DEFINITION AND USE OF ATTRIBUTE REEVALUATION IN ATTRIBUTED GRAMMARS

by

Stephen Karl Skedzeleski

Computer Sciences Technical Report #340

October 1978

DEFINITION AND USE OF ATTRIBUTE REEVALUATION IN ATTRIBUTED GRAMMARS

BY

STEPHEN KARL SKEDZELESKI

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

DOCTOR OF PHILOSOPHY

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

1978

DEFINITION AND USE OF ATTRIBUTE REEVALUATION
IN ATTRIBUTED GRAMMARS

by Stephen Karl Skedzeleski
Under the supervision of Assistant Professor Charles N. Fischer

ABSTRACT

Attributed grammars can be used to specify both the syntax and the semantics of programming languages, and provide an implementation for compilers. We discuss extensions to attributed grammars and a generalization of attribute evaluation that allow code optimization techniques to be efficiently implemented and easily specified.

A formal definition of the extended attributed grammars, called time-varying attributed grammars, is given. Time-varying attributed grammars provide a high-level, non-procedural specification of iterative algorithms. Evaluators for time-varying attributed grammars are described and potential problems that can result from their use are discussed. Examples of common code optimization techniques are given using time-varying attributed grammars. Techniques that reduce the overhead of evaluation, and also simplify the specification of attributed grammars are suggested.

ACKNOWLEDGEMENTS

My first thanks go to my advisor, Charles Fischer. He provided interesting ideas, discussion and criticism that shaped this work. His careful reading of endless drafts helped turn my ideas into a thesis.

Marvin Solomon gave me many insights into the details of my ideas, and a way of expressing them.

My fellow graduate students, in particular Bruce Rowland, Frank Horn, Marty Honda, and Bernard Dion, were a source of many thought-provoking discussions.

More personally, I thank my friends who have made this effort bearable. In particular, Vanessa Skedzeleski has given me support and kindness for the last four years. Cathy Ballard and Frank Horn have been especially important during the final, hectic months.

Finally, I thank the Mathematics Research Center of the University of Wisconsin for their fellowship support.

TABLE OF CONTENTS

TABLE OF FIGURES

Chapter 1: INTRODUCTION

The specification of programming languages is an important issue to their designers, users and implementors. Designers must have a clear definition mechanism in which to state and evaluate their ideas. Users should be able to understand both the syntax and meaning of a programming language, and implementors would like a descriptive technique that can be automatically transformed into a compiler. This thesis is an attempt to improve both the specification and the implementation of programming languages.

1.1 Background

The development of Backus-Naur Form (BNF) [Naur 63] constituted a major advance towards the automatic production of compilers, because it separated the syntax from the semantics of a programming language. Numerous parser generators exist which translate BNF into tables that drive a simple parsing subroutine. The BNF can be easily changed and the tables regenerated, thus facilitating syntax modification. Similarly, scanners can be specified using regular

expressions, and translated into tables that drive a scanning routine [HU 77]. Such generators remove a tedious and error-prone part of compiler construction and leave the implementer free to concentrate on other areas, notably semantic considerations and program optimization.

BNF is not adequate to specify the semantics of a programming language, since such semantic requirements as type compatibility are context-sensitive, and BNF can only describe context-free languages [HU 69]. Attributed grammars were introduced by Knuth in [Knu 68] as a method that extends BNF to include context-sensitive information. Attributed grammars are most often used to specify language semantics, but they can be used in other areas that require context-sensitive information. [Mil 77] describes a parsing method that uses attributes to choose a particular parse tree when an ambiguous context-free grammar has been specified.

In syntax-directed compilation, a compiler is "driven" by its parser, which invokes semantic routines as phrases are recognized. The semantic routines use syntactic information to manipulate data, check for type compatibility, and generate code. They are usually hand-coded in a traditional programming language, and this writing requires the bulk of the time spent in implementing a non-optimizing compiler.

Optimization is often a separate pass over the compiler output, and again is usually coded in a traditional programming language.

We wish to replace the hand-coding of semantics and program optimization by the use of attributed grammars. Given the specification of an attributed grammar, one must be able to take strings from the language that it specifies, build the parse tree that represents its context-free structure, and assign values to the attributes that are associated with each symbol of the parse tree. This process is called evaluation. An example of the specification of an attributed grammar is shown in Figure 1.1.

```
1. S ::= a + b
    1.1 S.sum = a.value + b.value
    1.2 a.value = 2
    1.3 b.value = 5

I(S) = Ø        I(a) = Ø        I(b) = Ø
S(S) = {sum}    S(a) = {value}  S(b) = {value}
```

Figure 1.1: Simple Attributed Grammar

The first line specifies the production in the context-free grammar; the next three lines give the attribute definitions. The set definitions at the end associate attributes with the symbols each symbol having two sets of attributes, I and S. One possible evaluation order is a.value, b.value and S.sum; the other is b.value, a.value and S.sum. The evaluator needs to determine that a.value and b.value must

be evaluated before S.sum.

The parse tree is easily built using the parser generated from the BNF. Evaluation can be a complicated task, since attribute dependence can occur among sibling nodes, and in both directions between parent and child nodes. Early work established that it was possible to write an evaluator for attributed grammars [Fan 72] and to completely specify a high-level language (SIMULA 67) by means of attributes [Wil 71]. Since then, others have developed evaluation methods that reduce the time or space needed to evaluate attributed grammars. Some of these other methods will be discussed in Chapter 2. For a more complete introduction to attributed grammars, consult [Rai 77], which contains a good introduction to attributed grammars as well as an extensive bibliography.

While attributed grammars have been successfully used to implement the compile-time (or static) semantics of programming languages, they have received comparatively little use in specifying code optimizations [Bab 77], [NAM 74]. Attributed grammars are a poor vehicle for writing iterative algorithms (since an attribute is only evaluated once), and iterative algorithms are pervasive in code optimization literature [Kil 73], [Sch 73], [Ken 72], [AU 77]. If iterative methods were to be used, they would have to occur

within a function that operates on attributes that are passed to an appropriate common ancestor node. The results of the iteration could be broadcast back down the parse tree to the appropriate nodes, for use in code generation.

We wish to be able to exploit the fact that the structure of a programming language (represented by a parse tree) contains much information about its flow of control. The parse tree is often discarded by the compiler before the code optimization process begins. Any flow information contained in the parse tree is lost, and must be rediscovered before code optimization algorithms can begin.

1.2 Time-varying Attributed Grammars

This thesis describes an extension to attributed grammars, called time-varying attributed grammars, that facilitates the specification of iterative algorithms. The time-varying attributes are given initial values that can be refined during the course of attribute evaluation. The specification of such attributes is non-procedural; the user specifies only the relation among attributes (e.g. A.set = A.set $\wedge$ B.set, where $\wedge$ represents set intersection), and

omits the instructions about how iteration is accomplished. The evaluator for time-varying atributes is more complicated than previous evaluators since it must schedule both initial and iterative evaluations of attributes. It must also be able to "remember" attribute dependencies after the initial evaluation has taken place so that changes in attribute values will propagate around the parse tree in accordance with the attribute definitions. An example of the specification of a time-varying attributed grammar is shown in Figure 1.2.

```
1. S ::= A B
     1.1 S.span = A.len + B.len
     1.2 A.len = 15
     1.3 if A.len > 10 then B.len = 2 fi

I(S) = Ø            I(A) = Ø            I(B) = {len=1}
S(S) = {span}       S(A) = {len}        S(B) = Ø
```

Figure 1.2: Simple Time-varying Attributed Grammar

An initial value has been given to B.len by writing len=1 in the specification of I(B). The reevaluation process will change B.len to 2 in accordance with definition 1.3, and S.span will have the value 17 at the end of the evaluation.

This thesis will also show that code optimization algorithms can be specified using flow information derived from the parse tree, and that such a specification can be efficiently implemented. Evaluators for time-varying attributes build upon the work done by others on traditional, or static

attribute evaluation. Specification of program optimizations using time-varying attributed grammars eliminates the overhead of flow analysis that usually preceeds the optimization process.

Chapter 2 contains definitions needed to discuss attributed grammars and a discussion of the advantages and disadvantages of time-varying attributed grammars. Chapter 3 describes the mechanism to evaluate time-varying attributes, and Chapter 4 gives two evaluators for time-varying attributed grammars. Chapter 5 discusses a notational extention that offers clearer and more efficient specifications of attributed grammars, and Chapter 6 gives several optimization algorithms using time-varying attributed grammars. Chapter 7 is a summary and contains suggestions for future research.

Throughout this thesis the following notational conventions will be used. Within a chapter, figures will be consecutively numbered, and definitions, lemmas, theorems and corollaries will receive a consecutive numbering that is independent of the numbering of figures. $\in$ and $\subseteq$ will represent set inclusion and set containment, and $\boxtimes$ will denote the end of a proof. Set union, set intersection and cross products are written $\cup$, $\wedge$, and $\times$, respectively. Overbars will denote the complement of Boolean vectors.

Chapter 2: ATTRIBUTED GRAMMARS

An attributed grammar may be thought of as a context-free grammar in which each grammar symbol possesses a fixed set of attributes. Each production has a set of attribute definitions which specify relations among the attribute values of its symbols. The attributes of an instance of a grammar symbol are given values that represent information about it. For example, the terminal symbol CONSTANT might have two attributes, CONSTANT.type and CONSTANT.value. Attributes can range over finite or infinite sets of values. Type, for example, could have the range {real,int,boolean,character}, while value might range over all integers.

2.1 Formal Definitions

We begin with a discussion of context-free grammars. We will follow the notational conventions of [AU 77 p.128].

Definition 2.1

A context-free grammar is a 4-tuple $G = (N,T,P,Z)$ where

- N is a finite set of non-terminal symbols.
- T is a finite set of terminal symbols, disjoint from N.
- P is a finite subset of $N \times (N \cup T)^*$, whose elements are called productions. A pair $\langle X_0, X_1 \ldots X_n \rangle$ in P will be denoted "$X_0 ::= X_1 \ldots X_n$". X_0 is called the left-hand side (LHS) of the production and $X_1 \ldots X_n$ is called the right-hand side (RHS). For convenience, we will assign an arbitrary but fixed numbering to the productions, so that we may refer to P^i, LHS^i and RHS^i, for $i = 1 \ldots |P|$.
- Z is a distinguished element of N, called the start symbol.

Definition 2.2

If $\alpha, \gamma \in (N \cup T)^*$ and $A ::= \beta \in P$, then we say that $\alpha A \gamma$ directly derives $\alpha\beta\gamma$, written $\alpha A \gamma \Longrightarrow \alpha\beta\gamma$. The reflexive transitive closure of $\Longrightarrow$ is denoted by $\overset{*}{\Longrightarrow}$. The language generated by G is

$$L(G) = \{w \mid w \in T^* \text{ and } Z \overset{*}{\Longrightarrow} w\}.$$

Definition 2.3

A symbol A in $N \cup T$ is useless if there does not exist a derivation of the form $Z \overset{*}{\Longrightarrow} xAz \overset{*}{\Longrightarrow} xyz$, where $x,y,z \in T^*$. The grammar G is reduced if $N \cup T$ contains no useless symbols.

We will only consider reduced grammars.

Context-free grammars have been studied in much detail. Properties of context-free grammars and parsing methods based on them can be found in [HU 69], [AU 73] and [AU 77]. Semantics can be added to a context-free grammar by the use of semantic attributes. The following notation is that of Räihä [Rai 77], with some modifications.

Definition 2.4

A specification of attributes A_G for a grammar G is a four-tuple $A_G = (S,I,R,V)$ where

- S is a finite set of synthesized attribute symbols disjoint from N and T.
- I is a finite set of inherited attribute symbols, disjoint from N, T and S.
- R is a collection of the sets of allowed attribute values.
- V is a mapping from $S \cup I$ to R, the range function.

Definition 2.5

An attribute association for G and A_G is a mapping A from $N \cup T$ to $2^{(S \cup I)}$. We define

$$S(X) = A(X) \wedge S \quad \text{for all } X \in (N \cup T)$$

$$I(X) = A(X) \wedge I \quad \text{for all } X \in (N \cup T)$$

Attributes are divided into two disjoint classifications called inherited and synthesized. The values of the synthesized attributes of a node in a parse tree are functions of attribute values found in its immediate subtrees. Thus, synthesized attributes represent data flow out of a subtree. Knuth [Knu 68] introduced inherited attributes, whose values are functions of the attributes of their parent or sibling nodes. Inherited attributes allow data flow into a subtree. Definitions of synthesized attributes appear when their associated grammar symbol is a LHS of the production; definitions of inherited attributes appear when their associated grammar symbol is in a RHS of the production.

Some authors (e.g. [Rai 77]) place restrictions on the kinds of attributes that may be associated with grammar symbols, particularly attributes of terminal symbols. Terminal symbols may not be allowed any attributes [Rai 77], may be allowed only inherited attributes [Knu 68], or may be allowed only synthetic attributes [KW 76]. We allow terminal symbols to have both synthesized and inherited attributes. As we shall see, this lack of restrictions fits in well with the general idea of time-varying attributes.

We need to be able to refer to a given attribute definition. Definitions are given an arbitrary but fixed numbering of the form production.definition, and the attributes associated with each grammar symbol are also given a fixed ordering. $X_j^p.a_k$ denotes the attribute a_k of symbol X_j in production p. We will define $X_j^p.a_k$ in terms of the values of attributes in the current production. These attributes are denoted $(h_1, \ldots, h_t)$, where t depends on the definition. More formally:

$$X_j^p.a_k = f_{pjk}(h_1, \ldots, h_t) \text{ where } t = t(p,j,k).$$

Definition 2.6

An attributed grammar GA is a four-tuple (G, A_G, A, sem) where

1) $G = (N,T,P,Z)$ is a reduced context-free grammar
2) $A_G = (S,I,R,V)$ is a specification of attributes
3) A is an attribute association for G and A_G
4) sem is a collection of attribute definitions

Definition 2.7

A time-varying attributed grammar is an attributed grammar in which each attribute occurrence may be given an initial value that can be changed during the course

of the evaluation.

