1_z ← D.z;

and the flow of attribute values is shown in Figure 3.2. Solid arrows show assignments resulting from downward catchup analysis. Note that the AG assignments are not in their final form. In particular, we have the assignment *B.p2n1_p3n1_z* ← *B.p2n1_p3n1_z*, which is clearly not allowed in an AG.

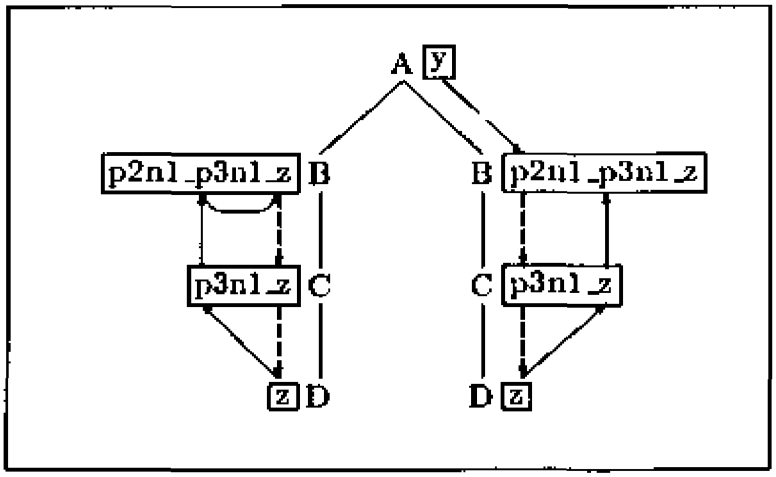


Figure 3.2: Downward Catchup Assignments for Figure 3.1.

3.1.4.2 Upward Catchups

Upward catchup assignments are necessary when an implicit upward assignment has been added for a rhs attribute *R.x* of a production *p*, but there is some production *q* with root symbol *R* that makes no assignment to *R.x*. Figure 3.3 shows an example of this situation. The figure shows the flow of information in

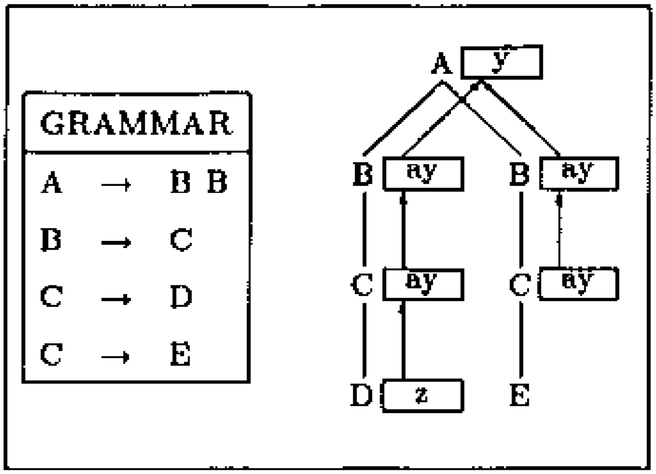


Figure 3.3: Upward Reference Anomaly

the parse tree corresponding to the string DE after implicit upward assignments have been added for the explicit assignment

[C,D] ⇒
    @D: ↑ A.y ← D.z; ,

using the grammar shown. In this case, p is the production "B →C" and q is "C →E." The attribute, x, that was added is ay. There is no assignment to $C.ay$ in production q, yet it is referenced in production p.

The solution to this problem is to add an assignment in the context of production q, that assigns the attribute occurrence $R.x$ the old value of the attribute on the lhs of the original explicit upward assignment. For our example, this adds the assignment

C.ay ← ↑ A.y;

in the context of production q. The BEFORE value for the added assignment is *true*. The WHEN symbol associated with this assignment is the root symbol, R, of production q. The assignment is added to the *end* of the explicit assignment list for the WHEN symbol, and *assigned_up* for $R.x$ is incremented.

The added assignment contains an upward reference. This assignment is subjected to upward name translation and implicit upward reference assignments are added. The AG assignments that result in the case of our example are

```
A → B B
      B.ay ← A.y;
      A.y ← B.ay;
      B(2).ay ← B.ay;
      A.y ← B(2).ay;
B → C
      C.ay ← B.ay;
      B.ay ← C.ay;
C → D
      C.ay ← D.z;
C → E
      C.ay ← C.ay;
```

and the flow of attribute values is shown in Figure 3.4. Solid arrows show assignments resulting from upward catchup analysis. Note once again that the AG assignments are not in their final form.

Another situation that causes reference problems results from an attribute numbering scheme that will be discussed later. Assignments added to correct these problems are termed *explicit catchup* assignments.

3.1.4.3 Explicit Catchups

An attribute numbering scheme that will be described in full in a later section requires explicit catchup assignments to be added. Explicit catchup assignments

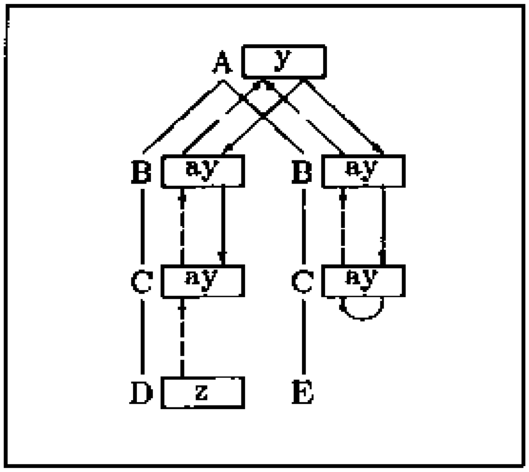


Figure 3.4: Upward Catchup Assignments for Figure 3.3.

are added after implicit assignments are changed to explicit ones, and ***maximum counts*** are determined. These two processes are described in the next two sections, but addition of explicit catchup assignments is discussed here because they relate to downward and upward catchup assignments. Explicit assignments are added in three different cases, related to the ***max_root***, ***max_before***, and ***max_after*** counts, which are described in Section 3.1.6.1.

1. For each attribute occurrence $L.x$, where L is the lhs symbol of production p, an explicit catchup assignment is added if there is no assignment made to $L.x$ in the context of p (***assigned_up*** and ***assigned_down*** are both zero), but the ***max_root*** count for this attribute is greater than zero. The assignment "$L.x \leftarrow L.x$" is added to the end of the explicit assignment list for the root symbol of p. BEFORE is arbitrarily chosen to be ***false*** for this assignment, and ***assigned_down*** for this attribute is incremented (to 1).

2. For each attribute occurrence $R.x$, where R is a rhs symbol of some production p, an explicit catchup assignment is added if the ***assigned_down***

count for $R.x$ is zero, but the ***max_before*** count for this attribute is greater than zero. The assignment "$R.x \leftarrow R.x$" is added to the end of the explicit assignment list for the root symbol of p. BEFORE is ***false*** for this assignment, and ***assigned_down*** for this attribute is incremented.

3. For each attribute occurrence $R.x$, where R is a rhs symbol of some production p, an explicit catchup assignment is added if the ***assigned_up*** count for $R.x$ is zero, but the ***max_after*** count for this attribute is greater than zero. The assignment "$R.x \leftarrow R.x$" is added to the end of the explicit assignment list for the symbol R in p. BEFORE is ***true*** for this assignment, and ***assigned_up*** for this attribute is incremented.

3.1.5 Convert Implicit Assignments

After all the implicit assignments have been generated, the majority of them are added to explicit assignment lists. Implicit downward references with ***false*** BEFORE values, and implicit downward assignments with ***true*** BEFORE values move information against the normal flow of a traversal. All other implicit assignments move information in a direction compatible with the normal flow of a traversal, so they are able to be considered as explicit assignments.

For each attribute x of each symbol S of each production p, implicit assignments are converted as follows:

1. If there is an **implicit downward assignment** to $S.x$, with a ***false*** BEFORE value, it is added to the ***beginning*** of the explicit assignment list for the lhs symbol of production p. This reflects the fact that the original explicit downward assignment generating this implicit assignment occurred at a point upward in the tree, and since BEFORE is ***false***, the downward assignment has already been applied. Therefore, the effect of the downward assignment should take place before any other assignments in the context

of production p. Recall that assignments associated with the root symbol of a production are always the first to be applied as a traversal visits the production.

2. If there is an **implicit upward assignment** referencing $S.x$, the implicit assignment is added to the *beginning* of the explicit assignment list for the symbol S. The remote upward assignment must occur at a point *below* S in the tree, so it is applied during the subtree visit to S. The effect of the explicit remote assignment should be realized as soon as the traversal returns from the subtree visit. Explicit assignments associated with a symbol are always executed immediately after the traversal returns from the subtree below that node.

3. If there is an **implicit downward reference** assignment that references $S.x$, with a *true* BEFORE value, it is added to the *end* of the explicit assignment list of the last node to be visited in p during this traversal. If *reversed*, this node is the first node on the rhs of p, and otherwise it is the last rhs node of p. The explicit assignment containing the downward reference that generated this implicit assignment occurs upward in the tree from p. Since it has a *true* BEFORE value, it has not yet been executed. The *most recent* value assigned to the downward attribute is propagated upward to the point of reference by executing the downward reference assignment just before the traversal leaves p. By making it the last assignment applied in the context of p, any assignments made to the actual remote attribute (which may be local to p) are reflected in the value that is passed upward.

4. If there is an **implicit upward reference** assignment to $S.x$, it is added to the *end* of the explicit assignment list for the symbol in production p that is visited just before S is visited.

- If *reversed*, this is the symbol following S in the textual representation of p, unless S is the last rhs symbol of p, in which case the symbol is the lhs symbol of p.
- If *reversed* is *false*, this is the symbol preceding S in the textual representation of p, unless S is the first symbol on the rhs of p, in which case the symbol is the lhs symbol of p.

The remote upward reference that generated the implicit assignment occurs in the subtree rooted at S. By executing the implicit assignment just before visiting S, the value most recently assigned to the upward attribute is passed downward to the point of the original upward reference. c

3.1.6 Attribute Suffixing

TreeSem allows multiple explicit assignments to the same attribute and implicit assignments may generate assignments in which attributes reference themselves. Both of these situations are not allowed in an AG specification. To solve this problem, attribute names are given an integer suffix, therefore creating new unique attributes. These suffixes reflect the order in which assignments are made to attributes in the TreeSem program. They are assigned in such a way that attribute *values* referenced in the original TreeSem program are the ones that are referenced in the AG translation, even though the attribute *names* are different. A "*ct*" associated with each attribute occurrence is used to determine the attribute suffixes. Initially, all *cts* are assigned to be the *max_old* value for the attribute, which is the maximum number of assignments that could have been made to the attribute during a previous traversal. At the start of translation, *max_old* is "0" for all attributes, and it is updated after translation of each traversal.

3.1.6.1 Maximum Assignment Counts

The strategy used to assign attribute suffixes is based on the fact that it is possible to determine, for each attribute occurrence, the maximum possible number of assignments made to this attribute *before*, *during*, and *after* the visit to the owning symbol of the attribute during a traversal. These counts are the same for all instances of a particular attribute occurrence. The term *attrigroup* will refer to the collection of all instances of a particular attribute occurrence across all productions. The maximum counts are computed for each attrigroup, and each member of the attrigroup references the same count. For an attrigroup with elements $R.x$,

1. *max_before* is the maximum *assigned_down* count associated with $R.x$ anywhere R appears as a rhs node of a production.