Definition 2.8

A parse tree is a tree that represents the derivation of some element of a context-free language. Let G = (N,T,P,Z) be a context-free grammar. A parse tree for an element of L(G) is a tree such that:

1. Every node has a label $\in N \cup T$.
2. The root node is labeled Z.
3. If a node n has at least one descendant other than itself and has the label A, then A must be an element of N.
4. In nodes $n_1\ n_2 \ldots n_k$ are the direct descendants of node n (in order from the left) with labels $A_1\ A_2 \ldots A_k$, respectively, then $A ::= A_1A_2\ldots A_k$ must be a production in P.

Definition 2.9

An attributed parse tree for an attributed grammar G is a parse tree for the underlying context-free grammar in which each node has value assigned to its associated attributes. The values of all attributes must be consistant with the attribute definitions of G in the following sense: For each production $X_\emptyset ::= X_1\ldots X_n$, the associated attribute definitions hold for each node

labelled $X_\emptyset$ with offspring $X_1, \ldots, X_n$, using attribute values found at there nodes.

There is a further classification that concerns the use of attribute instances that is analogous to the L-modes and R-modes of programming languages.

Definition 2.10

An attribute instance is a recipient if it is the target of an attribute definition. Given a production p,

$$X_\emptyset ::= X_1 \ldots X_n$$

the set of recipient attribute instances in p is

$$Rec(p) = \{X_i.a_j \mid i = \emptyset \text{ and } a_j \in S(X_\emptyset) \text{ or } 1 \leq k \leq n \text{ and } a_j \in I(X_k)\}$$

An attribute instance is a donor if it appears as an argument of an attribute definition function. The set of donor attribute instances for production p is

$$Don(p) = \{X_i.a_j \mid X_i.a_j \text{ is an argument to some } f_{pkm}\ \emptyset \leq i,k \leq n, \text{ where } a_j \text{ is an attribute of } X_i, \text{ and } a_m \text{ is an attribute of } X_k\}$$

A restriction suggested by Chirica and Martin [CM 76] requires that Don(p) and Rec(p) be disjoint. In non-time-varying applications this restriction causes no loss of expressive power, but it is unacceptable when we want to specify iteration. Therefore, Don(p) and Rec(p) may (and

usually will) overlap in time-varying attributed grammars.

Attribute definitions induce a relation on the attribute occurrences called dependency. An attribute occurrence $X_i.a_j$ directly depends on another attribute occurrence $X_k.a_m$ if $X_k.a_m$ appears as a donor in the definition of $X_i.a_j$. We may also define indirect dependence as the transitive closure of direct dependence. An attributed grammar is circular if in some parse tree of an element of the language, an attribute occurrence is indirectly dependent upon itself. In attributed grammars used with static evaluators, circularity implies that an error has occurred in the specification. In time-varying attributed grammars, circularity is allowed, and will often be very useful.

2.2 Attribute Availability

An attribute occurrence in a parse tree is said to be available (for use) if it has been given a value. To assign values to attributes, an evaluator must be able to find a starting point for an evaluation and an order of attribute evaluation. It is essential to know which attributes have been given values at any point in the course of an evaluation, since an attribute can be evaluated only when all donors are available. Several different evaluation methods have been suggested. Jazayeri and Walter [JW 75] describe an evaluator that makes both left-to-right and right-to-left passes over the parse tree. Kennedy and Warren [KW 76] give an algorithm for calculating the order of evaluation of attributes at a given node in the parse tree (called plans) while analyzing the grammar, so that there is little overhead required while analyzing a particular program. There are evaluation methods that interleave parsing and attribute evaluation (called "on-the-fly" evaluators), which decrease the overhead of parse tree storage. [LRS 74] and [Row 77] describe on-the-fly evaluators that work with different parsing methods. Parts of the parse tree can then be discarded when all attributes in a subtree have been evaluated. Some methods place restrictions on the types of definitions allowed ([LRS 74], [KW 76]). Fang [Fan 72] required the user to place a partial ordering on the definitions (reflecting the attribute dependence among them) while others simply required non-circular definitions ([Row 77], [Sch 76], [JW 75]). In all cases, a total evaluation order is obtained. It is from [Row 77] that we will borrow the definitions for expressing attribute interdependence, since those definitions will be useful in the extentions needed to allow time-varying attributes.

Each instance of a production in the parse tree will have two Boolean vectors associated with it as evaluation proceeds: one for the donor attributes and one for the recipient attributes. Each donor attribute is assigned a position in one vector; each recipient a place in the other. A one in the donor vector indicates that the attribute has been evaluated; a zero indicates that it has not been evaluated. A one in the recipient vector indicates that the attribute is is ready for evaluation; a zero indicates that at least one donor is missing.

Definition 2.11

The Boolean vectors $\underline{daav}_p$ (donor attribute availability vector) and $\underline{raav}_p$ (recipient attribute availability vector) represent the guaranteed availability of donor attributes and the potential availability of the recipient attributes, respectively, for an instance of production p.

A Boolean matrix can be used to represent the dependence of recipient attributes upon donor attributes for each production and its associated definitions.

Definition 2.12

MD_i is the dependency matrix for production i. MD_i is of dimension $|Rec(i)| \times |Don(i)|$. A row in the matrix represents the dependence of a recipient attribute upon its donors.

$$MD_i(m,n) = \begin{cases} = 1 & \text{if the } m^{th} \text{ recipient attribute } (X_j.a_k) \text{ uses the } n^{th} \text{ donor attribute in the associated definition function } f_{ijk}. \\ = 0 & \text{Otherwise} \end{cases}$$

The dependency matrix is equivalent to the dependency graph of [Knu 68] where ones in the matrix represent directed arcs in the graph from the donor to the recipient.

Finally, the equation that relates $raav_i$, $daav_i$ and MD_i (with overbars to denote bitwise complement) is

$$(2.1)\quad \overline{raav}_i = MD_i \times \overline{daav}_i$$

This represents the fact that an attribute is <u>not</u> ready to be evaluated if any of its donors has not been evaluated.

2.3 Time-varying Attributes

Implicit in the previous definitions, and almost all previous work on attributed grammars, is the assumption that attributes never change value once they have been evaluated. This assumption is pervasive in the literature; only Babich [Bab 77] speaks of attributes that change value. We will show that previous work can be readily extended to include time-varying attributed grammars, and that time-varying attributed grammars can directly specify algorithms that would have otherwise been hidden away in a function call of an attribute definition. We must emphasize that time-varying attributes will not increase the computational power of attributed grammars, which are known to be equivalent to Turing Machines. However they will often increase the attributed grammar's efficiency and readability.

We wish to provide a method to specify iterative algorithms on attributes. As noted above, we first remove the restriction that prohibits circularity in the attribute definitions since circularity is implicit in iteration.

Iterative algorithms commonly require a starting point for calculations, so we will specify initial values for some attributes. The scanner will provide initial values to both

synthetic and inherited attributes of terminal symbols. The inherited attributes of terminal symbols will not destroy the logical separation between the scanner and the rest of the compiler, since the definition of an inherited attribute of a terminal symbol will provide an update to a value initially provided by the scanner.

One of the goals of this work is to provide the specification of an iterative algorithm non-procedurally, and have the iteration take place automatically. Should an attribute change value, all attributes that depend on it will be automatically reevaluated. Reevaluation is distinct from initial evaluation, and the initial evaluation of an attribute lacking an initial value will not invoke the automatic reevaluation procedure. No explicit control structure is needed to guide the iteration.

In much of the literature on attributed grammars the term "evaluation rule" is used in place of our term, "definition". Our use of "definition" is meant to remove the suggestion that we are writing assignment statements. Although the definitions resolve to simple assignments in the static case, they are really statements of equality that must hold at the end of the evaluation. A user specifies the relation among attributes, not the mechanism used to effect it. (Of course, care must be taken to find the

correct relations and the correct initial values for the purpose!)

Automatic reevaluation is described in [Bab 77], where multiple passes are made over an entire parse tree. A check is made after each pass to see if any attributes have changed value. If a change has occurred, another pass is made; if not, the evaluation is complete. A drawback of this method is that it potentially takes quadratic time in the length of the input (at worst one pass for each new attribute evaluation). Babich describes two optimization algorithms; each is guaranteed to halt after two passes over the parse tree (a third pass would be redundant). Unfortunately, one evaluation must be done with left-to-right passes, and the other with right-to-left passes over the parse tree, so four passes are needed to effect both evaluations. Our approach will avoid passes over the entire parse tree by following a chain of changed values. A change in the value of an attribute has the potential to cause the reevaluation of all attributes that depend on it. Changes in value will first occur when an attribute with an initial value is reevaluated according to its definition and the new value is different from the old. At the time of the change, all attributes that depend on the attribute with the changed value are queued for reevaluation. A procedure is invoked to remove attributes from the queue and reevaluate them.

Since one change may cause another, the reevaluation may cascade, but at some point the queue should empty and normal evaluation can resume. There is no notion of a "direction" of reevaluation, so there is no need for forward and backward passes over the tree.

This reevaluation mechanism looks only at those attributes that are affected by changed attribute values; attributes not affected are not reevaluated. As long as the overhead of the reevaluation mechanism is small, this implementation of time-varying attributed grammar evaluation should compare favorably to static evaluation methods. Examples of two evaluators of time-varying attributed grammars are given in the next chapter.

2.4 Hazards of Time-varying Attributes

Attributes with time-varying values introduce several changes to the behavior of attributed grammars. The obvious change is a beneficial one: they allow more freedom in the semantic specification. On the other hand, some bad side effects can arise in their use:

a) Non-terminating algorithms can be specified.

b) Results may depend on the initial values.

c) Results may depend on the evaluation order.

These problems already exist in standard attributed grammars. The first two problems were hidden in the functions used in the attribute definitions; the last problem was implicit in the evaluation mechanisms.

As an example of a non-terminating algorithm, consider the following example.

```
1. S ::= A
     1.1 A.value = S.value+1
     1.2 S.value = A.value+1

I(S) = {value}        I(A) = {value=1}
S(S) = Ø              S(A) = Ø
```

Figure 2.1: Specification of a Non-terminating Algorithm

The initialization of A.value is given in the definition of I(A). This specification of attributes will lead to an unlimited number of evaluations of both A.value and S.value; no solution exists. This problem is not unique to time-varying attributes; static attribute definitions can have functions that never terminate.

Poor starting values for an iterative algorithm may cause an algorithm to converge to a wrong solution, or may cause divergence. For example, consider deciding when it is

possible to use the short form of an instruction that has both a short and long format. We wish to use short forms whenever possible, but they can only be used if the addressed object lies within a specified displacement (-MAX $\leq$ displacement $\leq$ MAX).

If we initialize all instructions to the long format and shorten them whenever possible, we will not necessarily get a solution using the minimal number of long format instructions ([Szy 76], [Rob 77]).

```
B: --------}
   --------}  MAX-1 locations
   --------}
a: ref A
b: ref B
   --------}
   --------}  MAX-1 locations
   --------}
A: --------
```

Figure 2.2: Poor Starting Values

If a and b are initially of length 2, we have a solution that satisfies all constraints, since neither a nor b can reference their targets in short form. Actually, we would like the algorithm to converge to a solution in which both a and b have length 1, but the initial values have caused convergence to a sub-optimal solution. If the algorithm begins with a and b in short form (length 1), we get the minimal number of long instructions.

The same problem can occur in standard attributed grammars. A function may be used that has the property that it will converge to different results depending on the initial conditions that it uses.

The lack of control over the order of attribute evaluation and reevaluation might seem to be a significant change from standard attributed grammars; in practice it is not. Any static evaluator uses a partial ordering of attributes (derived from attribute dependency) to schedule attribute evaluations. In most cases the ordering is not unique, so one of several orderings is chosen by the evaluator, not the user. It is assumed that any evaluation order obeying the partial ordering is acceptable. Unless the definitions use only pure functions (functions without side-effects) the result may be sensitive to the evaluation order. Examples of non-pure functions are a function that appends generated code to the end of a file and a function that generates the next free address for instructions or data.

There are cases when non-determinacy of results is acceptable; code generated for a labelled case statement is an example. The order in which code for each case is generated is irrevelant as long as the proper case is selected at execution time.

There are, of course, times when non-determinacy is not acceptable. It is up to the user to insure that all possible partial orderings produce acceptable results.

The same problem exists for users of time-varying attributed grammars. An example of a specification that depends on the order of evaluation follows. The non-determinacy in this example is similar to a race condition in an operating system; there are two time-varying attributes racing to determine the value of S.result.

```
1. S ::= A B
     1.1 S.result = MERGER(A.input,B.input)
     1.2 A.go = (S.result ≤ 10)
     1.3 B.go = (S.result ≥ 20)
     1.4 A.input = if A.go then 10 else NIL fi
     1.5 B.input = if B.go then 20 else NIL fi

I(S) = Ø              I(B) = {go=true,input=NIL}
S(S) = {result}       S(B) = Ø

I(A) = {go=true,input=NIL}
S(A) = Ø
```

Figure 2.3: A Race Condition

MERGER is a function that has been defined to choose a value from A.input and B.input; the non-NIL value is returned. In this example, the order of evaluation can effect the result. If A.input is evaluated before B.input, S.result gets 10. If B.go is evaluated next, B.input remains at NIL, and a stable solution has been found. Otherwise MERGER has two non-NIL arguments and gives an unspecified result. If B.input is evaluated before A.input, S.result gets 20.

Assuming that A.go is evaluated next, A.input remains NIL and another stable solution has been found. Otherwise MERGER has the same two non-NIL arguments and again gives an unspecified result.

The previous example might suggest that we should check for definitions that are functions of two or more time-varying attribute donors. Such a check can provide a warning that a potential evaluation-order-dependent result exists, but the following example shows that this restriction is not sufficient.

```
1. A ::= B
     1.1 A.a = Max(B.b,1)
     1.2 B.b = Min(A.a,2)

I(A) = Ø          I(B) = {b=2}
S(A) = {a=1}      S(B) = Ø
```

Figure 2.4: Evaluation-Order-Dependent Results

If A.a is evaluated before B.b, A.a gets the value 2 and both definitions are satisfied (A.a = B.b = 2). If B.b is evaluated first, B.b gets the value 1 and we have a solution with A.a = B.b = 1.

2.5 Correctness

While we cannot guarantee that a user will not unwittingly specify a time-varying attributed grammar that exhibits evaluation-order-dependent results, we can discuss the correctness of an evaluator for time-varying attributed grammars. To do so requires that we discuss the intermediate results of an evaluation.

Definition 2.13

Given a time-varying attributed grammar G, an incompletely evaluated attributed parse tree for a program Prog in L(G) is a parse tree for Prog in G's underlying context-free grammar in which some attributes are assigned values.

Definition 2.14

Given a time-varying attributed grammar G, an initial-valued attributed parse tree for a program Prog in L(G) is a incompletely evaluated parse tree for Prog in which each node has an initial value for its associated attributes given by the scanner or the specification of G.

Definition 2.15

Given a time-varying attributed grammar G, a step in the evaluaton of a program Prog in L(G) is the transformation of one incompletely evaluated attributed parse tree of Prog (T) to another (T') as follows: a new and different value, v, for some attribute, A, of some symbol S in Prog is determined using the attribute values of T and the attribute definitions of G. T' is then obtained from T by replacing the old value of S.A with v.

After each evaluation step, the incompletely evaluated attributed parse tree has exactly one attribute value updated. An evaluator correctly evaluates a time-varying attributed grammar by making a sequence of steps that converges to an attributed parse tree.

Definition 2.16

Given a time-varying attributed grammar G and a program Prog in L(G), and evaluator E correctly evaluates Prog if, starting with the initial-valued attributed parse tree for Prog, E iteratively applies evaluation steps until an attributed parse tree is obtained (i.e., no more steps are possible.)

Thus an evaluator may choose any evaluation order that obeys the partial ordering due to attribute dependence. If the

evaluator halts, all attribute values will satisfy the attribute definition rules. Further, the final attribute values will have been obtained directly from the initial attribute values provided. The solution obtained may or may not be unique, depending on the specification provided. In the same vein, convergence or divergence of the evaluator is entirely dependent on the specifications.

Chapter 3: REEVALUATION OF ATTRIBUTES

Attribute definition rules can be viewed as a system of equations that must be satisfied when a translation is completed. (This implies that we deal only with pure functions, so that the same arguments will always yield the same results.) An important tenet of this chapter is that an attribute shall be reevaluated if and only if the value of one of its donor attributes has changed. This approach differs from the method suggested in [Babich 1977], in which all attribute values are recomputed as multiple passes are made over the entire parse tree. Our reevaluation method will sharply reduce the number of attributes evaluated during any translation, assuming that most of the changed attribute values will have local effects.

As an example of the use of time-varying attributes, consider the problem of generating code for a computer that has two forms of instructions: long and short. We wish to use the short form whenever possible, but it can only be used when the operand address is within some address range (current-MaxDisp,current+MaxDisp), where MaxDisp is fixed.

We will look specifically at the branch instructions in the PDP-11 instruction set, although this optimization can be used, to a lesser extent, with other kinds of instructions. The algorithm will attempt to use only short-format branches, but will expand them to long-format branches when necessary. Since this process may cascade when an expanded branch causes another branch to need the long format, it is a good example of the use of variable attributes.

The algorithm described by the attributed grammar is straightforward. The starting address for the first instruction is passed down the parse tree by "start"; the starting addresses for following instructions are passed along the lists with the attribute "begin". Each instruction records the final address it uses in "end", which allows the next instruction to compute its starting address. A symbol table is maintained in the codesect node, and is passed down the tree via "table". Modifications to the table are passed up the tree in "update", which contains NIL (no changes), or a list of (label,address) pairs. If the label already exists in the table, its address field is changed to the new value in "update", else the pair is entered in the table. In the codesect node, the updated table is transferred to "list.table" and the new table becomes available to all instructions. The functions INSERT and ADR manipulate the table of (label,address) pairs: INSERT adds a pair (deleting any previous pair for the same label); ADR returns the address for a given label (we will assume that labels are unique). CAT is used to catenate an

update to a list of updates. The term "arbitrary" should be considered to be a constant whose value is not significant. Attribute changes begin when BR.len=2 occurs, which causes end to change, which causes begin to change, and so forth.

```
1. codesect ::= list
    1.1 codesect.table = INSERT(codesect.table,list.update)
    1.2 list.start = arbitrary
    1.3 list.table = codesect.table

2. list ::= list' inst
    2.1 list.end = inst.end
    2.2 list.update = CAT(list'.update,inst.update)
    2.3 list'.start = list.start
    2.4 list'.table = list.table
    2.5 inst.begin = list'.end + 1
    2.6 inst.table = list.table

3. list ::= inst
    3.1 list.end = inst.end
    3.2 list.update = inst.update
    3.3 inst.begin = list.start
    3.4 inst.table = list.table

4. inst ::= other  {non-branch instruction}
    4.1 inst.end = inst.begin + other.len - 1
    4.2 inst.update =
          if other.label = '' then NIL
          else (other.label,inst.begin)   fi

5. inst ::= BR
    5.1 inst.end = inst.begin + BR.len - 1
    5.2 inst.update =
          if BR.label = '' then NIL
          else (BR.label,inst.begin)   fi
    5.3 BR.offset = ADR(inst.table,BR.target) - inst.begin
    5.4 if abs(BR.offset) > Maxdisp then BR.len = 2 fi
```

Figure 3.1: Branch Length Optimization

Here is a list of the sets of inherited and synthesized attributes associated with each symbol, and a brief description of their purpose.


```
I(codesect) = Ø
S(codesect) = {table=empty}
    table is a table of (label,address) pairs that
          is passed down the tree (initially empty)

I(list) = {table,start}
S(list) = {end,update}
    end gives the last address used by this list of
        instructions.
    start is the start address for the first instruction
    update is a list of (label,address) entries to be
            made in table.

I(inst) = {begin,table}
S(inst) = {end,update}
    begin gives the starting address of the instruction

I(other) = Ø
S(other) = {label,len}
    label is the non-branch instruction's label
    len is the non-branch instruction's length

I(BR) = {offset,len=1}
S(BR) = {label,target}
    offset is the signed displacement of the branch relative
            to the current location
    len is the length of the branch instruction (initially 1)
    target is the label addressed by the branch
```

Figure 3.2: Attributes Used in Branch Length Optimization

Initialization of variable attributes is denoted in the declaration of the sets I() and S(). Iteration is accomplished on BR.len in 5.4, and on codesect.table in 1.1. Information flows both up and down the parse tree; list.update and inst.update send changes up to the root, while list.table and inst.table send information down to the leaves. In this example we have a very simple monotone property that guarantees that the iteration will halt. Once

len has the value 2, it will never change. This property is valuable in preventing oscillation of the value of BR.len. If BR.len were defined to be

5.4 BR.len = if ABS(BR.offset) > Maxdisp then 2 else 1 fi,

we could not be sure of finding a solution to the system of equations. BR.offset is the difference of ADR(inst.table,BR.target) and inst.begin. Given that the order of evaluation is uncertain, inst.begin could be incremented (due to another branch expanding to length 2) before BR.target was incremented to reflect the same change. ADR(inst.table,BR.target) - inst.begin could temporarily fall back within the range of a short instruction, and the evaluator might reevaluate BR.len, giving 1. This oscillation could continue indefinitely. We avoid this problem by writing a definition that can only change value a bounded number of times.

The preceding discussion should motive the need for care in the specification of iterative algorithms when one is using a non-procedural definition system.

3.1 An Evaluation Strategy

We will first consider an evaluator that requires that the entire parse tree be present (a treewalk evaluator). It will visit nodes (non-terminals) using depth-first search, evaluating some attributes, and deferring evaluation of others until all donor values are calculated. Deferred evaluations are later performed by either returning from a visit to a subtree or by scheduling another visit when an inherited attribute becomes available. To determine when an attribute is ready to be evaluated, we will use the Rowland's dependency matrix, recipient attribute availability vector, and donor attribute availability vector for production i (MD_i, $raav_i$, $daav_i$) as described in Chapter 2. As given above, the conditions that signal that an attribute is ready to be evaluated are:

$$(2.1) \quad \overline{raav}_i = MD_i \times \overline{daav}_i$$

where the overbars denote bitwise complementation. The conditions that signal that an attribute must be reevaluated are:

1) Its donors are all available

2) At least one of the donor values has "changed"

The first condition is determined as in (2.1) above. The second condition is determined by the product of the depen-

dency matrix and a vector of "changed" attributes. The resulting equation is:

$$(3.1) \quad rv = (MD_i \times cav) \wedge raav_i$$

where rv is the reevaluation vector, and cav is the change of attribute vector:

$$rv_j = \begin{cases} 1 & \text{iff the } j^{th} \text{ recipient must be reevaluated} \\ 0 & \text{otherwise} \end{cases}$$

$$cav_j = \begin{cases} 1 & \text{if the } j^{th} \text{ donor has changed value (since the last reevaluation of attributes at this node)} \\ 0 & \text{otherwise} \end{cases}$$

Reevaluation is necessary for all attributes that have a one in the corresponding element of rv. The evaluator must keep track of the nodes in which each attribute occurrence appears as a donor, in order to find the attribute occurrences that must be reevaluated. If an attribute occurrence changes value, a one is placed in the appropriate position of the cav in all nodes where the attributes occurrence is used as a donor. cav is cleared after (3.1) has been applied, to avoid redundant reevaluations if the node is visited again.

Whenever an attribute changes value, all of the attributes that must be reevaluated are added to a queue of pending evaluations. This queue is exhausted before the usual evaluation continues, following the strategy that attributes are (re)evaluated as soon as possible. This process is

interleaved with the evaluation steps in the static treewalk evaluator. Translation is complete when (and if) there are no more nodes to be visited in the parse tree. More precisely:

```
assign initial values to attributes;
repeat
  perform one attribute evaluation;
  if the attribute changed value then
    queue all definitions that must be reevaluated;
    while the reevaluation queue is non-empty do
        remove definition D from the queue;
        reevaluate D;
        if D has changed value then
            queue all definitions that must be
                reevaluated;
        fi
      endwhile
    fi
until evaluation is complete
```

Figure 3.3: Reevaluation Algorithm

Lemma 3.1

The while loop in Figure 3.3 repeatedly performs correct evaluation steps in the sense that:

1) All attributes whose donors have changed value are reevaluated.
2) Any further reevaluation is useless.
3) All values of time-varying attributes are consistant with their definitions.

Proof

Let A be an attribute whose value depends, directly or indirectly, on the value of a donor D. Call the chain of attribute definitions that link A and D a path. Let $A = x_n = f(\ldots,x_{n-1},\ldots), \ldots, x_1 = f(\ldots,D,\ldots)$ be a path of length n.

Part 1 is proved by induction on the length of the path from the changed value of the donor D to a given attribute A.

Induction hypothesis: All attribute occurrences on a path of length k are reevaluated if all attributes on the path change value.

n = 1 $\quad A = f(\ldots,D,\ldots)$

A is placed on the queue in the if statement on the first iteration of the while loop, since A directly depends on D. Since the while loop will not terminate

until A is removed (and evaluated), A will be reevaluated.

Assume the induction hypothesis for n = k

n = k+1

By induction, x_n has been reevaluated, If x_n changed value, A is added to the queue in statement 10, since A depends directly on x_n. Since the while statement executes so long as the queue is non-empty, A is guaranteed to be reevaluated. If x_n has not changed value, A does not need to be reevaluated.

Part 2

If any of A's donors had changed value, A would have been queued for reevaluation. Since A was not on the queue, reevaluation (should it occur) would take place with the same values as were used in the previous evaluation. Since we deal only with pure functions in attribute definitions, A's value could not change, and reevaluation would be useless.

Part 3

Follows from parts 1 and 2, since if A's value were inconsistant with the values of its donors, A would be on the reevaluation queue. ⊠

We will need one definition in the following proof.

Definition 3.2

The state of an evaluation is the partition of attributes into sets of evaluated and unevaluated attributes.

Theorem 3.3

If a static evaluation strategy correctly evaluates attributed grammars without time-varying attributes, the algorithm of Figure 3.3 will correctly evaluate time-varying attributed grammars.

Proof

Intuitively, the static evaluator does not notice any effects of reevaluation, except that some values may have changed. After an attribute has been evaluated, the evaluation is in some state, Q.

Case 1 : the attribute did not change value

The static evaluator is called again, and will make another evaluation move since it correctly performs static evaluation.

Case 2 : the attribute changed value

By (3.1) only those attributes that are already defined can be reevaluated. No unevaluated attributes are given values, and no defined attributes

become undefined by the reevaluation process, so the evaluation remains in state Q. The while loop makes an evaluation step each time an attribute is reevaluated, and the lemma guarantees that any attributes that depend (directly, or indirectly) on the changed value will be reevaluated. If the repeat loop is exited, all of the reevaluable attributes will hold consistent values. The static evaluator will make an evaluation step since we are still in state Q and it correctly performs static evaluation. ⊠

The preceding proof shows that the reevaluation mechanism is logically independent of the static evaluator to which it is grafted, under the assuption that the static evaluator is driven solely by evaluation state and not by an attributes value. (This assumption seems to always be valid.) In practice they should be combined, since both Algorithm 3.1 and the static evaluator must maintain information about donor and recipient availability. Duplication of such information is costly in both time and space. If the treewalk evaluator were to use the donor and recipient vectors and the dependency matrix to schedule visits, duplication would be eliminated. Such an evaluator will be discussed later.

EXAMPLE 1

As an example of this algorithm, consider production 2 and its attribute functions:

2. list ::= list' inst
    2.1 list.end = inst.end
    2.2 list.update = CAT(list'.update,inst.update)
    2.3 list.start = list.start
    2.4 list.table = list.table
    2.5 inst.begin = list'.end + 1
    2.6 inst.table = list.table

The associated matrix and vectors are:

	inherited list		synthesized list'		synthesized inst	
	table	start	end	update	end	update
list end	0	0	0	0	1	0
(syn) update	0	0	0	1	0	1
list' start	0	1	0	0	0	0
(inh) table	1	0	0	0	0	0
inst begin	0	0	1	0	0	0
(inh) table	1	0	0	0	0	0

MD_2

If we assume that all donors except list'.update are available, that is

$$daav_2 = [111011]$$

we obtain $raav_2 = [101111]$ (by 2.1)

so all attributes except list.update may be evaluated.