2. *max_after* is the maximum *assigned_up* count associated with $R.x$ anywhere R appears as a rhs node of a production.

3. *max_root* is the maximum of the sum of the *assigned_down* and *assigned_up* counts associated with $R.x$ for each instance of R as a lhs node of a production.

3.1.6.2 Downward Applied Downward Reference Suffixes

Explicit assignments containing remote downward references, with *false* BEFORE values, generate implicit downward reference assignments with *false* BEFORE values. Since the explicit assignment has a *false* BEFORE value, the node containing the downward referenced attribute has not yet been visited when the explicit assignment is applied. Therefore, implicit downward reference assignments are considered to be evaluated on a separate pass over the input tree, prior to the actual traversal. For each implicit downward reference assignment with a *false* BEFORE value,

1. The *ct* associated with the attribute on the lhs of the assignment is incremented, and the resulting value is assigned as the suffix of that attribute.

2. The *ct* associated with the attribute on the rhs of the assignment is incremented, unless the attribute is the actual downward attribute being referenced (this can be determined from the downward name specification). The resulting *ct* is assigned as the attribute suffix. The reason the *ct* is not incremented for the actual downward attribute is that no assignment will be made to it between the end of the previous traversal and the time it is referenced. The right hand sides of all other assignments will have been the left hand sides of previously executed assignments. Recall that no duplicate implicit assignments are allowed, so exactly one assignment is made to each attribute occurring on the lhs of a downward applied downward reference assignment.

For each attrigroup, $R.x$, if an attribute occurrence $R.x$ appears on the lhs of a downward applied downward reference (dadr) assignment, the corresponding *max_root* value is decremented, and the corresponding *max_before* value is incremented. A dadr assignment contributed to the *max_root* count but it should now contribute to the *max_before* count, since dadr assignments are applied during a separate pass *before* the actual pass. The *max_root* and *max_before* values for any attrigroup are only adjusted once.

For each attribute occurrence appearing on the lhs of a dadr assignment, the *ct* values for all members of its attrigroup are updated to the *ct* value of that attribute occurrence.

3.1.6.3 Explicit Assignment Suffixes

Suffixes are added to attributes occurring in explicit assignment statements in such a way that the *last* assignment made to each member of an attrigroup *before* the attribute's node is visited, *after* it is visited, and when it occurs as the *root* node of a production will have the same suffix. For each production, the assignments corresponding to each symbol of the production are considered in the order in which they appear in the explicit assignment list for that symbol. Suffixes are computed for the assignments of the root symbol of a production first, and then for the rhs symbols in the order indicated by the current value of *reversed*.

For each attribute of a lhs node of a production, the corresponding *ct* is assigned the the sum of the *max_before* and *max_old* values for its attrigroup. All assignments made to the lhs node of a production before the node is visited have already taken place when the node appeared on the rhs of some other production.

Immediately before determining the suffixes for the assignments associated with a rhs symbol, the *ct* values for all attributes of that symbol are assigned the sum of the corresponding *max_old*, *max_before*, and *max_root* counts. The assignments associated with a rhs are those that are applied immediately after the traversal returns from the subtree visit to that symbol, so all *before* and *root* assignments made to this symbol's attributes have already taken place.

For each assignment, the suffixes for all attributes appearing on the rhs of the assignment are assigned their current *ct* values. Then the *ct* for the attribute on the lhs of the assignment is modified in one of the following ways, using the counts appropriate for this attribute:

1. If the attribute's owning symbol is the lhs symbol of the production, and

its *ct* equals "*max_old* + *max_before* + *assigned_up* + *assigned_down* - 1", then *ct* becomes "*max_old* + *max_before* + *max_root*." This is the case when this assignment is the last one made to this root attribute in the context of this production.

2. If the attribute's owning symbol is a rhs symbol of the production, and its *ct* equals "*max_old* + *assigned_down* - 1," then *ct* becomes "*max_old* + *max_before*." In this situation, the assignment is the last one made to this attribute before the owning symbol is visited.

3. If the attribute's owning symbol is a rhs symbol of the production, and its *ct* equals "*max_old* + *max_before* + *max_root* + *assigned_up* - 1," then *ct* becomes "*max_old* + *max_before* + *max_root* + *max_after*." In this situation, the assignment is the last one made to the rhs symbol's attribute in the context of this production.

4. If none of the above cases apply, *ct* is incremented. This assignment is not critical to the rest of the numbering scheme.

The suffix for the attribute on the lhs of the assignment is the resulting value of the corresponding *ct*.

3.1.6.4 Upward Applied Downward Assignment Suffixes

As with downward applied downward reference assignments, upward applied downward assignment (uada) assignments are considered to take place on a separate pass over the tree. This extra pass takes place after the main traversal.

Since all implicit assignments are unique, and since all other assignments have already taken place at the time uada assignments are applied, suffixes for attributes occurring in these assignments are easily determined. The *ct* values for the attributes on the lhs and the rhs of all uada assignments are assigned as the sum of the corresponding *max_old*, *max_root*, *max_before*, and *max_after* counts. The suffixes for the attributes are the resulting *ct* values.

Note that at this point, the *ct* for any element of an attrigroup is the sum of *max_old*, *max_root*, *max_before*, and *max_after* for that attrigroup. These *ct* values will become the basis for computing the attribute suffixes for the next traversal.

3.1.7 ASE Adjustment

After attribute suffixes have been added to eliminate multiple assignments to the same attribute, there is one more translation step that must be performed to ensure that the generated AG will satisfy the conditions for Alternating Semantic Evaluation. It is possible that some assignment statements may contain references to attributes that were assigned to earlier in this pass, but after their owning symbols were visited or they have not yet been visited. This type of reference would cause the assignment to be delayed in the ASE. To remove this problem, a very simple form of symbolic execution is performed on the assignments contained in the explicit assignment lists for each symbol of each production. Note that the dadr and uada assignments are considered to occur during separate passes from the explicit assignments, and therefore do not have to be considered during this symbolic execution process.

For each production, the explicit assignments associated with symbols of that production are ordered in the order they will be applied during the traversal of that production. A simple concatenation of the explicit assignment list for the root symbol, followed by those of all the rhs symbols in the production, either from left to right if *reversed* is *false* or right to left if *reversed* is *true*, provides this ordering. For each assignment, in the order determined above, each of the attribute references on the rhs of the assignment is analyzed. If there is an earlier assignment to this attribute, the reference to this attribute is replaced with the rhs of the earlier assignment. If there is no earlier assignment to this attribute, the reference is left unchanged.

3.1.8 Intertraversal Ad Hocery

For each attrigroup, the *max_old* count becomes the sum of the *max_old*, *max_before*, *max_root*, and *max_after* counts. Then the *max_before*, *max_root*, and *max_after* counts are all set to zero.

For each production, all of the assignments in the explicit assignment lists for the symbols of that production are added to an initially empty list of assignments for that production. The explicit assignment lists are then cleared. For each attribute of each symbol, any dadr or uada assignments are added to the list of assignments for the production. The *assigned_up* and *assigned_down* counts for each attribute are set to zero.

After translation of all TRAVERSALs, the assignments associated with each production are the defining functions for the attributes of symbols in that production.

3.2 Translation from AG to TreeSem

The mapping of an ASE attribute grammar specification into an equivalent TreeSem specification is achieved by constructing a sequence of TreeSem TRAVERSALs corresponding to the passes executed by the Alternating Semantic Evaluator. The GUARDs and WHENs of TreeSem are introduced to mimic ASE control flow in the TreeSem program. This method of translation from an AG to a TreeSem program was developed simply to show that such a translation is possible.

The ASE membership algorithm (see page 14) is used to determine the number of passes, m, required for evaluation of the attribute grammar, and which attributes are evaluated during each pass. Each of the m passes generates a TreeSem TRAVERSAL whose ORDER is preorder. The TRAVERSALs appear in the order of their corresponding passes. A *reverse* statement is generated after every TRAVERSAL, except the last one.

The body of the TRAVERSAL corresponding to a particular pass, k, contains an assignment statement for every attribute occurrence, $R.x$, that is evaluated on pass k. The GUARD for each assignment is the production, p, with which the attribute occurrence is associated. If k is even, the WHEN corresponding to the assignment is the symbol following R in p, or the root symbol of p if R is the last symbol of p. If k is odd, the WHEN corresponding to the assignment is the symbol preceding R in p, or the last symbol of p if R is the root symbol of p. The reason for choosing the WHENs in this way is that evaluation of each assignment is postponed to exactly the point in the traversal that it would be evaluated by the Alternating Semantic Evaluator.

Chapter 4

Proof of Algorithms

This chapter provides proof that TreeSem and ASE attribute grammars are equivalent in terms of their power to express the semantics of programming languages. The equivalence is much more of a "natural" equivalence than that discussed by Knuth,[1] when he demonstrates that all attribute grammar subclasses have the same power by synthesizing all information to the root of the derivation tree and applying a single function to achieve the meaning of the tree. The equivalence of expressive power between ASE attribute grammars and TreeSem is shown by proving that the two translations of Chapter 3 produce specifications that are evaluable in their target languages, and have the same meaning as the original specifications. Neither of these translations requires new functions to be created. Once the expressive equivalence of these two specification methods has been established, readablity and efficiency determine which should be used.

Theorem 1 *Translation of a TreeSem program using the algorithm of section 3.1 results in an attribute grammar that satisfies the conditions for ASE membership.*

Proof of this theorem follows directly from the following Lemma, which places an upper bound on the number of passes required for evaluation of the generated

[1] Knuth, 1968.

AG.

Lemma 1 ***The attribute grammar resulting from translation of a TreeSem program containing k traversals is ASE evaluable in at most $k * 4$ passes.***

Proof of Lemma 1 is by induction on k, the number of traversals in the TreeSem program.

Basis Step. For $k = 0$, there are no assignment statements in the TreeSem program, since assignments can only occur inside traversals. Assignments in the AG program either (1) correspond directly to TreeSem assignments, (2) are generated by remote attribute specifications occurring in TreeSem assignments, or (3) are added as catchup assignments when some member of an attrigroup is assigned to more times than others. Since there are no TreeSem assignments, (1) the corresponding number of AG assignments is 0, (2) there are no remote specifications to generate AG assignments, and (3) all attribute occurrences are assigned to an equal number (zero) of times, so no catchup assignments are generated. Therefore, the AG contains no attribute assignments, and is trivially evaluable in $0 = 0 * 4 = k * 4$ passes.

Induction Hypothesis. For $k > 0$, assume that the AG program resulting from translation of a TreeSem program containing $k - 1$ traversals is evaluable in $(k-1)*4$ alternating passes. Then the AG program resulting from translation of a TreeSem program with k traversals is evaluable in $k * 4$ alternating passes.