A change to list'.end gives

$$cav = [001000]$$

$$\Longrightarrow rv = [000010] \quad \text{(by 3.1)}$$

so inst.begin must be reevaluated. Other.end and BR.offset may be queued as a result of the reevaluation of inst.begin since inst.begin is a donor of both of them.

However, a change to inst.update gives

$$cav = [000100]$$

$$\Longrightarrow rv = [000000] \quad \text{(by 3.1)}$$

and no reevaluations take place (since not all donors of update are available).

3.2 Tables

In a compiler, perhaps the most frequently used item is the symbol table. Each entry corresponds to a name and can contain fields for addresses, type, dimensions, bounds, etc. If the table were treated as an atomic attribute, considerable overhead would result from its use. Whenever any part of the table were changed, (via definition of a new entry, or a change in value of an existing entry) all references to the table would be awakened and reevaluated. Since a modif-

ication is only to a single entry, all references to other entries would be needlessly reevaluated. To avoid this overhead we will take special care when reevaluating attributes that are tables.

Definition 3.4

A table is a collection of tuples of attributes, each having a unique key. The number of fields in each tuple can vary among tables, and the number of tuples within a table can vary with time.

Tables exist on two levels: as a single attribute and as a collection of attributes. When scheduling the initial evaluation of definitions referencing a table, a table is treated the same as any other attribute, although it will usually have an initial value. Values of fields of an entry in the table are extracted by a suitable function that takes a table and a key as arguments. Since use-before-definition of individual entries can occur, we will have to address the case when a key is not matched by any entry in the table.

A solution is to include in the range of attribute values a value representing "undefined". Functions that operate on table entries must return "undefined" when any of the operands are "undefined". A reference to an non-existant entry will yield the value "undefined". An attri-

bute can change value from a value in its normal range to "undefined" if it was given an initial value, and reevaluation caused the value "undefined" to propagate to it. However, once a table entry loses the value "undefined" it can never reattain it. Thus there can be no unbounded oscillation of attribute values between "undefined" and a value in the normal range of an attribute, and "undefined" does not introduce non-termination in algorithms that are known to terminate in the absence of "undefined".

The distinction between an undefined attribute and an attribute with the value "undefined" is important if we wish to be able to plan attribute scheduling in advance, as with plans. When creating plans we assume that if all of the attributes are defined, we can evaluate the result. Tables lack such a property; a table may be defined for some entries and undefined for others.

Efficient reevaluation of references to a table will be accomplished by associating a list of users with each tuple in a table as references are made to the entry. Such lists must be dynamically assigned and maintained, since it is not possible to determine which attribute occurrences will refer to an table entry until the input has been parsed. Since an entry to a table may be referenced before it is defined, the functions that access the table must establish a link

between a non-existant entry and its use. When a non-existant entry is referenced, an entry is created in the table with the value "undefined" for all fields but the key. The reference list for an undefined entry is the same as an entry for defined entries. When the definition of an entry takes place, all entries that are on an undefined entry's reference list are queued for reevaluation. Thus, the reevaluation mechanism takes care of use-before-definition, and reevaluates only the definitions that are interested in a table entry that has changed value.

Supplying the value "undefined" allows us to use the existing scheduling techniques to handle all table references, and let the reevaluation mechanism fix up any forward references.

For example, if the branch length optimization of Figures 3.1 and 3.2 were applied to Figure 2.2

```
B: --------}
   --------} MAX-1 locations
   --------}
a: ref A
b: ref B
   --------}
   --------} MAX-1 locations
   --------}
A: --------
```

the following events would occur. References to the table (via ADR(A) and ADR(B)) would cause table entries of (A,undefined) and (B,undefined), and links to the nodes that

reference A and B would be set up. Any attribute that directly or indirectly depends on ADR(A) or ADR(B) gets the value "undefined". STL.updates then would pass up (B,loc), (a,loc+MAX-1), (b,loc+MAX) and (A,loc+MAX-1). Reevaluation of ADR(A) and ADR(B) would then yield loc+MAX-1 and loc, respectively, so no branch lengths increase to 2. The reevaluation would then halt.

If a user wishes to guard against duplicate donors to an entry in a table, each table entry needs a field to record the source of the information. Thus an insertion program can check that the source field is either "undefined" or the donors identification. If the check fails we know that this is a duplicate entry and corrective action can be taken.

It may be possible that it is safe to overwrite information in a table because of the nature of the problem (for example, a table entry might record the highest value of several occurrences of another attribute). The functions manipulating this sort of table will not bother to check that the source identifier matches this donors, since the order of evaluation is irrelevant.

3.3 Halting

One expects that any specification of attributes and functions will describe a recursive algorithm that always converges to a unique value (regardless of the order of attribute evaluation). Halting can be ensured by dealing with problems whose solutions have a well-founded partial ordering.

Definition 3.5

A partially ordered set (poset) is a pair $(S,\leq)$ where $\leq$ is a relation on S satisfying:

$a \leq a \ \forall\ a \in S$ (reflexitivity)

$a \leq b$ and $b \leq a \Longrightarrow a = b \ \forall\ a,b \in S$ (antisymmetry)

$a \leq b$ and $b \leq c \Longrightarrow a \leq c \ \forall\ a,b,c \in S$ (transitivity)

Also define the relations $<$, $>$, and $\geq$ as follows:

$a < b$ iff $a \leq b$ and $a \neq b$

$a > b$ iff $b < a$

$a \geq b$ iff $b \leq a$

A partial order is well-founded if there are no infinite descending chains

$x_0 > x_1 > x_2 \ldots$

A function $f:S \dashrightarrow S$ is non-increasing if $f(x) \leq x$ for all $x \in S$.

If we are careful to define well-founded attribute domains and non-increasing attribute functions, we well be able to guarantee that the algorithm will halt.

Theorem 3.6

Given $(S,\leq)$, a well-founded poset, $f:S \dashrightarrow S$ a non-increasing total function and x_0, an arbitrary element of S, define $x_{n+1} = f(x_n)$ for $n = 0,1,\ldots$. Then there exists an n such that $x_n = f(x_n)$.

Proof

If there were no n such that $x_{n+1} = f(x_n)$ we would have an infinite descending chain (since $f(x) < x$ if equality does not hold). Therefore there exists an n such that $x_n = f(x_n)$ ∎

Many of our examples use attribute domains falling into one of two general categories: splitting a finite set and counting down to zero. Both of these categories represent will-founded partial orderings. Examples of techniques that fit into these general schemes include:

1) Splitting a finite set into smaller, non-empty sets (S = a finite set, $\leq$ = $\subseteq$, f = intersection with any subset of S).
2) Merging sets over a finite universe (or any finite number of sets) into larger sets (S = a finite set, $\leq$ =

$\supseteq$, f = union with any subset of S).

3) Boolean conditions that are "pinned" at TRUE or FALSE as in [Babich 1975] (S = {true,false},false $\leq$ true, f = logical and), (S = {true,false}, true $\leq$ false, f = logical or).
4) Use of attributes which can only change value a bounded number of times (as in the branch length example).
5) Counting down to zero (assuming that we begin with a positive value) (S = non-negative integers, $\leq$ = "less than or equal", f = subtraction by any element of S).

Chapter 4: EVALUATORS FOR TIME-VARYING ATTRIBUTES

This chapter describes two treewalk evaluators for time-varying attributes. The first is very simple-minded, but shows that availability vectors and dependency matrices can be used to schedule both initial and iterative evaluation of attributes. The second is an extension of the method of Kennedy and Warren. It adds necessary provisions for initial values and breaking circularity, and incorporates the reevaluation mechanism into their evaluation technique. Each method is followed by a detailed example of its use.

4.1 A Simple Treewalk Evaluator

The treewalk evaluator we describe first is similar to that described by Fang [Fang 72], but is more sophisticated than his evaluator. The algorithm for evaluating attributes is recursive; the evaluator visits nodes, evaluating whatever is ready to be evaluated, then interleaves visits to offspring with further attempts to evaluate attributes at the current node. When nothing is left to evaluate, it

returns to the parent node, or halts (if it is visiting the root node). This algorithm differs from that of Fang in two ways. The evaluator will discover attribute dependencies, rather than requiring the user to give a partial ordering of evaluations within a node, and it will reevaluate attributes when necessary.

To discover the attributes that can be evaluated, we use the dependency matrix and donor availability vector. The daav is constructed by keeping a list of all occurrences in which a given attribute appears as a donor. When an attribute is evaluated, its position in each appropriate daav is set to true. Attributes are assigned a number within each symbol. Thus, to reference an attribute, a pair (node,position) is used. Attributes are also ordered in another sense. Each donor or recipient is assigned a position in the daav and raav of a node. This position is used to index an array of (node,position) pairs when attribute values are stored or retrieved, so we know where to find the ith donor or recipient. For each production i, the dependency matrices, MD_i, and the number of recipient and donor attributes, $NumR_i$ and $NumD_i$, are tabulated. (Node.LHS is used to index them.) MaxR is set to the maximum number of recipients over all productions, and MaxD is set to the maximum number of donors.

4.2 Node Structure

Each node represents a grammar symbol, and has a structure to represent the state of the evaluation. Non-terminal symbols have all of the information described below; terminal symbols without inherited attributes have only node.attr and node.visit. The latter is used to avoid visits to terminal nodes.

Node.attr is an array of structures, one structure for each attribute of this symbol. The attribute are indexed through the arrays node.donor and node.recip.

Node.attr.value contains the value of the attribute.

Node.attr.donorOf contains a list of (node, daav position) that correspond to the donor uses of this attribute.

Node.visit is initially true for non-terminals, set false once a visit has been made to this node. It is always false for terminals.

Node.cav is the changed (donor) attribute vector.

Node.daav is the donor attribute availability vector.

Node.raav is the recipient attribute availabilty vector.

Node.LHS contains the production for which this symbol is the left-hand side.

Node.donor is an array of pointers to the donor attributes. It is indexed by the corresponding position in

node.daav.

Node.recip is an array of pointers to the recipient attributes. It is indexed by the corresponding position in node.raav.

Node.children is an array of pointers to offspring nodes. It is indexed by the position of a symbol in the production's right-hand side.

Initialization is necessary before the treewalk can begin. Synthesized attributes of terminals, and time-varying attributes with initial values must have appropriate initialization in node.attr. The daav bits corresponding to attributes with initial values must be initialized to 1. However, the raav bits corresponding to such attributes will be initialized to zero since we want to evaluate this attribute again, using the iterative definition.

As an attribute is evaluated, all nodes that use it as a donor have corresponding bits set in their daav. When an offspring is visited, (2.1) can be used to determine the next attributes to be evaluated. Reevaluation begins when an attribute changes value. All attributes that depend on the changed attribute are placed on a queue, and procedure reeval is called. Reeval returns only when the reevaluation queue is exhausted.


```
procedure visit (this:node);
var
    child : node;
    ToBeVisited : set of node;
procedure eval; (* see Figure 4.2 *)
procedure reeval; (* see Figure 4.3 *)
begin (* procedure visit *)
    if this.visit then (* build ToBeVisited *)
        ToBeVisited := [this.children[i] |
                            this.children[i].visit]
        this.visit := false
    else
        ToBeVisited := empty
    fi;
    eval;
    while ToBeVisited ≠ empty do
        remove a child from ToBeVisited;
        visit (child);
        eval(this,ToBeVisited)
        (* eval may add nodes to ToBeVisited *)
    endwhile
end; (* visit *)
```

Figure 4.1: Procedure visit

```
procedure eval (this:node; ToBeVisited:set of node)
var
  other : node;
  pos,i,j : integer;
  newraav : array [1..$NumR_{this.LHS}$] of boolean;
begin
  i := this.LHS;
  $\overline{newraav}$ := $MD_i$ x this.$\overline{daav}$;
  while newraav $\neq$ this.raav do
    for each position j where $newraav_j \wedge$ this.$\overline{raav}_j$ do
        evaluate this.$recip_j$;
        for each (other,pos) on this.$attr_j$.donorOf do
            other.$daav_{pos}$ := true
        od;
        if other $\neq$ this then
            ToBeVisited := ToBeVisited + other
        fi;
        (* check for reevaluation *)
        if this.$recip_j$ has changed value then
            for each (other,pos) on this.$attr_j$.donorOf do
                ENQUEUE (other);
                other.$cav_{pos}$ := true;
            od
            reeval;
        fi
    od (* each position j *)
    this.raav := newraav;
    $\overline{newraav}$ := $MD_i$ x this.$\overline{daav}$;
  endwhile;
end; (* eval *)
```

Figure 4.2: Procedure eval

```
procedure reeval;
var
    revisit,other : node;
    pos,i,j : integer;
    rv : array [1..MaxR] of boolean;
begin
  while rvqueue $\neq$ empty do
    revisit := DEQUEUE;
    i := revisit.LHS;
    rv := ($MD_i$ x revisit.cav) $\wedge$ revisit.raav;
    revisit.cav := false;
    for each position j where $rv_j$ do
      reevaluate revisit.$recip_j$;
      if it has changed value then
        for each (other,pos) on this.$attr_j$.donorOf do
          ENQUEUE (other);
          other.$cav_{pos}$ := true
        od
      fi
    od;
  endwhile;
end; (* reeval *)
```

Figure 4.3: Procedure reeval

Theorem 4.1

Procedure Eval is correct in the sense that

(A) An attribute is evaluated iff all of its donors are available.

(B) An attribute is reevaluated iff a donor changes value.

Proof

(A) Initially, all synthetic attributes of terminal symbols, and all attributes with initial values have their bits in the appropriate daavs set to true. When an attribute is evaluated, procedure eval sets bits in all daavs where the attribute is used as a donor. Therefore equation (2.1) correctly determines the attributes that are evaluable. An attribute is evaluated by eval when $newraav_j \wedge \overline{raav}_j$ is 1, so it is evaluated only once, at the time that its donors are available. The first reevaluation of an attribute is done be eval, since raav is initially Ø for all attributes with an initial value.

It remains to show that each node is visited when an attribute at that node is ready to be evaluated.