Induction Step. Downward applied downward reference assignments resulting from translation of traversal k are evaluable on a single pass. A single downward reference specification generates a chain of n assignment statements, each of the form $lhs_n \leftarrow rhs_n$, evaluated in the context of n unique productions. As a result of the naming conventions for downward attributes and the requirement that no duplicate implicit assignments are allowed, no downward applied downward reference assignment occurs in more than one of these chains. For assignment i, $i \in \{1, \ldots, n\}$, lhs_i is a synthesized attribute of the lhs symbol of production i, and rhs_i is an attribute of one of the rhs symbols of production i. For $i \in \{1, \ldots, n-1\}$, $rhs_i \equiv lhs_{i+1}$. Since these are the first assignments being applied on this traversal, rhs_n must have been assigned on a previous traversal and, by the induction hypothesis, have been evaluated by the end of pass $(k-1)*4$, so lhs_n can be evaluated. For $i \in \{1, \ldots, n-1\}$, we can evaluate lhs_i if rhs_i has previously been evaluated. But since $rhs_i \equiv lhs_{i+1}$ and synthesized attributes of lhs symbols of a production are evaluated after the child nodes of that production have been visited, rhs_i will have be evaluated before it is time to evaluate lhs_i, so lhs_i can be evaluated. Thus, at the end of pass $(k-1)*4+1$, all downward applied downward reference assignments will have been evaluated.

All upward reference assignments, downward applied downward assignment assignments, explicit assignments, upward applied downward reference assignments, and upward assignment assignments resulting from translation of traversal k are evaluable in a single pass of the Alternating Semantic Evaluator. The direction of this pass must be left to right if *reversed* is *false* for this traversal, and right to left otherwise. Either pass $(k-1)*4+2$ or pass $(k-1)*4+3$ will be in the direction required for traversal k. We assume now that the proper pass is chosen, and show that all of the above mentioned assignments are evaluable during this pass.

The ASE membership algorithm eliminates an attribute from the set of attributes evaluable on the current pass only when it references an attribute that was not assigned on an earlier pass *and* results in a *false* β value. Recall that each assignment to an attribute occurrence generates a new attribute name, and therefore a new attribute occurrence in the resulting AG. By the inductive hypothesis, those attributes that resulted from assignments occurring in earlier traversals have been assigned to on a previous pass. Attributes resulting from downward applied downward reference assignments have been previously assigned on pass $(k-1)*4+1$. The only attribute references that could eliminate an attribute from the set of attributes evaluable on this pass, are references made to attributes that resulted from assignments occurring in *this* traversal; in particular, synthesized attributes of the lhs symbol of the production and attributes of rhs symbols that do not occur before the symbol of the attribute being defined. Since flow of control is explicitly defined in TreeSem, all attributes must have been previously assigned when they are referenced. This prohibits references to synthesized attributes of rhs symbols that are not visited before the symbol owning the attribute being defined, in the path of the traversal. The ASE Adjustment algorithm successively (in the direction of the traversal) replaces all references to attributes that were assigned in the context of this production, prior to this assignment, with their definitions. Thus, no references to inherited attributes of rhs symbols or synthesized attributes of the lhs symbol, that were assigned during this traversal, remain. Therefore, β will always return *true*, so no attributes will be eliminated.

At the end of pass $(k-1)*4+3$, the only attributes generated as a result of traversal k, that are *not yet* assigned, are attributes occurring on the lhs of upward applied downward assignment assignments. These can all be assigned on a single pass of the Alternating Semantic Evaluator. As with downward applied downward references, each downward assignment specification generates a chain of n assignments, each of the form $lhs_n \leftarrow rhs_n$. For assignment i, $i \in \{1,\ldots,n\}$, lhs_i is an inherited attribute of a rhs symbol of production i, and rhs_i is an inherited attribute of the root symbol of production i. Rhs_1 occurs as the lhs of the original *explicit* TreeSem assignment statement, and therefore has been assigned to by the end of pass $(k-1)*4+3$. Thus lhs_1 can be assigned. For

$i \in \{2,\ldots,n\}$, we note that, since lhs_i is an inherited attribute of a rhs symbol and rhs_i is a synthesized attribute of the root symbol, $\beta(lhs_i, rhs_i)$ always yields *true*, so lhs_i can be assigned during this pass. Adding this additional pass, all attributes resulting from assignments in traversals $1,\ldots,k$ are evaluated by the end of pass $(k-1)*4+3+1$, or $k*4$. □

Theorem 2 *Translation of an* m*-pass ASE attribute grammar using the algorithm of section 3.2 results in a TreeSem program containing* m *traversals.*

This theorem is included for completeness. Its proof should be obvious from examination of the algorithm, which states, "each of the m passes generates a TreeSem traversal ...." No other traversals are generated by the algorithm. □

A *significant* attribute of an AG resulting from translation of a TreeSem program is one that corresponds directly to a local attribute in the TreeSem program.

Theorem 3 *The attribute grammar resulting from translation of a TreeSem program using the algorithm of section 3.1 assigns all significant attributes the same value as the value assigned to the corresponding attribute by the TreeSem program.*

In order to prove this theorem, several Lemmas are introduced.

Lemma 2 *At the end of a TreeSem traversal, the values assigned to attributes by downward remote assignments are the same values that are assigned to these attributes when downward assignment propagation (Section 3.1.3.2) has been used to replace the remote assignments with chains of implicit assignments.*

First consider the case where the BEFORE value of the implicit assignments is *false*. The original remote assignment must have been executed before the visit to the subtree containing the goal attribute of the assignment. If no other downward remote assignments are made to this attribute before the production containing the symbol owning the goal attribute is visited, it is clear that the implicit assignment chain assigns the proper value to the goal attribute. If such an assignment, s, does occur, it must be applied in the context of one of the productions in the original downward specification. Since implicit downward assignment assignments are executed as soon as the root symbol of a production is visited, s must be executed after the implicit assignment, overwriting the value assigned by the implicit assignment. This new value will be propagated downward to the goal attribute, which is just what is desired. Of course, if more than one of these downward assignments occurs in the context of the same production, the value assigned by the last one executed will be propagated downward.

Now consider the case where the BEFORE value of the implicit assignments is *true*. The implicit assignments are executed on the downward pass of a separate traversal after the main traversal. The uppermost assignment made to a downward attribute will be assigned to the lhs of the first implicit assignment in the chain of implicit assignments leading to that attribute. Since these downward assignments were applied on the upward pass of the main traversal, this assignment was the last to be executed, and therefore this is the value that should be assigned to the downward attribute. All other implicit assignments on the chain will reference the value assigned to the attribute on the lhs of the previous implicit assignment in the chain, thus assigning this value to the goal attribute. □

Lemma 3 *At the end of a TreeSem traversal, the values assigned to attributes by upward remote assignments are the same values that are assigned to these attributes when upward assignment propagation (Section 3.1.3.4) has been used* *to replace the remote assignments with chains of implicit assignments.*

Implicit upward assignment assignments associated with a symbol are executed as soon as the traversal returns from the subtree visit to that symbcl. If only one upward remote assignment is made to a symbol, it is clear that the chain of implicit assignments will assign the correct value to the remote attribute. If more than upward assignment is made to the same remote attribute, the paths of the implicit assignment chains generated by these assignments must ultimately coincide. In the production in which these paths coincide, the implicit assignments of both paths assign to the same attribute of the root node of the production. Consider such a production with root node *r* and child nodes *c1* and *c2*, where *c1* is visited before *c2* in this traversal. The upward remote assignment occurring in the subtree with root *c1* is executed before the one in the subtree with root *c2*. As soon as the visit to *c1* returns, the implicit assignment on the upward path through *c1* is executed. When the visit to *c2* returns, the implicit assignment on the upward path through *c2* is executed, overwriting the value assigned by the implicit assignment on the path through *c1*. The value passed upward is the value assigned to the upward attribute by the most recently executed upward remote assignment. If an upward remote assignment statement occurring in the context of the same production containing *c1* and *c2* were executed after the subtree visit to *c2*, it would again overwrite the value of the root attribute, and that value would be passed upward for assignment to the remote attribute. □

Lemma 4 *The value assigned to the attribute introduced as the result of a downward reference specification is the desired value of the goal attribute at the time the assignment containing the reference is executed.*

First consider the case where the implicit assignments generated by the downward reference have *false* BEFORE values. Clearly the chain of assignments assigns the value of the goal attribute at the beginning of the traversal to the attribute appearing on the lhs of each assignment in the chain. In particular, this value is assigned to the attribute x that replaced the original downward reference specification. If no downward assignments are made to the goal attribute before the assignment, s, containing the downward reference is executed, this is the value desired. If, however, some downward assignment is made to the goal attribute before s is executed, the new value of the goal attribute is the desired value. Since we are considering assignments with *false* BEFORE values, the DOWN_SPEC symbol of the downward specification has not yet been visited. Since this symbol has not been visited, neither have any of the productions in the subtree below it, so any assignment made to the goal attribute must have been made as a downward assignment, that generated downward implicit assignments with *false* BEFORE values. In order to assign to the same goal attribute, the chain of implicit downward assignments must include an assignment to the attribute x that replaced the original downward reference specification. If the downward assignment was made in this production, it was an explicit assignment that occurred earlier than s, and the statement was applied before s. If the assignment occurred in some other production, then the implicit assignment to x was applied before s, since implicit downward assignment assignments are executed before any explicit assignments. So in either case, x holds the newly assigned value of the goal attribute when the statement containing the downward reference is made.

Now consider the case where the implicit assignments generated by the downward reference have *true* BEFORE values. Label the assignments in the chain generated by the downward specification, $a_1, \ldots, a_n$, where a_n is the assignment associated with the symbol owning the goal attribute. Since these assignments have *true* BEFORE values, all the symbols in the subtree with the DOWN_SPEC symbol of the downward specification as its root, have been visited. This includes the goal symbol of the downward specification. The implicit upward applied downward reference assignments of a production are executed just before the traversal leaves the production, so the value assigned to the attribute on the lhs of a_n is the value of the goal attribute as the traversal ascends. If no upward applied downward assignments are made to the goal attribute, then this value will be propagated upward by $a_{n-1}, \ldots, a_1$ and correctly referenced. If such an assignment is made, it must occur *before* the implicit assignment, since implicit downward reference assignments are executed after all other assignments in the context of a given production. The downward assignment will assign to the same attribute that is referenced by the implicit assignment, so this value will be passed upward. In fact, since the downward assignment to the attribute referenced by the implicit assignment occurs before the implicit assignment, the ASE Adjustment algorithm will replace the rhs of the implicit assignment with the value being assigned to the downward attribute. □

Lemma 5 *The value assigned to the attribute introduced as the result of an upward reference specification is the desired value of the goal attribute at the time the assignment containing the reference is executed.*

All references to upward attributes must occur in the subtree with the symbol owning the referenced attribute as its root. Implicit upward reference assignments generated by assignments occurring in a subtree are executed just before the root symbol of the subtree is visited. It is obvious then, that if no assignments are made to the upward attribute while the subtree is being visited, the chain of implicit upward reference assignments will propagate the value of the upward remote attribute to the attribute that replaced the upward reference specification. If, however, some assignment to the upward attribute is made in the subtree, at a point in the traversal before the assignment containing the upward reference is executed, this becomes the desired value. If this assignment occurs in the context of a production on the path from the reference to the goal attribute, the rhs of the implicit assignment in that production will be replaced with the rhs of the assignment to the upward attribute, by the ASE Adjustment algorithm. If this assignment occurs in a production not on this path, the chain of implicit assignments propagating this upward assigned value must ultimately coincide with the path of the upward reference chain. In the production in which this path coincides, the implicit upward assignment is executed as soon as the traversal returns from visiting the node through which the path passes. This must be *before* the traversal descends to the node in this production that is on the upward path, and therefore, before the implicit upward reference assignment is executed. The upward assignment assignment is made to the attribute that is referenced by the upward reference assignment. So the ASE Adjustment algorithm will replace the rhs of the implicit upward reference assignment with the rhs of the upward assignment assignment, which contains the newly assigned value of the upward attribute. □