1) If the attribute is not dependent (directly or indirectly) on an inherited attribute, the attribute must be evaluated before the first visit ends (after

all of its offspring have been visited). This visit is guaranteed by the first statement of the main program, when ToBeVisited is built.

2) If the attribute depends on an inherited attribute, its node will be placed on ToBeVisited by procedure eval when the inherited attribute is evaluated.

The main loop of the program guarantees that every node in ToBeVisited will be visited.

(B) The cav has bits set to one in only two places: where eval or reeval detects that an attribute has changed value. At the same time, the corresponding node is enqueued. When the node is removed from the queue, equation (3.2) can be applied, and rv will indicate exactly those attributes that must be reevaluated. Since cav is cleared after the reevaluation vector is calculated, no attributes will be redundantly reevaluated (although they may be reevaluated if their donors change value again). Thus no extraneous reevaluations will occur. ⊠

Theorem 4.2

If no reevaluation is done, the number of visits scheduled by Algorithm Eval is linear in the number of nodes in the parse tree.

Proof

The maximum number of subtree visits at any node is equal to the number of offspring of the node plus the number of inherited attributes of the right hand side, since visits can only be rescheduled if an inherited attribute is evaluated. We can bound the number of offspring by a constant N_O. We can bound the number of inherited attributes in any production by N_I. Therefore we have at most $N_O + N_I$ visits to subtrees at any node. ⊠

Corollary 4.3

The number of visits to nodes is linear in the size of the input, if we are walking the parse tree of an unambiguous context-free grammar.

Proof

For unambiguous context-free grammars, the size of the parse tree is linear in the size of the input. ⊠

4.3 An Example

As an example of the operation of the simple treewalk evaluator, we will use a simplification of the short-long branch optimization example (Figures 3.1 and 3.2). In this example, the offset is computed without a symbol table, and the "list" is exactly two instructions long. The offset of A will be computed by adding A's and B's length, which implies that we know tnat we want to skip everything else in A and B. We also violate Chirica & Martin's [CM 76] restrictions on the use of A.len and A.offset as both donors and recipients in one production. An initial-valued attribute, len, is used, and reevaluation will be necessary.

```
1. S ::= A B
    1.1 S.span = B.start + B.len
    1.2 A.offset = A.len + B.len
    1.3 if abs(A.offset) > 10 then A.len = 2
    1.4 B.start = A.start + A.len

2. A ::= a
    2.1 A.start = a.value

3. B ::= b
    3.1 B.len = b.len

I(S) = Ø                        I(a) = Ø
S(S) = {span}                   S(a) = {value}

I(A) = {len=1,offset}           I(b) = Ø
S(A) = {start}                  S(b) = {len}

I(B) = {start}
S(B) = {len}
```

Figure 4.4: Offset Calculation

The language of this grammar has only one element, ab, and only one parse tree. The scanner is assumed to have provided the values 24 and 15 for a.value and b.len, respectively. The data structure for each node is given below, and each reflects the initialization of values and daavs. The dependency matrices and constants are given at the end. We will use the name of a symbol to refer to the node of the parse tree corresponding to that symbol. No ambiguity arises in this example since each grammar symbol appears just once in the parse tree.

A note about the queueing mechanism is necessary. Reeval queues nodes that contain attributes to be reevaluated, rather than attribute occurrences, since the cav and dependency matrix are used to determine the attributes to be reevaluated. A node therefore is queued only once, even if more than one of its attributes needs to be reevaluated.

Node (S)

```
S.attr[1] (* span *)
    value = undefined
    donorOf = Ø (* span is not a donor of any attribute *)
S.LHS = 1
S.visit = true
S.cav = [00000]
S.daav = [10000]   (* length depends on the production *)
S.raav = [0000]
S.donor[1] = (A,1) (* A.len *)
       [2] = (A,2) (* A.offset *)
       [3] = (A,3) (* A.start *)
       [4] = (B,1) (* B.len *)
       [5] = (B,2) (* B.start *)
S.recip[1] = (S,1) (* S.span *)
       [2] = (A,1) (* A.len *)
       [3] = (A,2) (* A.offset *)
       [4] = (B,2) (* B.start *)
S.children[1] = A
          [2] = B
```

Node (A)

```
A.attr[1] (* len *)
    value = undefined
    donorOf = (S,1) (* A.len has position 1 in S.daav *)
A.attr[2] (* offset *)
    value = undefined
    donorOf = (S,2)
A.attr[3] (* start *)
    value = undefined
    donorOf = (S,3)
A.LHS = 2
A.visit = true
A.cav = [0]
A.daav = [1]
A.raav = [0]
A.donor[1] = (a,1) (* a.value *)
A.recip[1] = (A,3) (* A.start *)
A.children[1] = a
```

Node (B)

```
B.attr[1] (* len *)
    value = undefined
    donorOf = (S,4) (* B.len has position 4 in S.daav *)
S.attr[2] (* start *)
    value = undefined
    donorOf = (S,5)
B.LHS = 3
B.visit = true
B.cav = [0]
B.daav = [1]
B.raav = [0]
B.donor[1] = (b,1) (* b.len *)
B.recip[1] = (B,1) (* B.len *)
B.children[1] = b
```

Node (a)

```
a.attr[1] (* value *)
    value = 24 supplied by scanner
    donorOf = (A,1) (* a.value has position 1 in A.daav *)
a.visit = false
```

Node (b)

```
b.attr[1] (* len *)
    value = 15 supplied by scanner
    donorOf = (B,1) (* b.len has position 1 in B.daav *)
b.visit = false
```

PRECALCULATED ITEMS

$NumR_1 = 4$ $NumR_2 = 1$ $NumR_3 = 1$ $MaxR = 4$
$NumD_1 = 5$ $NumD_2 = 1$ $NumD_3 = 1$ $MaxD = 5$

$$MD_1 = \begin{vmatrix} 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{vmatrix} \qquad MD_2 = [1] \quad MD_3 = [1]$$

Figure 4.5: Node Structure for Offset Calculation

A TREEWALK

The treewalk begins by visiting the root.

ToBeVisited := [A,B]

(2.1) is applied to S.daav; newraav is [0000]

no evaluation takes place

Visit A

ToBeVisited := [] (local to A)

(2.1) is applied to A.daav; newraav is [1]

A.recip[1] = (A,3) is evaluated (A.start = 24)

A.attr.donors[3] = (S,3) so $S.daav_3 = 1$

return to S

call eval;

(2.1) is applied to S.daav; newraav is [0001]

S.recip[4] = (B,2) is evaluated (B.start = 25)

B.attr.donors[2] = (S,5) so $S.daav_5 = 1$

(2.1) is applied to S.daav; newraav is [0001]

eval exits

Visit B

ToBeVisited = [] (local to B)

(2.1) is applied to B.raav; newraav is [1]

B.recip[1] = (B,1) is evaluated (B.len = 15)

B.attr.donors[1] = (S,4) so $S.daav_4 = 1$

return to S

eval computes newraav = [1011]

S.recip[1]=(S,1) and S.recip[3]=(A,2) are evaluated

S.span = 40

A.offset = 16

The latter sets $S.daav_2$ = 1, so S.daav = [11111]

(2.1) is applied, and S.recip[2] = (A,1) is evaluated

A.len = 2 and HAS CHANGED VALUE

Since A.attr.donor[1] = (S,1)

$S.cav_1$ = 1 and S is ENQUEUED

reeval is called

S is removed from the queue.

(4.1) is applied; rv = [0011]

s.cav = false

S.recip[3] = (A,2) is reevaluated (A.offset = 17)

THIS IS A CHANGE IN VALUE

S is placed back on the queue

$s.cav_2$ = 1

S.recip[4] = (B,1) is reevaluated (B.start = 26)

THIS IS A CHANGE IN VALUE

S is enqueued; the queue is unchanged.

$s.cav_5$ = 1

S is removed from the queue.

(4.1) is applied; rv = [1100]

s.cav = false

S.recip[1] = (S,1) is reevaluated (S.span = 41)

THIS IS A CHANGE IN VALUE

S.span is not used as a donor

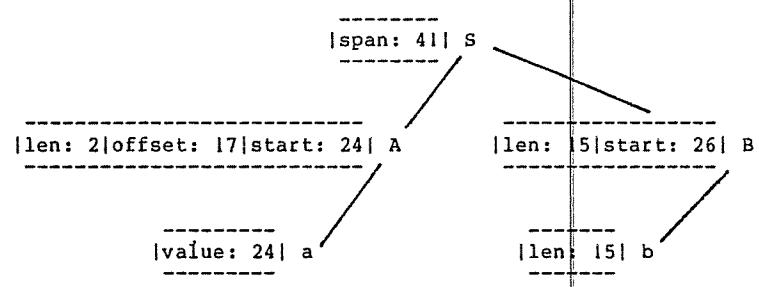
S.recip[2] = (A,1) is reevaluated (A.len = 2)

NO CHANGE IN VALUE

reeval exits

Nothing remains in ToBeVisited

The treewalk ends, with the following values for attributes:



4.4 Plans

In order to avoid run-time scheduling of visits to nodes for evaluation of attributes, the idea of a plan was developed by Kennedy and Warren [KW 76]. An evaluation is done by walking a complete parse tree, but the order of evaluation of attributes and visits to nodes is calculated at evaluator generation time, rather than at treewalk time. This method requires limitations on the type of definitions

that are allowed. Basically, the definition must be non-circular (so that an evaluation order can be found), but it also must have the property that different subtrees rooted by given non-terminal symbol yield their attributes in more or less the same order (so that evaluation be scheduled in advance). The former restriction will obviously have to be relaxed if time-varying attributes are used in iterative algorithms. The latter restriction will be retained to minimize the effect of adding time-varying attributes.

A plan for visiting nodes of a parse tree contains two types of instructions: attribute evaluations and (recursive) visits to offspring nodes. The control structure for such an evaluator is simple. As a node is entered, a given set of straight-line code sequences is executed. When finished, the evaluator returns to its caller. No decisions need to be made about attribute availability or dependency. Since nodes may be visited more than once, a tag conveying state information is left when a visit has ended. State information is also passed from a parent to an offspring node at the beginning of a visit. With these two pieces of information, it will be possible to choose a plan that was previously created.

In order to create plans, the dependency of attributes must be discovered. Kennedy and Warren require a condition

that they call "absolute non-circularity". It is detected by creating a directed graph that shows the dependence of attributes within a production. Graphs headed by the same non-terminal are merged together, and if the resulting graph is acyclic, the plans can be created.

Circularity of attribute definitions will cause the merged dependency graphs to have cycles, and the planning mechanism will fail. However, there is a way to make plans involving time-varying attributes by modifying the plan creation algorithm in a simple way. Whenever an attribute depends on a donor that has an initial value, the dependency is ignored (as regards the scheduling mechanism). If this is done, it should break circular dependency among attributes. (If the modified grammar is circular, the original grammar must also have been circular.)

The planning algorithm is concerned about whether or not an attribute has been defined. If we assume that once defined, an attribute never becomes undefined, we know that an initial-valued attribute's availability can be taken for granted. Ignoring its use as a donor is equivalent to assuming that it is always available. In the simple tree-walk example, A.len depended on A.offset, and vice-versa. Since A.len has an initial value, we can break the circularity by ignoring A.offset's dependence on A.len when schedul-

ing initial evaluation. It may be that after ignoring dependence, an attribute appears not to depend on anything. This causes no problem, since it is analogous to the case of a definition like "A.offset = 1", where A.offset is immediately evaluated upon the first visit to node A.

Attributes with initial values are treated in exactly the same manner as all other attributes, with respect to the use of their iterative definition. One application of the iterative definition of a time-varying attribute will be scheduled by the planning algorithm when all of its donors are available. Any other applications of the iterative definition will be done by the reevaluation mechanism, as in the previous evaluator.

The plans that are created will schedule visits to initially evaluate all of the attributes, and will provide the first reevaluation of attributes with initial values. What remains is to graft the reevaluation mechanism onto the plans.

Recall that reevaluation is done only when a donor has changed value, and the recipient attribute has already been evaluated. Thus the starting point for reevaluation will be when an attribute with an initial value is evaluated using its "iterative" definition. At that point a new kind of instruction is executed to check for a change in value. If

a change has occurred, all attributes that depend on the changed attribute are enqueued and the reevaluation procedure is called. It is significant that the only time that an attribute needs to be checked (by an evaluation plan) for a change in value is when the attribute just evaluated has an initial value. No other attributes can possibly have changed value, except through the reevaluation procedure, which does its own checking for changed values.

Within the reevaluation procedure, every evaluation of an attribute is checked for a change in value. This causes changes to propagate throughout the parse tree. It also limits the need for a check for a change in value to those evaluations that are on a reevaluation cycle.

The reevaluation is much like that done in the previous evaluator. At evaluator generation time, we can enumerate those attributes that directly depend on a given attribute. If we store a flag with each attribute that tells whether or not an attribute has been evaluated, we will know whether or not to reevaluate a given attribute. Thus, the "plan" for reevaluation is to queue all evaluated attributes that depend on a given attribute, and invoke the reevaluation procedure. Lists of attributes that directly depend on a given attribute can be tabulated at evaluator-generation time, so there is very little run-time overhead.

```
procedure reevalplan;
var
    atr : attribute;
begin
    while rvqueue ≠ empty do
        atr := DEQUEUE;
        evaluate atr;
        if atr has changed value then
            ENQUEUE all elements of USES(atr)
            that have been evaluated;
        fi
    endwhile
end; (* reevalplan *)
```

Figure 4.6: Algorithm Reevalplan

We will call the evaluator resulting from adding reevaluation to [KW 76] the modified plan evaluator.

Theorem 4.4

When time-varying attributes are not present in an attributed grammar, the modified plan evaluator is identical to the plan evaluator of [KW 76].

Proof

All changes to the plan creation routines of [KW 76] occur when an attribute with an initial value is processed. Non-time-varying attributes do not have initial values, so the plans will be identical. ⊠

Theorem 4.5

The modified plan evaluator will correctly reevaluate attributes in the sense that:

(A) An attribute is evaluated iff all of its donors are available.

(B) An attribute is reevaluated iff a donor changes value.