Lemma 6 *The downward catchup assignments added by the method of Section 3.1.4.1 prevent implicit assignments generated from downward remote attribute assignments from changing the values of significant attributes that are not the goal attributes of the assignments.*

Downward catchup assignments are added whenever an attribute is referenced by an implicit assignment in a chain of assignments resulting from a downward remote assignment, but no implicit or explicit assignment with the same BEFORE value as the implicit assignment containing the reference has assigned this attribute a value during this traversal. The lhs of the catchup assignment is the attribute being referenced and the rhs of the assignment is a downward reference to the attribute occurring on the lhs of the downward remote assignment that generated the implicit assignment. If BEFORE is *false* for the implicit assignment, the catchup assignment is associated with the lhs symbol of the production containing the implicit assignment, ensuring it will be executed before the implicit assignment is executed. If BEFORE is *true* for the implicit assignment, the catchup assignment is associated with the last symbol of the production to be visited during this traversal. This ensures that no assignment to the downward attribute can be made between execution of the catchup assignment and execution of the implicit assignment containing the reference. (If there were such an assignment, it would have to occur in the context of this production and have a *true* BEFORE value, so the catchup assignment would not have been added.) In either the *true* or the *false* case, the value assigned to the attribute on the lhs of the catchup assignment is the value of the downward remote attribute just prior to execution of the implicit assignment. Lemma 4 guarantees that this value will be properly referenced. The chain of implicit downward assignments will now propagate the old value of the remote attribute as the new value of the downward attribute, thus leaving it unchanged. □

Lemma 7 *The upward catchup assignments added by the method of Section 3.1.4.2 prevent implicit assignments generated from upward remote attribute assignments from changing the values of significant attributes that are not the goal attributes of the assignments.*

Upward catchup assignments are added whenever an attribute is referenced by an implicit assignment in a chain of assignments resulting from an upward remote assignment, but no implicit or explicit assignment has assigned this attribute a value during this traversal. The lhs of the catchup assignment is the attribute being referenced and the rhs of the assignment is an upward reference to the attribute occurring on the lhs of the upward remote assignment that generated the implicit assignment. The catchup assignment is associated with the lhs symbol of the production containing the referenced attribute. The value of the upward attribute can not change between the time the catchup assignment is executed and the time the implicit assignment containing the reference is executed, since any upward remote assignment to that attribute would have resulted in an assignment to our "problem" attribute, and the catchup assignment would not have been added. Lemma 5 guarantees the current value of the upward attribute will be referenced by the catchup assignment and assigned to the "problem" attribute. This attribute is then referenced by the implicit upward assignment assignment and the old value of the remote attribute is propagated as the new value for this attribute, thus leaving it unchanged. □

Lemma 8 *Explicit catchup assignments do not change attribute values.*

The proof of this lemma is immediately obvious, since the lhs and the rhs of an explicit catchup assignment always refer to the same local attribute. □

Lemma 9 *The effect of a TreeSem program is unchanged after application of remote attribute translation, downward reference propagation, downward assignment propagation, upward reference propagation, upward assignment propagation, downward catchups, upward catchups, and explicit catchups as described in Chapter 3.*

This lemma follows directly from Lemmas 2–8. □

Lemma 10 *Using the scheme of Section 3.1.6, the suffix assigned an attribute reference is always the same as the suffix assigned to the attribute the last time it occurred on the lhs of an assignment, as assignments are considered in the order in which they are executed in a TreeSem program of the form of Lemma 9.*

At the beginning of translation for each traversal, the *ct* for each attribute reflects the suffix of the last assignment previously made to that attribute. This is initially true, since all attributes are assumed to have been assigned some (either a useful or a default) value, and all attribute suffixes are initially 0. At the end of translation of each traversal, the maximum number of assignments made to each attribute previously, *max_old*, is updated, and the *ct* value for each attribute is assigned the *max_old* value for its attrigroup.

Suffixes for attributes occurring in downward applied downward reference assignments are assigned first, since these assignments are the first to be evaluated. As mentioned above, the *cts* hold the suffix of the last assignment made to each attribute on previous traversals. For each implicit assignment, the suffix assigned to the attribute on the lhs of the assignment is one more than the *ct* for that attribute, since it is being newly assigned. The *ct* for the attribute is incremented to reflect the new assignment to the attribute. If the rhs of the assignment is a reference to a goal attribute, this reference is suffixed with the *ct* for the attribute. All other references are suffixed with the *ct* for the referenced attribute plus one, because they are assigned *once* before they are referenced. The *cts* for these attributes are all incremented, so they still contain the suffix of the last assigned attribute. The *cts* are updated for all occurrences of these attributes belonging to rhs symbols of productions, so they still contain the value of the last suffix assigned to each attribute. Also, since all these assignments are made to attributes of lhs symbols of productions, they would have previously contributed to *max_root* counts. Now they have been executed on a previous traversal, so they contribute to the *max_before* counts and not the *max_root* counts anymore. The *max_before* and *max_root* counts are updated to reflect this change.

The strategy used to assign suffixes to attributes occurring in explicit assignments distinguishes three types of *critical* assignments: *before*, *root*, and *after*.

1. If the *max_before* count for an attrigroup is non-zero, then the last assignment made to each *rhs* attribute occurrence belonging to this attrigroup *before* the symbol owning the attribute is visited, is a *critical before* assignment. The suffix assigned to the attribute on the lhs of such an assignment is *max_old* + *max_before*.

2. If the *max_root* count for an attrigroup is non-zero, then the last assignment made to each *lhs* attribute occurrence belonging to this attrigroup is a *critical root* assignment. The suffix assigned to the attribute on the lhs of such an assignment is *max_old* + *max_before* + *max_root*.

3. If the *max_after* count for an attrigroup is non-zero, then the last assignment made to each *rhs* attribute occurrence belonging to this attrigroup *after* the symbol owning the attribute is visited, is a *critical after* assignment. The suffix assigned to the attribute on the lhs of such an assignment is *max_old* + *max_before* + *max_root* + *max_after*.

The method of adding explicit catchup assignment ensures that, in each of the three categories, if any member of an attrigroup has an assignment made to it, all members will have at least one assignment made to them. For each attribute in an attrigroup, the suffix of the last assignment to this attribute in each category is the same. This enables the suffixing scheme to consistently suffix attribute occurrences across productions, even though the context-free nature of the underlying grammar makes it impossible to know exactly which production will be applied to derive a symbol from above or which production will be used to expand a symbol below.

The first step in assigning suffixes for attributes occurring in the assignments of the explicit assignment lists, assigns the *cts* of all attributes of lhs symbols *max_old* + *max_before*, which includes downward reference chain assignments and the maximum number of assignments that could have occurred when this symbol was on the rhs of a production and *before* the rhs symbol was visited. The last assignment to a rhs attribute before its symbol is visited is always suffixed with this count.

Rhs attributes could not have been assigned earlier in this traversal, except for in downward reference chains, and they have all been updated, so all *lhs* and *rhs cts* contain the maximum number of assignments that could have been made to any attribute of a production before the production is visited, and also the actual suffix of the last assignment made to that attribute before the production was visited.

Just before the suffixes for attributes occurring in assignments for a symbol are assigned, the *cts* for all of the attributes of this symbol are changed to *max_old* + *max_before* + *max_root*. These are the same suffixes assigned to these attributes on the left hand sides of *critical root* assignments.

For each assignment in each production, in the order in which they are executed in the production,

1. the suffix of each attributes on the rhs of the assignment is assigned the current *ct* for that attribute – these *cts* are *always* the suffixes of the last assignment made to the attributes.

2. the *ct* for the lhs attribute is incremented ***and*** the suffix is assigned this new *ct*, so the *ct* still reflects the suffix of the most recent assignment to this attribute occurrence. If the assignment is a *critical* one, the *ct* is assigned the critical value and the lhs of the assignment is suffixed with this *ct*.

In light of the above discussion concerning *critical assignments*, it should be evident that the scheme used to assign suffixes to attributes ocurring in explicit assignments assigns a referenced attribute the same suffix as was assigned to the attribute when the attribute was most recently assigned.

Since implicit upward applied downward assignment assignments are unique and they are applied after all other assignments have been executed, the suffix of the attributes on both the lhs and rhs of these assignments is the sum of *max_old*, *max_before*, *max_root*, and *max_after* for these attributes. The *ct*s for each of these attributes is updated to the suffix assigned for the attribute. Thus, after suffixes have been assigned for all the attributes occurring in assignments generated from an original TreeSem traversal, the *ct* s for all attributes are *max_old* + *max_before* + *max_root* + *max_after*. As mentioned earlier, the *max_old* counts for each attrigroup are updated to this sum as well, so suffixes will be consistently applied across traversals.

Since the suffix assigned to an attribute reference is always the same as the suffix assigned to the most recent assignment to that attribute as a TreeSem program is executed, the value referenced by an AG assignment is always the same value referenced by the corresponding TreeSem assignment. □

Lemma 11 *The value assigned to an attribute is unchanged as a result of applying the ASE Adjustment algorithm in Section 3.1.7.*

At the point when the ASE Adjustment Algorithm is applied, each attribute is uniquely assigned. There are no multiple assignments to an attribute. The algorithm simply replaces some references to attributes with their definitions. There is only one definition for each attribute, so a reference to either an attribute or its definition will yield the same value. Thus, the values computed by the right hand sides of assignments, which are the values assigned to the attributes on the left hand sides of assignments, are the same with or without application of ASE Adjustment. □

The proof of Theorem 3 follows from application of the preceding lemmas. Since by lemma 9 the effect of the program is unchanged before suffixing is applied, and by lemmas 10 and 11, suffixing ensures the same values will be referenced by AG assignments as are references in the corresponding TreeSem assignments, the value assigned to any significant attribute in an AG specification is the same value assigned to the corresponding attribute in the original TreeSem specification. □

Theorem 4 *The values assigned to attributes in a TreeSem program generated from an ASE attribute grammar using the translation of Section 3.2 are the same values assigned to these attributes in the attribute grammar.*

Each attribute is assigned exactly once in the generated TML program, using the same definition as in the AG specification. The only consideration, then, is that all attributes have been assigned before they are referenced. This is exactly the same requirement imposed by the ASE membership algorithm for assigning an attribute for evaluation on a particular pass. So by evaluating an attribute at exactly the same point in a traversal of the tree as it is evaluated by the ASE interpreter, the TreeSem evaluator is guaranteed that any attributes referenced will have been previously assigned. Thus the values assigned to attributes by the TreeSem program are the same ones assigned by the AG specification. □

Since the effects of both TreeSem programs and ASE attribute grammars are expressed as the values assigned to their attributes, and by Theorems 3 and 4, the values assigned to attributes by either method are the same, the theoretical specification power of TreeSem is the same as that of ASE attribute grammars. TreeSem, therefore, is sufficiently powerful as to be used as a semantic specification language.

Chapter 5

Conclusion

This chapter presents a brief summary of the purpose of the research presented in this paper, the results obtained by this research, and topics related to this research which may warrant further consideration.

5.1 Summary

This thesis suggests the use of Tree Manipulation Languages for formally specifying programming language semantics. Semantic specifications of this sort serve two purposes: (1) They are used by language designers to define language semantics. (2) They are used as input to automatic compiler generation systems, which, given the language specification, produce a compiler (or some portions of one) for the described language. Compilers parse strings of a language (programs) and typically represent them internally in the form of a tree. Semantic analysis and code generation and optimizations are performed utilizing the information about the program as it is stored in the tree. Since the tree structure is fundamental to the workings of compilers, it seems natural that a language designed specifically for describing operations on trees (TML) be employed to specify the actions of the compiler as it performs semantic analysis and code generation and optimization.

To this end, TreeSem is proposed as a TML for use as a semantic specification language. The features chosen for inclusion in TreeSem and its syntactic structure were selected with the two purposes of semantic specification languages, as stated above, in mind. Firstly, explicit flow of control, multiple attribute assignments, and remote attribute references ease human understanding of a specification as it is either written or read, as compared with the effort necessary to understand other semantic specifications, notably those based on attribute grammars. Secondly, the features chosen for inclusion in TreeSem provide a language that is fully capable of describing language semantics, and they are readily analyzed so that an automatic compiler generator can easily construct a compiler based on a TreeSem specification.

To show that TreeSem is sufficiently powerful to express programming language semantics, algorithms were developed to translate a TreeSem program into a semantically equivalent ASE attribute grammar, and to translate an ASE attribute grammar into an equivalent TreeSem specification. The correctness of the translations provided by these algorithms was shown. The translations provide knowledge that the semantic specification power of TreeSem and ASE attribute grammars are equivalent. Since all subclasses of attribute grammars are known to be expressively equivalent, and since attribute grammars are able to fully express programming language semantics, TreeSem is also fully capable in its semantic specification power.

An additional result provided by these translation algorithms is that any gains in the efficiency of translators based on either attribute grammars or TreeSem can be taken advantage of in interpretation of the other method. Thus the writer of a specification to be used for automatic compiler generation need not be concerned over which method will result in the most efficient compiler. He is free to used the method he is most comfortable with. The translation of a TreeSem program into an attribute grammar also provides a method of executing TreeSem programs without actually having a TreeSem interpreter.

5.2 Future Research

Future research related to the work described in this dissertation falls into three areas: (1) empirical studies, (2) enhancements to TreeSem, and (3) methods for interpretation of TreeSem programs.

It was claimed in the previous section, that TreeSem programs are easier to read and write than the corresponding attribute grammar specifications. Although this would seem to be the case, no actual studies have been performed to support this claim. Whereas ease of use is a subjective concept, objective measurements as the time required to write a specification and the number of errors in a specification could be used as a basis for empirical observations. Additional statistics need to be determined on the comparative time and space required for evaluation and storage of semantic trees based on attribute grammars and TreeSem.

Although TreeSem is fully capable of expressing programming language semantics, enhancements or extensions to the language may make programming easier without decreasing the efficiency of the compiler generator or, more importantly, the resulting compiler. Recursively called traversals and nested traversals were considered as TreeSem was being defined; however, the usefulness of these control flow constructs seemed limited, and they would interfere significantly in the understanding and analysis of TreeSem programs. Remote references other than upward and downward remote references (particularly across references), were also considered and found to have little practical application. If applications were found to significantly benefit from inclusion of any of these or other language features, TreeSem could be extended.

Many aspects of analysis and interpretation of TreeSem specifications require further research and/or development. The translation algorithms presented in Chapter 3 were developed primarily for their theoretical results. In order to make use of them in a productive environment, they need to be refined, making use of available information to reduce attribute storage and increase execution speed. Techniques for determining the optimal methods for interpreting tree manipulation languages in general need to be applied to the implementation of an actual TreeSem interpreter. A final research topic in this area is analysis of TreeSem programs to determine what portions of the input tree could be visited simultaneously. The traversal structure of TreeSem facilitates identification of operations that can be executed in parallel.

Appendix A

TreeSem Syntax

The BNF syntax of TreeSem is presented below. Nonterminal symbols are upppercased. The angle-bracketed nonterminal symbols ⟨id_sym⟩, ⟨number⟩, and ⟨end_of_line⟩ are expected to be returned from a scanner when an identifer, number, or end of line are encountered. All other symbols are terminals. Chapter 2 explains the syntax and semantics of TreeSem in detail.

```
PROGRAM          → DECLS GRAMMAR TRAVSEQ
DECLS            →
                  | attribute types DECL_LIST
DECL_LIST        → DECL
                  | DECL_LIST DECL
DECL             → ⟨id_sym⟩ = ⟨id_sym⟩ ;
GRAMMAR          → grammar PRODLIST end_grammar
PRODLIST         → PROD
                  | PRODLIST PROD
PROD             → LHS → RHSLIST ⟨end_of_line⟩
LHS              → ⟨id_sym⟩
RHSLIST          →
                  | ⟨id_sym⟩ RHSLIST
```

```
TRAVSEQ          → TRAVREV
                   | TRAVSEQ TRAVREV
TRAVREV          → TRAVERSAL
                   | REVERSAL
TRAVERSAL        → traverse ORDER GUARD_LIST end_traverse
ORDER            → inorder
                   | preorder
                   | postorder
REVERSAL         → reverse ;
GUARD_LIST       → GUARD_STMT
                   | GUARD_LIST GUARD_STMT
GUARD_STMT       → GUARD ⇒ SEM_LIST
GUARD            → SQUARE_TREE
SEM_LIST         → WHEN_ASGN
                   | SEM_LIST WHEN_ASGN
WHEN_ASGN        → WHEN ASGN_LIST
WHEN             → @ ID_SYM_PLUS :
ASGN_LIST        → ASSIGNMENT
                   | ASGN_LIST ASSIGNMENT
SQUARE_TREE      → [ IDLIST ]
IDLIST           → OPT_DOWNID
                   | OPT_DOWNID , IDLIST
OPT_DOWNID       → ID_SYM_PLUS
                   | DOWNID
DOWNID           → # ID_SYM_PLUS
ID_SYM_PLUS      → ⟨id_sym⟩ ( ⟨number⟩ )
                   | ⟨id_sym⟩
```

```
SQUARE_TREE_LIST →
                   | SQUARE_TREE_LIST SQUARE_TREE
DOWN_SPEC          → DOWNID < SQUARE_TREE_LIST >
                     SQUARE_TREE . (id_sym)
OPTARGLIST         →
                   | ( ARGLIST )
ARGLIST            → TREE
                   | TREE , ARGLIST
ASSIGNMENT         → DIRTREE ← TREE ;
DIRTREE            → ↑ID_SYM_PLUS . (id_sym)
                   | DOWN_SPEC
                   | ID_SYM_PLUS . (id_sym)
TREE               → DIRTREE
                   | FN_CALL
```

Appendix B

Implementation Statistics

The translations described in Chapter 3 have been implemented as C programs and are currently running on Sun Workstations under UNIX$^{\text{TM}}$. An attribute evaluator was written in Sheffield Pascal and runs on a Prime 9950. The following is a tabulation of the files comprising each of the programs.

File	Lines	Description
tml.l	106	lex specification for TreeSem translation
tml.procs.c	1891	TreeSem translation procedures
tml.types.h	195	type declarations for TreeSem translation
tml.y	380	yacc specification for TreeSem translation
att.l	40	lex specification for AG translation
att.procs.c	603	AG translation procedures
att.types.h	101	type declarations for AG translation
att.y	253	yacc specification for AG translation
att.pas	1642	ASE attribute evaluator

Appendix C

Example Specification and Translations

The specifications in this appendix illustrate the use of TreeSem features to specify the semantics of a useful programming language, and show how the translations presented in Chapter 3 act on these specifications. The example is an adaptation of an attribute grammar specification for translating the programming language Pam into a simple symbolic machine language. The original attribute grammar specification appears in Pagan, pages 92-97.