Proof

(A) The planning algorithm has had two changes: dependence upon attribute occurrences with initial values is ignored, and a check is made for a change of value if an attribute has an initial value. The former change is benign to the planning algorithm, as discussed earlier. The latter does not effect the availability of attributes, so we depend on the correctness of [KW 76] to guarantee that attributes are evaluated when its donors are available.

(B) Reevaluation is initiated when an attribute with an initial value is evaluated in accordance with its definition, and the modified plan evaluator potentially initiated reevaluation at exactly those times. The reevaluation algorithm parallels that of the previous evaluator, as does its proof, which will not be elaborated again here. ⊠

4.5 Another Example

As an example of using plans, we will again use the specification of the short-long branch optimization example in Figures 4.1 and 4.2. The data structure for each node is much simpler when plans are used, since many decisions are made in advance. Only the attribute values, the state of the node, the production number and the array of offspring are needed.

```
1. S ::= A B
    1.1 S.span = B.start + B.len
    1.2 A.offset = A.len + B.len
    1.3 if abs(A.offset) > 10 then A.len = 2
    1.4 B.start = A.start + A.len

2. A ::= a
    2.1 A.start = a.value

3. B ::= b
    3.1 B.len = b.len
```

```
I(S) = Ø                    I(a) = Ø
S(S) = {span}               S(a) = {value}

I(A) = {len=1,offset}       I(b) = Ø
S(A) = {start}              S(b) = {len}

I(B) = {start}
S(B) = {len}
```

Figure 4.7: Offset Calculation (repeated)

Node (S)

```
S.value[1] = undefined (* span *)
S.LHS = 1
S.children[1] = A
          [2] = B
S.state = q1
```

Node (A)

```
A.value[1] = 1 (* len *)
       [2] = undefined (* offset *)
       [3] = undefined (* start *)
A.LHS = 2
A.children[1] = a
A.state = q2
```

Node (B)

```
B.value[1] = undefined (* len *)
       [2] = undefined (* start *)
B.LHS = 3
B.children[1] = b
B.state = q3
```

Node (a)

```
a.value[1] = 24 (* value *) supplied by scanner
```

Node (b)

```
b.value[1] = 15 (* len *) supplied by scanner
```

PRECALCULATED SETS

```
USES(S.span)     = Ø
USES(A.len)      = {A.offset,B.start}
USES(A.offset)   = {A.len}
USES(A.start)    = {B.start}
USES(B.len)      = {S.span,A.offset}
USES(B.start)    = {S.span}
USES(a.value)    = {A.start}
USES(b.len)      = {B.len}
```

Figure 4.8: Node Structure II

Dependency graphs are used to build the plans. They perform the same function as the dependency matrices of Rowland, linking donor and recipient attributes by directed arcs. The dependency graphs, DG_i, for this example are:

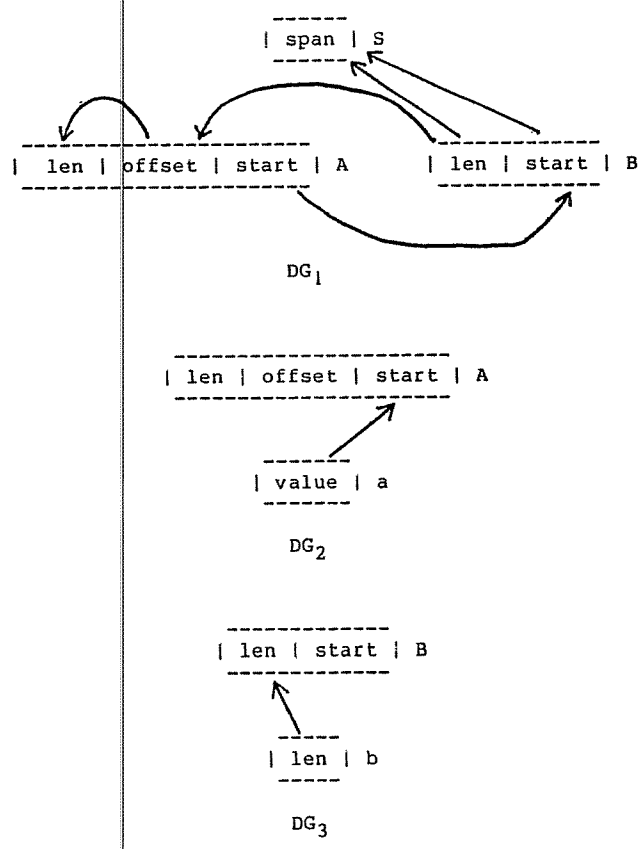


Figure 4.9: Dependency Graphs for Offset Calculation

These graphs reflect the modification for breaking circularity; A.len has no arcs leaving its node. In this example, the closure of the dependency matrices graphs is the same as the original graph, since each non-terminal is on the left-hand side of only one production. The plan construction is straightforward. Productions 2 and 3 have only one possible plan, regardless of the set of attributes available upon their call.

Plan 2

Evaluate A.start

A.state = q_{end}

return

Plan 3

Evaluate B.len

B.state = q_{end}

return

There is also only one possible plan for node S, which occurs when the treewalk is begun. Immediately upon examining the root an attempt is made to evaluate some attributes. This fails, so the planning algorithm tries a visit to the offspring, followed by a new attempt at evaluation at the root. To do so, the "yield" is calculated for each closure of subtrees headed by a given non-terminal (The yield is the

set of attributes it evaluates). The yields depend on the attributes that are available when the visit is made; in this case the "input set" of attributes is empty. After a visit has been made to a subtree, a guaranteed available set of attributes is known.

After node A is visited, A.start is guaranteed available, so B.start can be evaluated. Similarly, after node B is visited, B.len is guaranteed available, so both S.span and A.offset are evaluable. This leaves A.len ready to be evaluated, and the plan to visit node S is complete. However, A.len has an initial value, so the plan must include an instruction to check for a change in A.len's value and a call to the reevaluation routine. The resulting plan is:

```
PLAN 1
    visit offspring 1 with attribute set Ø
    evaluate B.start
    visit offspring 2 with attribute set {B.start}
    evaluate S.span
    evaluate A.offset
    evaluate A.len
    if A.len has changed value then
        enqueue all attributes that depend on A.len
        call reeval
        fi
```



```
    S.state = q_end
    return
```

A "GO TO" table is used to determine the appropriate plan. It is indexed by the current state, and a set of inherited attributes. The table is very simple in this example; it does not depend on the inherited attribute set at all. It is included, however, for completeness.

	attribute set: Ø	{B.start}
state q_1	q_1	--
state q_2	q_2	---
state q_3	--	q_3

Figure 4.10: A GO TO table

A TREEWALK

The treewalk begins with VISIT(S,Ø).

Plan 1 is selected, since

S.state = q_1,

attribute set is Ø,

and GOTO(q_1,Ø) is 1.

VISIT(A,Ø) selects plan 2.

A.start is evaluated (= 24).

A.state = q_{end},

return from plan 2.

B.start is evaluated (= 25).

VISIT(B,{B.len}) selects plan 3.

B.len is evaluated (= 15).

B.state = q_{end},

return from plan 3.

S.span is evaluated (= 40).

A.offset is evaluated (= 16).

A.len is evaluated (= 2)

A.len is checked for a change in value;

THIS IS A CHANGE IN VALUE.

USES(A.len) are placed on rvqueue

(A.offset and B.start).

reevalplan is called.

A.offset is removed from rvqueue

and is reevaluated (= 17)

THIS IS A CHANGE IN VALUE.

A.len is enqueued (rvqueue=B.start,A.len).

B.start is removed from rvqueue

and is reevaluated (= 26).

THIS IS A CHANGE IN VALUE.

S.span is enqueued (rvqueue=A.len,S.span).

A.len is removed from rvqueue

and is reevaluated (= 2).

NO CHANGE IN VALUE

(rvqueue=S.span).

S.span is removed from rvqueue (rvqueue=empty).

and is reevaluated (= 41).

THIS IS A CHANGE IN VALUE.

USES(S.span) = Ø, so nothing is enqueued.

reevalplan ends.

S.state = q_{end}.

Plan 1 returns with the completely evaluated tree.

The resulting decorated tree is omitted, since it is exactly the same as the result of the previous example. However, the amount of run-time overhead for scheduling evaluation is far less than that of the previous evaluator, since we have tabled the scheduling order in advance of the evaluations. This demonstrates the probable superiority of plans over the previous evaluator in the evaluation of time-varying attributed grammars.

Chapter 5: NON-LOCAL ATTRIBUTES

Wilner [Wil 71] noted that in practice a significant number of attribute definitions are identities; that is certainly the case in the branch length optimization example (Figures 3.1 and 3.2). Therefore, he allowed such definitions to be omitted if the attributes had the same name and were associated with symbols on the left-hand and right-hand sides of the same production. In a more general fashion, it is possible to allow references to non-local attributes (attributes of symbols that do not explicitly appear in the production containing the definition) if one restricts reference to symbols that can be "easily and uniquely resolved". (We will call the production containing the definition the referencing production.) One interpretation of "easily and uniquely resolved" is to search up the parse tree, and use the closest occurrence of the non-local symbol. This search need not be done at attribute evaluation time, since we can process the grammar in advance and decide which productions have the symbol associated with the non-local attribute. Since a production can appear in different contexts depending on the program being parsed, we must insure that there is always an occurrence of the non-local symbol on the path from the left-hand side of the referenc-

ing production to the root of the parse tree. The following definition extends Definition 2.9 to include the use of non-local attribute references.

Definition 5.1

An attributed parse tree for an attributed grammar G with non-local attribute references is a parse tree for the underlying context-free grammar of G with values assigned to the attributes of the nodes in a way consistant with the attribute definitions in the following sense: Given a production p, $X_0 ::= X_1 \ldots X_n$ and a non-local reference to A.attr ($A \notin \{X_0, \ldots, X_n\}$), the following conditions must hold:

1) Each node labelled X_0 with offspring $X_1 \ldots X_n$ must have an ancestor node labelled A.
2) If A.attr is a donor is a definition at production p (i.e. $X_j.a_k = f_{pjk}(\ldots, A.attr, \ldots)$ $0 \leq j \leq n$) then the definition must hold when the values of the attributes of $X_0, \ldots X_n$ and the value of attr found in X_0's nearest ancestor node labelled A are substituted in the definition.
3) If A.attr is a recipient in a definition at production p then at any node labelled X_0 with offspring $X_1 \ldots X_n$ the value of attr found in X_0's nearset ancestor node labelled A must agree with the value of the right-hand side of A.attr's definition.

using values of attributes of $X_0, \ldots, X_n$.

5.1 Example of Non-local Attributes

The use of non-local symbols allows the specification of the branch length optimization of Figures 3.1 and 3.2 to be shortened and rendered more readable. In the following specification, UPDATE is a function that enters (label,address) pairs in the symbol table, and ADR returns the address of a label in a given table. The non-local references to attributes in this example are codesect.start in 3.1, list.end in 4.1 and 5.1, and codesect.table in 4.2, 5.2 and 5.3.

```
1. codesect ::= list
    1.1 codesect.start = arbitrary

2. list ::= list' inst
    2.1 inst.begin = list'.end + 1

3. list ::= inst
    3.1 inst.begin = codesect.start

4. inst ::= other
    4.1 list.end = inst.begin + other.len - 1
    4.2 if other.label ≠ '' then codesect.table =
        UPDATE(codesect.table,other.label,inst.begin) fi
```



```
5. inst ::= BR
    5.1 list.end = inst.begin + BR.len - 1
    5.2 if BR.label ≠ '' then codesect.table =
            UPDATE(codesect.table,BR.label,inst.begin) fi
    5.3 BR.offset = ADR(codesect.table,BR.target)-inst.begin
    5.4 if abs(BR.offset) > Maxdisp then BR.len = 2 fi

I(codesect) = Ø                        I(list) = Ø
S(codesect) = {table=empty,start}      S(list) = {end}

I(inst) = {begin}                      I(other) = Ø
S(inst) = Ø                            S(other) = {label,len}

I(BR) = {offset,len=1}
S(BR) = {label,target}
```

Figure 5.1: Branch Length Optimization II

Figure 5.1 uses five fewer attributes and eleven fewer attribute definitions than the specification without non-local attributes (Figures 3.1 and 3.2). It seems that the specification is more readable when non-local attributes are used. It also seems to be easier to write such specifications using non-local attributes.

Codesect.table and inst.begin are synthetic attributes since they are defined in terms of values from their subtrees. Any attribute that is used non-locally as a recipient will be synthesized.

5.2 Removing Non-local Attributes

To show that non-local attributes can be introduced with no new complications for the attribute evaluators we give an algorithm that converts an attributed grammar with non-local attribute references to an equivalent attributed grammar without non-local references.

A simple way of eliminating non-local attribute references is to modify the attributed grammar by replacing a non-local attribute reference with a chain of identity definitions. Attributed parser generators can add identity definitions in much the same way that a parser generator can automatically remove chain productions. Thus non-local attributes are a convenient abbreviation for larger and more detailed attribute specifications in standard attributed grammars.

We will use a depth-first backtracking algorithm to find the identity definitions needed to remove the non-local attribute references. At each stage of the recursion, the algorithm looks at a production. If the symbol associated with the non-local attribute is the LHS of that production, we connect the current production to the referencing production. If the symbol does not exist in the current production, the algorithm builds a set of all possible ancestors (all productions that have the current LHS symbol somewhere in their RHS) and recursively visits them.

The algorithm backtracks when (1) the required symbol has been found or (2) no immediate ancestor of this production (that has not already been visited) exists. A stack (kept in parallel with the recursion) is used to contain the path of productions from the referencing production to the current production. The production that is currently being processed is on the top of the stack, hence no explicit parameters are used. The information in the stack is used to add attribute definitions that link the non-local symbol to the referencing production.

If we reach a point in the algorithm where no unvisited ancestors of the production exist, we have a potential erroneous reference to a non-local symbol (some derivation has a path from the referencing production to the root of the parse tree that does not contain the non-local symbol). When non-local symbols are used, all paths in the depth-first search must yield the non-local symbol so that we can guarantee that the reference is valid in all possible parse trees. The algorithm terminates when no partially-explored paths remain.

Non-local attributes can be used either as donors or recipients, and their use determines whether we add definitions for inherited or synthesized attributes. If a non-local attribute is used as a donor, information is flowing into the subtree and the attribute is inherited. Conversely, the use of a non-local attribute as a recipient implies that the attribute is synthesized.

Figures 5.2 and 5.3 give procedures that add attribute definitions that connect uses of non-local attributes with the actual occurence of the attribute. Walkup is the part of the algorithm that does the search. Its result is a set of pairs of production numbers (i,j), which signify that LHS[i].attribute and LHS[j].attribute need to be connected by an identity definition at production i. The size of a stack s is denoted |s|, and its top element is denoted Top(s). To avoid infinite recursion due to self-embedding non-terminals, the set "Used" is used to prevent visits to productions that have already been visited.

AddIdentity is the main program of the algorithm. Its arguments are Goal, the symbol assocaiated with the non-local attribute attr, k, the production containing the non-local attribute reference, and donor, a Boolean that determines whether inherited or synthesized definition are added. AddIdentity calls Walkup to find the necessary identity

definitions and adds them to the attributed grammar.

There are two restrictions on the use of AddIdentity.

<u>Restriction 5.2</u>

An attribute cannot be referenced non-locally as both a donor and a recipient.

<u>Restriction 5.3</u>

There cannot be any definition of "attr" at a node lying between a node that non-locally references "attr" and the node containing "attr".

These restrictions insure that the identity definitions will not conflict with any definitions in the original attributed grammar. They will be discussed later in this section.

```
procedure AddIdentity (Goal:symbol, attr:attribute;
                       k:production; donor:boolean);
var
    s : stack of production;
    Used : set of production;
    P : set of productionPair;
    procedure Walkup; (* see Figure 5.3 *)
begin
    s := empty;
    push k on s;
    P := empty;
    Used := [LHS[k]];
    Walkup;
    if donor then
        Add the defintions B.attr = A.attr
        for all occurrences of B.attr in production i
        where the pair (i,j) is in P,
        A = LHS[j] and B = LHS[i];
    else
        Add the definition A.attr = B.attr
        for all occurrences of B.attr in production i
        where the pair (i,j) is in P,
        A = LHS[j] and B = LHS[i];
    fi
end;
```

Figure 5.2: Procedure AddIdentity


```
procedure WalkUp;
var
    Ancestor : set of production;
    x : production;
    i : integer;
begin
    Ancestor := [x | LHS[top(s)] in RHS[x]] - Used;
    if Ancestor is empty then
        print ('Error - no occurrence of',Goal,
                        'on the path:');
        for i := 1 to |s| do print (s[i]) od
    else
    for each x ∈ Ancestor do
        push x on s;
        Used := Used ∪ LHS[x]
        if LHS[x] = Goal then
        (* the stack has a path of productions
            that link p to Goal *)
            for i := 1 to |s|-1 do
                P := P ∪ (s[i+1],s[i])
        else WalkUp
        fi;
        Used := Used - LHS[x];
        pop s;
    od
end; (* WalkUp *)
```

Figure 5.3: Procedure Walkup

Theorem 5.4

Given a attributed grammar G containing non-local attribute references that do not violate Restrictions 5.2 and 5.3, AddIdentity produces an attributed grammar G' containing no non-local attribute references such that

1) if there is an attributed parse tree in G' for a program Prog, then we can delete the attributes not used in G and have an attributed parse tree for Prog in G,
2) any attributed parse tree for Prog in G can be modified by adding attributes along a path from an attribute to its non-local reference (copying its value). The resulting parse tree is an attributed parse tree for Prog in G'.

Proof

1) AddIdentity only adds identity definitions that link an occurrence of an attribute to its non-local use. The attributes that are added all have the same value as the non-local attribute, so the use of any of their values is equivalent to the use of the non-local attribute's value. Thus if they were deleted we would leave attributes and values consistant with those of an attributed parse tree in G.

2) AddIdentity adds exactly those attribute definitions along a path from the use of a non-local attribute to its occurrence. Regardless of whether inherited or synthesized attributes were added, the result of AddIdentity is to introduce attributes equal to the occurrence of a non-locally referenced attribute, so we can add the attributes described in part 2 of the theorem to an attributed parse tree in G and get an attributed parse tree in G'. ⊠

We give an example of AddIdentity by removing the non-local references to codesect.table in definition 3.1 of Figure 5.1. The call to AddIdentity is AddIdentity(codesect,start,3,true). The result of WalkUp is the set {(1,2),(1,3),(2,3)}. AddIdentity changes 3.1 to

3.1' inst.begin = list.start

and adds the definitions

1.2 list.start = codesect.start

3.2 list'.start = list.start.

A problem arises if a non-local attribute appears as a recipient in more than one production. In that case AddIdentity might add two definitions,

A.nonlocal = C.atr1

A.nonlocal = D.atr2.

In general, atr1 and atr2 are not equal in value, so we have

a potential race condition. Therefore an evaluator that allows non-local attribute references must detect when an attribute has multiple (distinct) definitions, and issue a warning of a possible race condition (or forbid its use).

A second problem occurs when a non-local attribute is referenced as both a donor and a recipient. In Figure 5.1 Codesect.table appears as both a donor and a recipient in definitions 4.2 and 5.2. AddIdentity assumes that a symbol is either a donor or a recipient, but not both. One set of identity definitions will not suffice, since we need to add definitions of both inherited and synthesized attributes. Rather than trying to modify AddIdentity to add 2 sets of definitions (and connect them to get updated tables (synthesized use) broadcast back into the tree (inherited use)), we prefer the approach described below.

5.3 Attribute Reference by Pointer

We can clearly reference non-local attributes by passing a pointer (to the non-local attribute) along the parse tree, rather than passing the attribute itself. Pointers can yield a savings in both time and space. The space sav-

ing occurs when large attributes (such as tables) are being non-locally referenced. It should be much faster to copy a pointer than to copy a large data structure. Pointers are also of benefit to the reevaluation mechanism. An attribute may be arbitrarily far away from its non-local use (due to self-embedding productions like LIST ::= LIST ; ST) and it may be reevaluated several times. Once the pointer has been defined, the non-local reference directly references the attribute value, rather than waiting for a new copy to be passed down the tree. Thus pointers can reduce the overhead of reevaluation.

AddIdentity no longer needs to consider whether the new defiitions are for an inherited or a synthesized attribute. In either case we copy the pointer to the attribute occurrence as an inherited attribute, since we are passing it into a subtree. The problem that arose when an attribute was used as both a donor and a recipient attribute disappears. The evaluator merely references or defines that attribute through the pointer.

This approach requires two changes to the standard evaluation mechanisms. We must insure that a pointer only references an attribute that has been defined. This can be insured by considering the pointer assignment to be an implicit reference to the attribute to which the pointer

refers. With this change the usual mechanisms that use attribute availability to schedule attribute evaluation will work properly. (The pointer will not be considered defined until the attribute that it references has been defined.) We can then guarantee that pointers always refer to attributes that have been given values.

Pointers also require a change to standard evaluators to accomodate reevaluation. An evaluator using pointers must be modified to build lists of all non-local donor references to an attribute, so that if a non-local donor changes value, all of the definitions that depend of its value will be reevaluated. (These lists are analogous to the lists of section 3.2 for each entry in a table.) If a non-local recipient use is made of an attribute no list entry is needed. If this attribute changes value, its reference list is used to queue reevaluation of other attributes.

Chapter 6: APPLICATIONS

It was stated in the introduction that one of the primary reasons for studying time-varying attributes was to be able to express optimizations cleanly. This chapter gives examples of algorithms that are commonly used in program optimization. These are not new algorithms, but rather they are known algorithms placed in non-procedural form. It is hoped that these examples will demonstrate that the use of time-varying attributes does lead to concise, readable specification of iterative algorithms.

6.1 Optimization Algorithms

Optimization methods can be divided into two categories: local and global. Local optimization attempts to optimize only a small fraction of the program, and never operates on segments of code that contain conditional branches. Examples of local optimization algorithms are the use of immediate load instructions when possible, and rearrangement of an expression to minimize the number of registers that it uses. Global optimization operates on non-

straight-line code, and uses information about the program flow to improve the generated code. Examples of global optimizations include the detection of common subexpressions in different parts of a program, the removal of redundant calculations, and the movement of an expression out of a loop (if its value is loop-invariant). This chapter will discuss several global optimization algorithms. In order to use traditional global optimization algorithms, it is necessary to construct the program flow graph.

Definition 6.1

A program flow graph is a directed graph whose nodes are basic blocks (sequences of code not containing branch instructions or labels) and whose edges represent the potential flow of a program among these basic blocks. A reversed program flow graph is a program flow graph with all arcs reversed. An entry node is a node in the program flow graph with no incoming arcs. An exit node is a node with no arcs leaving it.

A program flow graph can be built for arbitrary code sequences, and this is the first step in many program optimization routines. This construction requires a pass over the entire compiler output (in intermediate code). The parse tree, however, can contain all of the information needed for optimization algorithms, if attributes are used

to gather the flow information. When using attributed grammars to specify optimization algorithms, we will modify algorithms to operate on nodes of the parse tree, rather than basic blocks. The flow information is easy to collect if we assume that the language being parsed is "structured" (by structured we mean that program control is done via structured constructs rather than arbitrary GOTO statements), although arbitrary GOTO's could be handled (with more overhead) via a label table. By using attributes we can skip the extra pass over the compiler output, and do code improvement directly on the parse tree.

Optimization algorithms often require knowledge of the set of nodes that precede or follow the current node, in the dynamic sense. These sets allow information about variable usage, common subexpressions, constant expressions, etc., to be passed around the program flow graph. They can be determined using attributes that have nodes (or more correctly, the names of nodes) as values.

The use of nodes as possible values for an attribute is potentially dangerous because there can be more than one attribute instance involved in defining or referencing a value. For example, if "predecessor" and "pool" are attributes of ST (a statement node), and predecessor ranges over nodes, ST.pool, ST.predecessor.pool and

ST.predecessor.predecessor.pool are all valid attribute occurrences. Unfortunately, an attribute occurrence of this form wreaks havoc when we try to determine attribute availability. The problem is to determine when all occurrences of predecessor, and the pool that they reference, are available. It is not sufficient to know that predecessor is available; we must know that both predecessor and predecessor.pool are available. There is no way of determining such availability in advance, as is done with plans, since the values of predecessor are not determined until the input string has been parsed. To avoid problems concerning attribute availability, a restriction is suggested on the use of node-valued attributes:

> Attributes of node-valued attributes must have initial values.

Thus we can be sure that whenever a node-valued attribute is defined, the attributes referenced through it are also defined.

6.2 Predecessors and Successors

We will use the attributes predecessor and successor for the sets of predecessor and successor nodes. The skeleton of the description of a typical structured programming language follows. It includes the standard structured programming constructs, with the exception of the procedure statement. We use a right recursive definition of STL to allow easy access to both the first and last ST nodes on a list. The EXIT statement is a transfer out of the immediately containing while or repeat loop. In the following grammar, the production STL ::= EXIT is used, rather than ST ::= EXIT. Since there are no GOTO statements, any statements following an EXIT (in the current statement list) would never be executed and can therefore be disallowed. Declarations (<declare> nodes) are not discussed for simplicity.

Successor and predecessor nodes are passed along the tree using semantic knowledge about each control structure. In a list of statements, statements are executed serially, so the node for a statement list relays information between its statement and the next statement list.

STL.previous is the set of nodes that immediately precede this statement list.

STL.first gives the name of the statement that begins this statement list

STL.last is the set of exit nodes for this list.

STL.successor is the set of successor nodes of this statement list.

Selection statements (if-then-else-fi) pass the common predecessor and successor sets to each statement list. Repetition statements (while and repeat) include back loops and a conditional branch around the statement list, so their predecessors and successors have multiple values. EXIT statements require two attributes to assign appropriate predecessors and successors.

STL.exitpoint is inherited by the EXIT statement, and gives the set of successor nodes of the immediately surrounding while or repeat loop (the target of the EXIT).

STL.exitlist is synthesized by the statement list, and gives the names of the predecessors of the EXIT statements (that become predecessors of the nodes following the loop).

In short, all of the above constructs have successors and predecessors that are easy to determine from the parse tree.

```
1. <program> ::= program id <declare> STL end
    1.1 <program>.entry = program.first
    1.2 <program>.exit = end.last
    1.3 program.first = STL.first
    1.4 end.last = STL.last
```



```
2. ST ::=  <if head> then STL else STL' fi
    2.1 ST.last = STL.last ∪ S        TL'.last
    2.2 ST.exitlist = STL.exitlist ∪ STL'.exitlist
    2.3 <if head>.predecessor = ST.predecessor
    2.4 <if head>.successor = STL.first ∪ STL'.first
    2.5 STL.exitpoint = ST.successor
    2.6 STL.previous = <if head>
    2.7 STL.successor = ST.successor
    2.8 STL'.exitpoint = ST.successor
    2.9 STL'.previous = <if head>
    2.10 STL'.successor = ST.successor

3. ST ::=  <while head> do STL od
    3.1 ST.last = <while head> ∪ STL.exitlist
    3.2 ST.exitlist = Ø
    3.3 <while head>.predecessor = ST.predecessor ∪ STL.last
    3.4 <while head>.successor = STL.first ∪ ST.successor
    3.5 STL.exitpoint = ST.successor
    3.6 STL.previous = <while head>
    3.7 STL.successor = <while head>

4. ST ::=  repeat STL <until part>
    4.1 ST.last = <until part> ∪ STL.exitlist
    4.2 ST.exitlist = Ø
    4.3 STL.exitpoint = ST.successor
    4.4 STL.previous = ST.predecessor ∪ <until part>
    4.5 STL.successor = <until part>
    4.6 <until part>.predecessor = STL.last
    4.7 <until part>.successor = ST.successor ∪ STL.first

5. ST ::=  id := EXP
    5.1 ST.last = ST
    5.2 ST.exitlist = Ø

6. STL ::= ST ; STL'
    6.1 STL.first = ST
    6.2 STL.last = STL'.last
    6.3 STL.exitlist = ST.exitlist ∪ STL'.exitlist
    6.4 ST.predecessor = STL.previous
    6.5 ST.successor = STL'.first
    6.6 STL'.previous = ST.last
    6.7 STL'.successor = STL.successor
```

```
7. STL ::= ST
    7.1 STL.first = ST
    7.2 STL.last = ST.last
    7.3 STL.exitlist = ST.exitlist
    7.4 ST.successor = STL.successor
    7.5 ST.predecessor = STL.previous

8. STL ::= EXIT
    8.1 STL.first = STL.exitpoint
    8.2 STL.last = Ø
    8.3 STL.exitlist = STL.previous


I(ST) = {predecessor,successor}
S(ST) = {exitlist,last}

I(STL) = {exitpoint,previous,successor}
S(STL) = {exitlist,first,last}

I(<if head>) = {predecessor,successor}
S(<if head>) = Ø

I(<while head>) = {predecessor,successor}
S(<while head>) = Ø

I(<until part>) = {predecessor,successor}
S(<until part>) = Ø
```

Figure 6.1: Predecessors and Successors

6.3 Constant Propagation

An algorithm for constant progagation is described by Kildall in [Kil 73 p.194]. The algorithm builds sets of

(identifier, constant value) pairs at each statement node, then refines these sets by propagating them along the parse tree. When done, an (identifier, constant value) pair indicates that an identifier must have the corresponding constant value at the beginning of the statement. Rather than repeating all of the details of the algorithm, we quote the informal description from [Kil 73 p.195].

a) Start with an entry node in the program graph, along with a given entry pool corresponding to this entry node. Normally, there is only one entry node, and the entry pool is empty.

b) Process the entry node, and produce optimizing information (in this case, a set of propagated constants) which is sent to all immediate successors of the entry node.

c) Intersect the incoming optimizing pools with that already established at the successor nodes (if this is the first time the node is encountered, assume the incoming pool as the first approximation and continue processing).

d) Considering each successor node, if the amount of optimizing information is reduced by this intersection (or if the node has been encountered for the first time) then process the successor in the same manner as the initial entry node (the order in which the successor nodes are processed is unimportant).