C.1 TreeSem Semantics for Pam

```
grammar
   program      ::= series
   series       ::= statement
   series       ::= series semi statement
   statement    ::= input
   statement    ::= output
   statement    ::= assign
   statement    ::= conditional
   statement    ::= def_loop
   statement    ::= indef_loop
   input        ::= read var_list
   output       ::= write var_list
   var_list     ::= variable
   var_list     ::= var_list comma variable
   assign       ::= variable colon equal expression
   conditional  ::= if comparison then series fi
   conditional   ::= if comparison then series else series fi
   def_loop      ::= to expression do series end
   indef_loop    ::= while comparison do series end
   comparison    ::= expression relation expression
   expression    ::= term
   expression    ::= expression weak_op term
   term          ::= element
   term          ::= term strong_op element
   element       ::= constant
   element       ::= variable
   element       ::= left expression right
   constant      ::= digit
   constant      ::= constant digit
   variable      ::= letter
   variable      ::= variable letter
   variable      ::= variable digit
   relation      ::= equal
   relation      ::= equal less
   relation      ::= less
   relation      ::= greater
   relation      ::= greater equal
   relation      ::= less greater
   weak_op       ::= plus
   weak_op       ::= minus
   strong_op     ::= star
   strong_op     ::= divide
   digit         ::= 0
   digit         ::= 9
   letter        ::= a
   letter        ::= z
end_grammar

traverse preorder

   [program, series] -->
      @ program :
         series.temp := fn 0;
         series.label := fn 0;
      @ series :
         program.code := fn append(series.code, fn HALT);

   [series, statement] -->
      @ series :
         statement.label := series.label;
      @ statement :
         series.label := statement.label;

   [series, series, semi, statement] -->
      @ series :
```

```
      series(2).temp := series.temp;
      series(2).label := series.label;
   @ series(2) :
      statement.label := series(2).label;
   @ statement :
      series.label := statement.label;
      series.code := fn concat(series(2).code, series.code);

[statement, conditional] -->
   @ statement :
      conditional.label := statement.label;
   @ conditional :
      statement.label := conditional.label;

[statement, def_loop] -->
   @ statement :
      def_loop.label := statement.label;
   @ def_loop :
      statement.label := def_loop.label;

[statement, indef_loop] -->
   @ statement :
      indef_loop.label := statement.label;
   @ indef_loop :
      statement.label := indef_loop.label;

[input, read, var_list] -->
   @ input :
      #var_list<[var_list, #var_list(2), comma, variable]>
                [var_list, #variable].opcode := fn GET;
   @ var_list :
      ~series.code := var_list.code;

[output, write, var_list] -->
   @ output :
      #var_list<[var_list, #var_list(2), comma, variable]>
                [var_list, #variable].opcode := fn PUT;
   @ var_list :
      ~series.code := var_list.code;

[var_list, variable] -->
   @ variable :
      var_list.code := fn make_op(variable.opcode, variable.tag);

[var_list, var_list, comma, variable] -->
   @ var_list :
      var_list(2).opcode := var_list.opcode;
   @ variable :
      var_list.code := fn append(var_list(2).code, fn make_op(
         #var_list(2)<[var_list, #var_list(2), comma, variable]>
            [var_list, #variable].opcode, variable.tag));

[assign, variable, colon, equal, expression] -->
   @ assign :
      expression.temp := ~series.temp;
   @ expression :
      ~series.code := fn append(expression.code,
         fn make_op(fn STO, variable.tag));

[conditional, if, comparison, then, series, fi] -->
   @ conditional :
      series.temp := ~series.temp;
      comparison.label := fn plus(conditional.label, fn 1);
      series.label := fn plus(conditional.label, fn 1);
   @ fi :
      conditional.label := series.label;
      ~series.code := fn concat(comparison.code, series.code,
         fn make_op(fn plus(fn label(conditional.label), fn 1),
            fn LAB));

[conditional, if, comparison, then, series, else, series, fi] -->
   @ conditional :
      series.temp := ^series.temp;
      series(2).temp := ~series.temp;
      comparison.label := fn plus(conditional.label, fn 1);
      series.label := fn plus(conditional.label, fn 2);
   @ series :
      series(2).label := series.label;
   @ fi :
      conditional.label := series(2).label;
      ~series.code := fn concat(comparison.code, series.code,
         fn make_op(fn J,
            fn label(fn plus(conditional.label, fn 2))),
         fn make_op(fn label(fn plus(conditional.label, fn 1)),
            fn LAB),
         series(2).code,
         fn make_op(fn label(fn plus(conditional.label,
         fn 2)), fn LAB));

[def_loop, to, expression, do, series, end] -->
   @ def_loop :
      expression.temp := fn plus(~series.temp, fn 1);
      series.temp := fn plus(~series.temp, fn 1);
      series.label := fn plus(def_loop.label, fn 2);
   @ end :
      def_loop.label := series.label;
      ~series.code := fn concat(expression.code,
         fn make_op(fn STO, fn temporary(
            fn plus(^series.temp, fn 1))),
         fn make_op(fn label(fn plus(def_loop.label, fn 1)), fn LAB),
         fn make_op(fn LOAD, fn temporary(
            fn plus(^series.temp, fn 1))),
         fn make_op(fn SUB, fn 1),
         fn make_op(fn JN, fn label(fn plus(def_loop.label, fn 2))),
         fn make_op(fn STO, fn temporary(
            fn plus(^series.temp, fn 1))),
         series.code,
         fn make_op(fn J, fn label(fn plus(def_loop.label, fn 1))),
         fn make_op(fn label(fn plus(def_loop.label, fn 2)),fn LAB));

[indef_loop, while, comparison, do, series, end] -->
   @ indef_loop :
      series.temp := ^series.temp;
      comparison.label := fn plus(indef_loop.label, fn 2);
      series.label := fn plus(indef_loop.label, fn 2);
   @ end :
      indef_loop.label := series.label;
      ^series.code := fn concat(
         fn make_op(fn label(fn plus(indef_loop.label, fn 1)),
            fn LAB),
         comparison.code,
         series.code,
         fn make_op(fn J, fn label(fn plus(indef_loop.label, fn 1))),
         fn make_op(fn label(fn plus(indef_loop.label, fn 2)),
            fn LAB));

[comparison, expression, relation, expression] -->
   @ comparison :
      expression.temp := fn plus(^series.temp, fn 1);
      expression(2).temp := fn plus(^series.temp, fn 1);
   @ expression(2) :
      comparison.code := fn concat(expression.code,
         fn make_op(fn STO, fn temporary(
            fn plus(^series.temp, fn 1))),
         expression(2).code,
         fn make_op(fn SUB, fn temporary(
            fn plus(^series.temp, fn 1))),
         fn make_op(comparison.opcode, fn label(comparison.label)));

[expression, term] -->
   @ expression :
      term.temp := expression.temp;
   @ term :
      expression.code := term.code;

[expression, expression, weak_op, term] -->
   @ expression :
      expression(2).temp := expression.temp;
      term.temp := fn plus(expression.temp, fn 2);
   @ term :
      expression.code := fn concat(expression(2).code,
         fn selectcode(term.code, expression.temp, weak_op.opcode));

[term, element] -->
   @ element :
      term.code := element.code;
[term, element] -->
   @ term :
      element.temp := term.temp;

[term, term, strong_op, element] -->
   @ term :
      term(2).temp := term.temp;
      element.temp := fn plus(term.temp, fn 2);
   @ element :
      term.code := fn concat(term(2).code,
         fn selectcode(element.code, term.temp, strong_op.opcode));

[element, constant] -->
   @ constant :
      element.code := fn make_op(fn LOAD, constant.num);

[element, variable] -->
   @ variable :
      element.code := fn make_op(fn LOAD, variable.tag);

[element, left, expression, right] -->
   @ element :
      expression.temp := element.temp;
   @ right :
      element.code := expression.code;

[constant, digit] -->
   @ digit :
      constant.num := digit.num;

[constant, constant, digit] -->
   @ digit :
      constant.num := fn concat(constant(2).num, digit.num);

[variable, letter] -->
   @ letter :
      variable.tag := letter.tag;

[variable, variable, letter] -->
   @ letter :
        variable.tag := fn concat(variable(2).tag, letter.tag);

  [variable, variable, digit] -->
     @ digit :
        variable.tag := fn concat(variable(2).tag, digit.num);

  [relation, equal] -->
     @ equal :
        ^comparison.opcode := fn JNP;

  [relation, equal, less] -->
     @ less :
        ^comparison.opcode := fn JN;

  [relation, less] -->
     @ less :
        ^comparison.opcode := fn JNZ;

  [relation, greater] -->
     @ greater :
        ^comparison.opcode := fn JPZ;

  [relation, greater, equal] -->
     @ equal :
        ^comparison.opcode := fn JP;

  [relation, less, greater] -->
     @ greater :
        ^comparison.opcode := fn JZ;

  [weak_op, plus] -->
     @ plus :
        weak_op.opcode := fn ADD;

  [weak_op, minus] -->
     @ minus :
        weak_op.opcode := fn SUB;

  [strong_op, star] -->
     @ star :
        strong_op.opcode := fn MULT;

  [strong_op, divide] -->
     @ divide :
        strong_op.opcode := fn DIV;

  [digit, 0] -->
     @ 0 :
        digit.num := fn 0;
```

```
   [digit, 9] -->
      @ 9 :
         digit.num := fn 9;

   [letter, a] -->
      @ a :
         letter.tag := fn a;

   [letter, z] -->
      @ z :
         letter.tag := fn z;
 end_traverse
```

C.2 AG Translation of Section C.1

```
attribute types
   comparisonopcode1 = integer;
   num1 = integer;
   tag1 = integer;
   seriescode1 = integer;
   opcode1 = integer;
   p13n1_p12n1_opcode1 = integer;
   p13n1_p12n1_opcode2 = integer;
   code1 = integer;
   code2 = integer;
   label1 = integer;
   label2 = integer;
   seriestemp1 = integer;
   temp1 = integer;

<letter> ::= <z>
rules
     <letter>1.tag1 := z;
selur

<letter> ::= <a>
rules
     <letter>1.tag1 := a;
selur

<digit> ::= <9>
rules
     <digit>1.num1 := 9;
selur
```

```
<digit> ::= <0>
rules
     <digit>1.num1 := 0;
selur

<constant> ::= <constant> <digit>
rules
     <constant>1.num1 := concat(<constant>2.num1, <digit>1.num1);
selur

<constant> ::= <digit>
rules
     <constant>1.num1 := <digit>1.num1;
selur

<strong_op> ::= <divide>
rules
     <strong_op>1.opcode1 := DIV;
selur

<strong_op> ::= <star>
rules
     <strong_op>1.opcode1 := MULT;
selur

<element> ::= <left> <expression> <right>
rules
     <expression>1.temp1 := <element>1.temp1;
     <element>1.code1 := <expression>1.code1;
selur

<element> ::= <variable>
rules
     <element>1.code1 := make_op(LOAD, <variable>1.tag1);
selur

<element> ::= <constant>
rules
     <element>1.code1 := make_op(LOAD, <constant>1.num1);
selur

<weak_op> ::= <minus>
rules
     <weak_op>1.opcode1 := SUB;
selur

<weak_op> ::= <plus>
rules
     <weak_op>1.opcode1 := ADD;
```

```
selur

<term> ::= <term> <strong_op> <element>
rules
     <term>2.temp1 := <term>1.temp1;
     <element>1.temp1 := plus(<term>1.temp1, 2);
     <term>1.code1 := concat(<term>2.code1, selectcode(<element>1.code1,
                           <term>1.temp1, <strong_op>1.opcode1));
selur

<term> ::= <element>
rules
     <element>1.temp1 := <term>1.temp1;
     <term>1.code1 := <element>1.code1;
selur

<relation> ::= <less> <greater>
rules
     <relation>1.comparisonopcode1 := JZ;
selur

<relation> ::= <greater> <equal>
rules
     <relation>1.comparisonopcode1 := JP;
selur

<relation> ::= <greater>
rules
     <relation>1.comparisonopcode1 := JPZ;
selur

<relation> ::= <less>
rules
     <relation>1.comparisonopcode1 := JNZ;
selur

<relation> ::= <equal> <less>
rules
     <relation>1.comparisonopcode1 := JN;
selur

<relation> ::= <equal>
rules
     <relation>1.comparisonopcode1 := JNP;
selur

<comparison> ::= <expression> <relation> <expression>
rules
     <expression>1.temp1 := plus(<comparison>1.seriestemp1, 1);
     <expression>2.temp1 := plus(<comparison>1.seriestemp1, 1);
     <comparison>1.opcode1 := <relation>1.comparisonopcode1;
     <comparison>1.code1 := concat(<expression>1.code1,
         make_op(STO, temporary(plus(<comparison>1.seriestemp1, 1))),
         <expression>2.code1,
         make_op(SUB, temporary(plus(<comparison>1.seriestemp1, 1))),
         make_op(<relation>1.comparisonopcode1,
            label(<comparison>1.label1)));
selur

<expression> ::= <expression> <weak_op> <term>
rules
     <expression>2.temp1 := <expression>1.temp1;
     <term>1.temp1 := plus(<expression>1.temp1, 2);
     <expression>1.code1 := concat(<expression>2.code1,
         selectcode(<term>1.code1, <expression>1.temp1,
         <weak_op>1.opcode1));
selur

<expression> ::= <term>
rules
     <term>1.temp1 := <expression>1.temp1;
     <expression>1.code1 := <term>1.code1;
selur

<variable> ::= <variable> <digit>
rules
     <variable>1.tag1 := concat(<variable>2.tag1, <digit>1.num1);
selur

<variable> ::= <variable> <letter>
rules
     <variable>1.tag1 := concat(<variable>2.tag1, <letter>1.tag1);
selur

<variable> ::= <letter>
rules
     <variable>1.tag1 := <letter>1.tag1;
selur

<var_list> ::= <var_list> <comma> <variable>
rules
     <var_list>2.p13n1_p12n1_opcode1 := <var_list>1.p13n1_p12n1_opcode1;
     <var_list>2.opcode1 := <var_list>1.opcode1;
     <var_list>1.code1 := append(<var_list>2.code1,
         make_op(<var_list>2.p13n1_p12n1_opcode2, <variable>1.tag1));
     <var_list>1.p13n1_p12n1_opcode2 := <var_list>2.p13n1_p12n1_opcode2;
selur
```

```
<var_list> ::= <variable>
rules
     <variable>1.opcode1 := <var_list>1.p13n1_p12n1_opcode1;
     <var_list>1.code1 := make_op(<var_list>1.p13n1_p12n1_opcode1,
          <variable>1.tag1);
     <var_list>1.p13n1_p12n1_opcode2 := <var_list>1.p13n1_p12n1_opcode1;
selur

<indef_loop> ::= <while> <comparison> <do> <series> <end>
rules
     <series>1.temp1 := <indef_loop>1.seriestemp1;
     <comparison>1.label1 := plus(<indef_loop>1.label1, 2);
     <series>1.label1 := plus(<indef_loop>1.label1, 2);
     <comparison>1.seriestemp1 := <indef_loop>1.seriestemp1;
     <indef_loop>1.label2 := <series>1.label2;
     <indef_loop>1.seriescode1 := concat(
         make_op(label(plus(<series>1.label2, 1)), LAB),
         <comparison>1.code1, <series>1.code2,
         make_op(J, label(plus(<series>1.label2, 1))),
         make_op(label(plus(<series>1.label2, 2)), LAB));
selur

<def_loop> ::= <to> <expression> <do> <series> <end>
rules
     <expression>1.temp1 := plus(<def_loop>1.seriestemp1, 1);
     <series>1.temp1 := plus(<def_loop>1.seriestemp1, 1);
     <series>1.label1 := plus(<def_loop>1.label1, 2);
     <def_loop>1.label2 := <series>1.label2;
     <def_loop>1.seriescode1 := concat(<expression>1.code1,
         make_op(STO, temporary(plus(<def_loop>1.seriestemp1, 1))),
         make_op(label(plus(<series>1.label2, 1)), LAB),
         make_op(LOAD, temporary(plus(<def_loop>1.seriestemp1, 1))),
         make_op(SUB, 1), make_op(JN, label(plus(<series>1.label2, 2))),
         make_op(STO, temporary(plus(<def_loop>1.seriestemp1, 1))),
         <series>1.code2,
         make_op(J, label(plus(<series>1.label2, 1))),
         make_op(label(plus(<series>1.label2, 2)), LAB));
selur

<conditional> ::= <if> <comparison> <then> <series> <else> <series> <fi>
rules
     <series>1.temp1 := <conditional>1.seriestemp1;
     <series>2.temp1 := <conditional>1.seriestemp1;
     <comparison>1.label1 := plus(<conditional>1.label1, 1);
     <series>1.label1 := plus(<conditional>1.label1, 2);
     <comparison>1.seriestemp1 := <conditional>1.seriestemp1;
     <series>2.label1 := <series>1.label2;
     <conditional>1.label2 := <series>2.label2;
     <conditional>1.seriescode1 := concat(<comparison>1.code1,
          <series>1.code2,
          make_op(J, label(plus(<series>2.label2, 2))),
          make_op(label(plus(<series>2.label2, 1)), LAB),
          <series>2.code2,
          make_op(label(plus(<series>2.label2, 2)), LAB));
selur

<conditional> ::= <if> <comparison> <then> <series> <fi>
rules
     <series>1.temp1 := <conditional>1.seriestemp1;
     <comparison>1.label1 := plus(<conditional>1.label1, 1);
     <series>1.label1 := plus(<conditional>1.label1, 1);
     <comparison>1.seriestemp1 := <conditional>1.seriestemp1;
     <conditional>1.label2 := <series>1.label2;
     <conditional>1.seriescode1 := concat(<comparison>1.code1,
          <series>1.code2,
          make_op(plus(label(<series>1.label2), 1), LAB));
selur

<assign> ::= <variable> <colon> <equal> <expression>
rules
     <expression>1.temp1 := <assign>1.seriestemp1;
     <assign>1.seriescode1 := append(<expression>1.code1,
          make_op(STO, <variable>1.tag1));
selur

<output> ::= <write> <var_list>
rules
     <var_list>1.p13n1_p12n1_opcode1 := PUT;
     <output>1.seriescode1 := <var_list>1.code1;
selur

<input> ::= <read> <var_list>
rules
     <var_list>1.p13n1_p12n1_opcode1 := GET;
     <input>1.seriescode1 := <var_list>1.code1;
selur

<statement> ::= <indef_loop>
rules
     <indef_loop>1.label1 := <statement>1.label1;
     <indef_loop>1.seriestemp1 := <statement>1.seriestemp1;
     <statement>1.seriescode1 := <indef_loop>1.seriescode1;
     <statement>1.label2 := <indef_loop>1.label2;
selur

<statement> ::= <def_loop>
rules
     <def_loop>1.label1 := <statement>1.label1;
     <def_loop>1.seriestemp1 := <statement>1.seriestemp1;
     <statement>1.seriescode1 := <def_loop>1.seriescode1;
     <statement>1.label2 := <def_loop>1.label2;
selur

<statement> ::= <conditional>
rules
     <conditional>1.label1 := <statement>1.label1;
     <conditional>1.seriestemp1 := <statement>1.seriestemp1;
     <statement>1.seriescode1 := <conditional>1.seriescode1;
     <statement>1.label2 := <conditional>1.label2;
selur

<statement> ::= <assign>
rules
     <assign>1.seriestemp1 := <statement>1.seriestemp1;
     <statement>1.seriescode1 := <assign>1.seriescode1;
selur

<statement> ::= <output>
rules
     <statement>1.seriescode1 := <output>1.seriescode1;
selur

<statement> ::= <input>
rules
     <statement>1.seriescode1 := <input>1.seriescode1;
selur

<series> ::= <series> <semi> <statement>
rules
     <series>2.temp1 := <series>1.temp1;
     <series>2.label1 := <series>1.label1;
     <statement>1.label1 := <series>2.label2;
     <statement>1.seriestemp1 := <series>1.temp1;
     <series>1.code1 := <statement>1.seriescode1;
     <series>1.label2 := <statement>1.label2;
     <series>1.code2 := concat(<series>2.code2,
         <statement>1.seriescode1);
selur

<series> ::= <statement>
rules
     <statement>1.label1 := <series>1.label1;
     <statement>1.seriestemp1 := <series>1.temp1;
     <series>1.code2 := <statement>1.seriescode1;
     <series>1.label2 := <statement>1.label2;
selur
```

```
<program> ::= <series>
rules
     <series>1.temp1 := 0;
     <series>1.label1 := 0;
     <program>1.code1 := append(<series>1.code2, HALT);
selur
```

C.3 TreeSem Translation of Section C.2

```
grammar
   letter       ::= z
   letter       ::= a
   digit        ::= 9
   digit        ::= 0
   constant     ::= constant digit
   constant     ::= digit
   strong_op    ::= divide
   strong_op    ::= star
   element      ::= left expression right
   element      ::= variable
   element      ::= constant
   weak_op      ::= minus
   weak_op      ::= plus
   term         ::= term strong_op element
   term         ::= element
   relation     ::= less greater
   relation     ::= greater equal
   relation     ::= greater
   relation     ::= less
   relation     ::= equal less
   relation     ::= equal
   comparison   ::= expression relation expression
   expression   ::= expression weak_op term
   expression   ::= term
   variable     ::= variable digit
   variable     ::= variable letter
   variable     ::= letter
   var_list     ::= var_list comma variable
   var_list     ::= variable
   indef_loop   ::= while comparison do series end
   def_loop     ::= to expression do series end
   conditional  ::= if comparison then series else series fi
   conditional  ::= if comparison then series fi
   assign       ::= variable colon equal expression
   output       ::= write var_list
   input        ::= read var_list
   statement    ::= indef_loop
   statement     ::= def_loop
   statement     ::= conditional
   statement     ::= assign
   statement     ::= output
   statement     ::= input
   series        ::= series semi statement
   series        ::= statement
   program       ::= series
end_grammar

traverse preorder

   [letter, z] -->
      @ z(1) :
         letter(1).tag1 := fn z;

   [letter, a] -->
      @ a(1) :
         letter(1).tag1 := fn a;

   [digit, 9] -->
      @ 9(1) :
         digit(1).num1 := fn 9;

   [digit, 0] -->
      @ 0(1) :
         digit(1).num1 := fn 0;

   [constant, constant, digit] -->
      @ digit(1) :
         constant(1).num1 := fn concat(constant(2).num1, digit(1).num1);

   [constant, digit] -->
      @ digit(1) :
         constant(1).num1 := digit(1).num1;

   [strong_op, divide] -->
      @ divide(1) :
         strong_op(1).opcode1 := fn DIV;

   [strong_op, star] -->
      @ star(1) :
         strong_op(1).opcode1 := fn MULT;

   [element, left, expression, right] -->
      @ left(1) :
         expression(1).temp1 := element(1).temp1;
      @ right(1) :
         element(1).code1 := expression(1).code1;
```

```
[element, variable] -->
  @ variable(1) :
    element(1).code1 := fn make_op(fn LOAD, variable(1).tag1);

[element, constant] -->
  @ constant(1) :
    element(1).code1 := fn make_op(fn LOAD, constant(1).num1);

[weak_op, minus] -->
  @ minus(1) :
    weak_op(1).opcode1 := fn SUB;

[weak_op, plus] -->
  @ plus(1) :
    weak_op(1).opcode1 := fn ADD;

[term, term, strong_op, element] -->
  @ term(1) :
    term(2).temp1 := term(1).temp1;
  @ strong_op(1) :
    element(1).temp1 := fn plus(term(1).temp1, fn 2);
  @ element(1) :
    term(1).code1 := fn concat(term(2).code1,
      fn selectcode(element(1).code1, term(1).temp1,
        strong_op(1).opcode1));

[term, element] -->
  @ term(1) :
    element(1).temp1 := term(1).temp1;
  @ element(1) :
    term(1).code1 := element(1).code1;

[relation, less, greater] -->
  @ greater(1) :
    relation(1).comparisonopcode1 := fn JZ;

[relation, greater, equal] -->
  @ equal(1) :
    relation(1).comparisonopcode1 := fn JP;

[relation, greater] -->
  @ greater(1) :
    relation(1).comparisonopcode1 := fn JPZ;

[relation, less] -->
  @ less(1) :
    relation(1).comparisonopcode1 := fn JNZ;
```

```
[relation, equal, less] -->
   @ less(1) :
      relation(1).comparisonopcode1 := fn JN;

[relation, equal] -->
   @ equal(1) :
      relation(1).comparisonopcode1 := fn JNP;

[comparison, expression, relation, expression] -->
   @ comparison(1) :
      expression(1).temp1 :=
         fn plus(comparison(1).seriestemp1, fn 1);
   @ relation(1) :
      expression(2).temp1 :=
         fn plus(comparison(1).seriestemp1, fn 1);
   @ expression(2) :
      comparison(1).code1 := fn concat(expression(1).code1,
          fn make_op(fn STO,
             fn temporary(fn plus(comparison(1).seriestemp1, fn 1))),
          expression(2).code1,
          fn make_op(fn SUB,
             fn temporary(fn plus(comparison(1).seriestemp1, fn 1))),
          fn make_op(relation(1).comparisonopcode1,
             fn label(comparison(1).label1)));
      comparison(1).opcode1 := relation(1).comparisonopcode1;

[expression, expression, weak_op, term] -->
   @ expression(1) :
      expression(2).temp1 := expression(1).temp1;
   @ weak_op(1) :
      term(1).temp1 := fn plus(expression(1).temp1, fn 2);
   @ term(1) :
      expression(1).code1 := fn concat(expression(2).code1,
         fn selectcode(term(1).code1, expression(1).temp1,
            weak_op(1).opcode1));

[expression, term] -->
   @ expression(1) :
      term(1).temp1 := expression(1).temp1;
   @ term(1) :
      expression(1).code1 := term(1).code1;

[variable, variable, digit] -->
   @ digit(1) :
      variable(1).tag1 := fn concat(variable(2).tag1, digit(1).num1);

[variable, variable, letter] -->
   @ letter(1) :
      variable(1).tag1 :=
         fn concat(variable(2).tag1, letter(1).tag1);

[variable, letter] -->
   @ letter(1) :
      variable(1).tag1 := letter(1).tag1;

[var_list, var_list, comma, variable] -->
   @ var_list(1) :
      var_list(2).opcode1 := var_list(1).opcode1;
      var_list(2).p13n1_p12n1_opcode1 :=
         var_list(1).p13n1_p12n1_opcode1;
   @ variable(1) :
      var_list(1).p13n1_p12n1_opcode2 :=
         var_list(2).p13n1_p12n1_opcode2;
      var_list(1).code1 := fn append(var_list(2).code1,
          fn make_op(var_list(2).p13n1_p12n1_opcode2,
          variable(1).tag1));

[var_list, variable] -->
   @ var_list(1) :
      variable(1).opcode1 := var_list(1).p13n1_p12n1_opcode1;
   @ variable(1) :
      var_list(1).p13n1_p12n1_opcode2 :=
         var_list(1).p13n1_p12n1_opcode1;
      var_list(1).code1 :=
         fn make_op(var_list(1).p13n1_p12n1_opcode1,
            variable(1).tag1);

[indef_loop, while, comparison, do, series, end] -->
   @ while(1) :
      comparison(1).label1 := fn plus(indef_loop(1).label1, fn 2);
      comparison(1).seriestemp1 := indef_loop(1).seriestemp1;
   @ do(1) :
      series(1).label1 := fn plus(indef_loop(1).label1, fn 2);
      series(1).temp1 := indef_loop(1).seriestemp1;
   @ end(1) :
      indef_loop(1).seriescode1 := fn concat(
          fn make_op(fn label(fn plus(series(1).label2, fn 1)),
             fn LAB),
          comparison(1).code1,
          series(1).code2,
          fn make_op(fn J, fn label(
             fn plus(series(1).label2, fn 1))),
          fn make_op(fn label(fn plus(series(1).label2, fn 2)),
             fn LAB));
      indef_loop(1).label2 := series(1).label2;

[def_loop, to, expression, do, series, end] -->
   @ to(1) :
      expression(1).temp1 := fn plus(def_loop(1).seriestemp1, fn 1);
   @ do(1) :
      series(1).label1 := fn plus(def_loop(1).label1, fn 2);
      series(1).temp1 := fn plus(def_loop(1).seriestemp1, fn 1);
   @ end(1) :
      def_loop(1).seriescode1 := fn concat(expression(1).code1,
         fn make_op(fn STO,
            fn temporary(fn plus(def_loop(1).seriestemp1, fn 1))),
         fn make_op(fn label(
            fn plus(series(1).label2, fn 1)), fn LAB),
         fn make_op(fn LOAD,
            fn temporary(fn plus(def_loop(1).seriestemp1, fn 1))),
         fn make_op(fn SUB, fn 1),
         fn make_op(fn JN, fn label(
            fn plus(series(1).label2, fn 2))),
         fn make_op(fn STO,
            fn temporary(fn plus(def_loop(1).seriestemp1, fn 1))),
         series(1).code2,
         fn make_op(fn J, fn label(fn plus(series(1).label2, fn 1))),
         fn make_op(fn label(fn plus(series(1).label2, fn 2)),
            fn LAB));
      def_loop(1).label2 := series(1).label2;

[conditional, if, comparison, then, series, else, series, fi] -->
   @ if(1) :
      comparison(1).label1 := fn plus(conditional(1).label1, fn 1);
      comparison(1).seriestemp1 := conditional(1).seriestemp1;
   @ then(1) :
      series(1).label1 := fn plus(conditional(1).label1, fn 2);
      series(1).temp1 := conditional(1).seriestemp1;
   @ else(1) :
      series(2).label1 := series(1).label2;
      series(2).temp1 := conditional(1).seriestemp1;
   @ fi(1) :
      conditional(1).seriescode1 := fn concat(
         comparison(1).code1,
         series(1).code2,
         fn make_op(fn J, fn label(fn plus(series(2).label2, fn 2))),
         fn make_op(fn label(
            fn plus(series(2).label2, fn 1)), fn LAB),
         series(2).code2,
         fn make_op(fn label(
            fn plus(series(2).label2, fn 2)), fn LAB));
      conditional(1).label2 := series(2).label2;

[conditional, if, comparison, then, series, fi] -->
   @ if(1) :
      comparison(1).label1 := fn plus(conditional(1).label1, fn 1);
      comparison(1).seriestemp1 := conditional(1).seriestemp1;
   @ then(1) :
      series(1).label1 := fn plus(conditional(1).label1, fn 1);
      series(1).temp1 := conditional(1).seriestemp1;
   @ fi(1) :
      conditional(1).seriescode1 := fn concat(comparison(1).code1,
         series(1).code2,
         fn make_op(fn plus(
            fn label(series(1).label2), fn 1), fn LAB));
      conditional(1).label2 := series(1).label2;

[assign, variable, colon, equal, expression] -->
   @ equal(1) :
      expression(1).temp1 := assign(1).seriestemp1;
   @ expression(1) :
      assign(1).seriescode1 := fn append(expression(1).code1,
         fn make_op(fn STO, variable(1).tag1));

[output, write, var_list] -->
   @ write(1) :
      var_list(1).p13n1_p12n1_opcode1 := fn PUT;
   @ var_list(1) :
      output(1).seriescode1 := var_list(1).code1;

[input, read, var_list] -->
   @ read(1) :
      var_list(1).p13n1_p12n1_opcode1 := fn GET;
   @ var_list(1) :
      input(1).seriescode1 := var_list(1).code1;

[statement, indef_loop] -->
   @ statement(1) :
      indef_loop(1).label1 := statement(1).label1;
      indef_loop(1).seriestemp1 := statement(1).seriestemp1;
   @ indef_loop(1) :
      statement(1).label2 := indef_loop(1).label2;
      statement(1).seriescode1 := indef_loop(1).seriescode1;

[statement, def_loop] -->
   @ statement(1) :
      def_loop(1).label1 := statement(1).label1;
      def_loop(1).seriestemp1 := statement(1).seriestemp1;
   @ def_loop(1) :
      statement(1).label2 := def_loop(1).label2;
      statement(1).seriescode1 := def_loop(1).seriescode1;

[statement, conditional] -->
   @ statement(1) :
      conditional(1).label1 := statement(1).label1;
      conditional(1).seriestemp1 := statement(1).seriestemp1;
      @ conditional(1) :
         statement(1).label2 := conditional(1).label2;
         statement(1).seriescode1 := conditional(1).seriescode1;

   [statement, assign] -->
      @ statement(1) :
         assign(1).seriestemp1 := statement(1).seriestemp1;
      @ assign(1) :
         statement(1).seriescode1 := assign(1).seriescode1;

   [statement, output] -->
      @ output(1) :
         statement(1).seriescode1 := output(1).seriescode1;

   [statement, input] -->
      @ input(1) :
         statement(1).seriescode1 := input(1).seriescode1;

   [series, series, semi, statement] -->
      @ series(1) :
         series(2).label1 := series(1).label1;
         series(2).temp1 := series(1).temp1;
      @ semi(1) :
         statement(1).seriestemp1 := series(1).temp1;
         statement(1).label1 := series(2).label2;
      @ statement(1) :
         series(1).code1 := statement(1).seriescode1;
         series(1).code2 := fn concat(series(2).code2,
            statement(1).seriescode1);
         series(1).label2 := statement(1).label2;

   [series, statement] -->
      @ series(1) :
         statement(1).seriestemp1 := series(1).temp1;
         statement(1).label1 := series(1).label1;
      @ statement(1) :
         series(1).code2 := statement(1).seriescode1;
         series(1).label2 := statement(1).label2;

   [program, series] -->
      @ program(1) :
         series(1).label1 := fn 0;
         series(1).temp1 := fn 0;
      @ series(1) :
         program(1).code1 := fn append(series(1).code2, fn HALT);

end_traverse
```

Bibliography

[1] *The* UNIX^{TM} *System User's Manual*. AT&T Information Systems, Englewood Cliffs, New Jersey, 1986.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[3] Henk Alblas. One-pass transformations of attributed program trees. *Acta Informatica*, 24:299–352, 1987.

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

[5] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. Programming environments based on structured editors: the MENTOR experience. In *Interactive Programming Environments*, pages 128–140, McGraw-Hill, 1985.

[6] R. Farrow and D. Yellin. A comparison of storage optimizations in automatically-generated attribute evaluators. *Acta Informatica*, 23:393–427, 1986.

[7] E. T. Irons. A syntax-directed compiler for ALGOL 60. *Communications of the ACM*, 4(1):51 55, 1961.

[8] E. T. Irons. Towards more versatile mechanical translators. In *Experimental Arithmetic, High Speed Computing and Mathematics (Proceedings of Symposia in Applied Mathematics 15)*, pages 41–50, American Mathematics Society, Providence, RI, 1963.

[9] M. Jazayeri. A simpler construction showing the intrinsically exponential complexity of the circularity problem for attribute grammars. *Journal of the ACM*, 28(4):715–720, 1981.

[10] M. Jazayeri, W. F. Ogden, and W. C. Rounds. On the complexity of the circularity test for attribute grammars. In *Conference Record of the Second ACM Symposium on Principles of Programming Languages*, pages 119–129, Association for Computing Machinery, New York, 1975.

[11] M. Jazayeri and K. G. Walter. Alternating semantic evaluator. In *Proceedings of the ACM National Conference*, pages 230–234, Association for Computing Machinery, New York, 1975.

[12] U. Kastens. Ordered attribute grammars. *Acta Informatica*, 13(3):229–256, 1980.

[13] U. Kastens, B. Hutt, and E. Zimmermann. Gag: a practical compiler generator. In *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1982. Number 141.

[14] K. Kennedy and S. K. Warren. Automatic generation of efficient evaluators for attribute grammars. In *Conference Record of the Third ACM Symposium on Principles of Programming Languages*, pages 32–49, Association for Computing Machinery, New York, 1976.

[15] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127 146, 1968.

[16] D. E. Knuth. Semantics of context-free languages: correction. *Mathematical Systems Theory*, 5:95–96, 1971.

[17] L. J. Morell. *OPTIMAL: A Parser Generator Interface.* Technical Report, College of William and Mary, Williamsburg, 1985.

[18] Robert E. Noonan and W. Robert Collins. *The Mystro System, Version 7.8: Parser Generator User's Guide.* Technical Report, College of William and Mary, Williamsburg, Virginia, 1985.

[19] Frank G. Pagan. *Formal Specification of Programming Languages.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[20] K. Raiha and M. Saarinen. An optimization of the alternating semantic evaluator. *Information Processing Letters*, 6(3):97–100, 1977.

[21] K. Raiha and J. Tarhio. A globalizing transformation for attribute grammars. In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, pages 74 84, Association for Computing Machinery, New York, 1986.

[22] William M. Waite and Gerhard Goos. *Compiler Construction.* Springer-Verlag, New York, 1984.

VITA

Randall P. Meyer

Born in Oak Park, Illinois, September 1, 1961. Graduated from Lee-Davis High School in Mechanicsville, Virginia, June 1979. Earned B.S. in Biology from the College of William and Mary in Virginia, December 1982. Completed M.S. in Computer Science, William and Mary, May 1985. Ph.D. with Computer Science concentration expected from William and Mary, May 1988.

Current research interests include techniques for automated compiler generation, natural language, automatic programming, and neural networks.