Figure 6.2: Kildall's Algorithm

Constant propagation can be implemented by adding attributes to ST and <until part>. (<if head> and <while head> will use the attribute of their associated statement

node since they neither create nor destroy (id,const) pairings). The pool will be represented by a vector of values indexed by each identifier in the program. Operations on the pool are done in parallel for each component. The optimization operator will be represented by $\wedge$, the meet of a semi-lattice of the values. The values range over a set that is appropriate for the type of the identifier, augmented with $\{\top,\bot\}$, Scott's top and bottom. Xval (extended Val) = Val $\cup$ $\{\top,\bot\}$ forms a flat semilattice with $\wedge$:

$x \wedge \top = x \;\forall\; x \in Xval$

$x \wedge \bot = \bot \;\forall\; x \in Xval$

$x \wedge x = x \;\forall\; x \in Xval$

$x \wedge y = \bot \;\forall\; x,y \in Val,\; x \neq y.$

Intuitively the pair (x,c) in a node's pool represents the information that x may be assumed to contain the constant value c at that node. The pair $(x,\bot)$ indicates that x cannot be assumed to contain a constant value, either because x is assigned different values on different paths to the node (its value is "variable") or because x has never been assigned any value (its value is "undefined"). $(x,\top)$ means that the algorithm has not yet determined anything about x's value. Each node will have an initial constant pool called **U**, in which every identifier has the value $\top$.

Kildall's algorithm states that the entry node must be given a special empty set, in which all identifiers have the

value $\bot$. We will assign such a set to <program>.entry.pool. Constant pools are passed along the successor links. As processing continues, three things might happen:

a) An identifier may be given a constant value, so that its value field is changed when it is passed to the successors. The function INSERTVAL(pool,id,val) returns the constant pool, replacing the old extended value associated with id by "val".

b) An identifier may be removed from the pool by the intersection of pools from multiple predecessors, by assigning a non-constant value to it or by any other use that can modify an identifier's value (e.g. read statements or reference parameters). If an identifier is removed from the pool, it is defined, but it has no known constant value. INSERTVAL(pool,id,$\bot$) removes an id from the constant pool.

ST and <until part> have an inherited attribute <u>pool</u>, the set of (identifier, value) pairs in effect at the beginning of this statement node (initially **U**).

EXP.<u>isconstant</u> is a boolean that is true iff the expression consists entirely of constants (i.e. all identifiers used in the expression are in the constant pool).

EXP.<u>value</u> contains the constant value if EXP.isconstant is true. If any identifier in the expression has the value "undefined", the value of the expression is

"undefined".

These definitions use a shorthand notation for manipulating attributes that are sets of nodes. When a set name.attribute is used, it means to apply the definition repeatedly to each element in the set.

```
1.5 <program>.entry.pool = {pool with all ⊥}

2.11 <if head>.successor.pool =
        <if head>.successor.pool  ∧ ST.pool

3.8 <while head>.successor.pool =
        <while head>.successor.pool  ∧ ST.pool

4.8 <until part>.successor.pool =
        <until part>.successor.pool  ∧ <until part>.pool

5.3 ST.successor.pool = if EXP.isconstant then
      ST.successor.pool ∧ INSERTVAL(ST.pool,id,EXP.value)
    else
      ST.successor.pool ∧ INSERTVAL(ST.pool,id,⊥)
    fi

now I(ST) = {pool=U,predecessor,successor}
    S(ST) = {last}

I(<if head>) = {predecessor,successor}
S(<if head>) = Ø

I(<while head>) = {predecessor,successor}
S(<while head>) = Ø

I(<until part>) = {pool=U,predecessor,successor}
S(<until part>) = Ø
```

Figure 6.3: Constant Propagation

This definition gives the invariant relation among all constant pools: a node's constant pool must be contained in the intersection of all of its predecessor's constant pools.

Since EXP begin by finding all simple constants, and will check each expression as a new constant pool is formed, we will end up with a maximal pool of constants. A proof of this is given in [Kil 73 p.197], where one can also find proofs of halting and uniqueness of the result.

Statement list and EXIT nodes are not directly involved in constant propagation. Their contribution has been to establish links among the the predecessor and successor nodes. Once these links have been defined, the statement lists and EXITS are no longer involved in the optimization functions.

It is significant that the definition is stated in terms of successor.pool, rather than predecessor.pool. An alternate definition might be

```
5.3' ST.pool = if EXP.isconstant then
        INSERTVAL(intersection of all predecessor.pool,
                  id,EXP.value)
        else
            intersection of all predecessor.pool
        fi
```

This version requires that the evaluator automatically establish a reverse link between nodes and their successors, such that if a pool is changed, all attributes that depend on it are also changed. Our algorithm explicitly specifies a two-way link using successor and predecessor, but there is no way for the evaluator to know that they are related, unless we include dynamic links to donors, as was done with

one solution to non-local attribute reference. Such a feature is unnecessary, since we have a non-local recipient. We know that we have potentially changed the value of successor.pool, and if a change occurred, we can look at the successor node to propagate the changes.

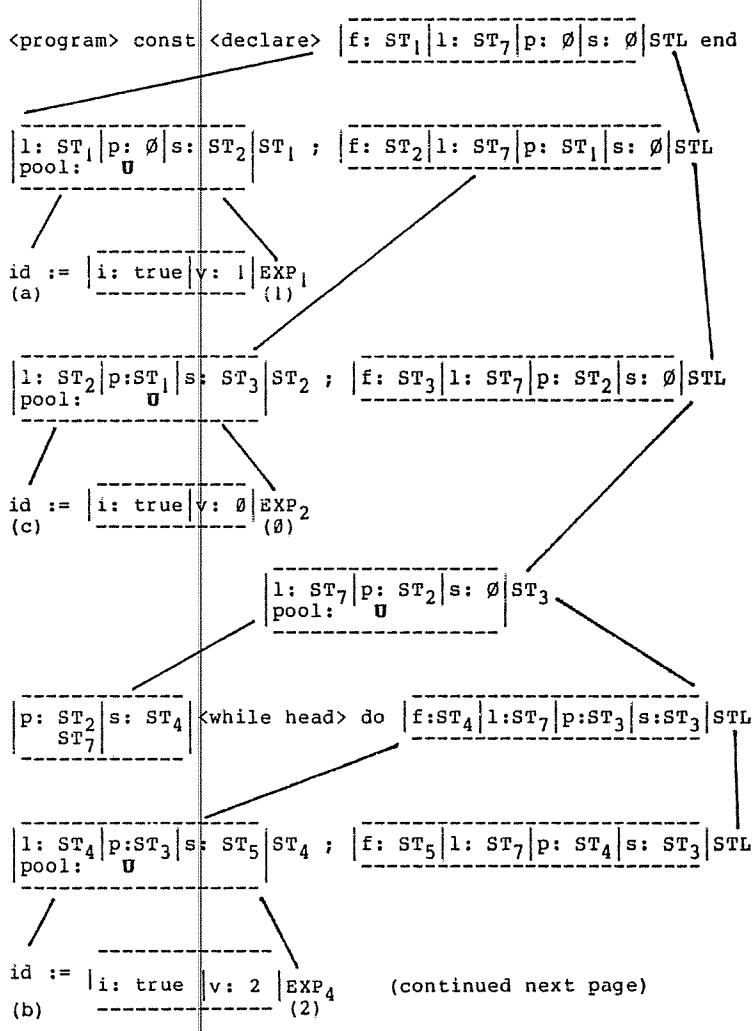
6.4 Example

We will give the example of constant propagation of Kildall [Kil 73 p.194] in attribute form. He used an example that included a loop, which we have represented by a while loop. The reevaluation begins when <program>.entry.pool is given a null pool. Since the successor will use its pool in calculating its successors' pools, the constant information will propagate along successors by means of the reevaluation mechanism. We assume that the constant pools are passed into the expression subtree, so that constant expressions can be detected. Such definitions are not explicitly shown, since the structure of an expression is not detailed. With appropriate definitions, the subtree can reevaluate its attributes and discover constant expressions that might propagate back to the highest

level expression. EXP.isconstant and EXP.value would receive appropriate values so that new constants would be added to the pool.

The diagram that follows represents the state of the evaluation after predecessor and successor sets have been calculated, and after the initial expression analysis, but before the constant propagation has begun. Some attribute names have been abbreviated to their first letter. We will represent the constant pools by a set of (id,value) pairs rather than by a vector, to make the example more readable. By convention, any pairs of the form (id,$\perp$) will be omitted from the constant pools, since they represent non-constant values. The program being analyzed is:

```
a := 1;
c := 0;
while a>c do
    b := 2;
    d := a+b;
    e := b+c;
    c := 4
od;
```



(continued next page)

Figure 6.4: Constant Propagation Tree (part 1)

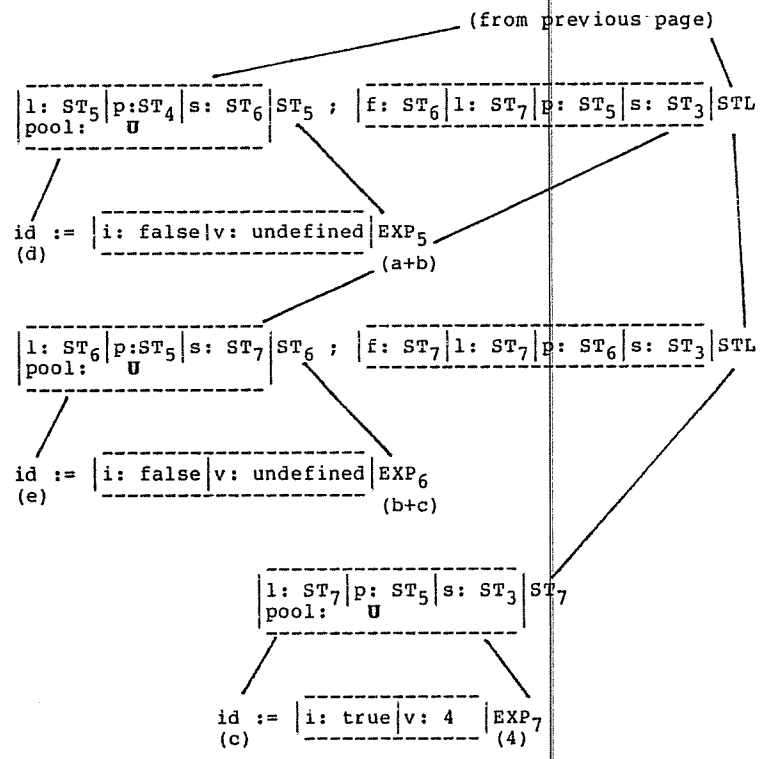


Figure 6.5: Constant Propagation Tree (part 2)

Constant propagation begins when ST_1 is given the pool

$\{(a,\perp),(b,\perp),(c,\perp),(d,\perp),(e,\perp)\}$, written $\{\}$.

A change of value has occurred, and EXP_1 and ST_1 are queued (since they both have definitions that use ST_1.pool).

EXP_1 is removed from the queue.

EXP_1.isconstant and EXP_1.value are unchanged.

ST_1 is removed from the queue.

ST_1.successor is ST_2 and EXP_1.isconstant is true,

so ST_2.pool gets $\{(a,1)\}$.

EXP_2 and ST_2 are enqueued.

EXP_2 is removed from the queue.

EXP_2.isconstant and EXP_2.value are unchanged.

ST_2 is removed from the queue.

ST_3.pool gets $\{(a,1),(c,0)\}$.

ST_3 is enqueued.

ST_3 is removed from the queue.

ST_4.pool gets $\{(a,1),(c,0)\}$ by (3.8).

EXP_4 and ST_4 are enqueued.

EXP_4 is removed from the queue.

EXP_4.isconstant and EXP_4.value are unchanged.

ST_4 is removed from the queue.

ST_5.pool gets $\{(a,1),(b,2),(c,0)\}$.

EXP_5 and ST_5 are enqueued.

EXP_5 is removed from the queue.

EXP_5.isconstant gets true and EXP_5.value gets 3.

ST_5 is removed from the queue.

ST_6.pool gets $\{(a,1),(b,2),(c,0),(d,3)\}$.

EXP_6 and ST_6 are enqueued.

EXP_6 is removed from the queue.

EXP_6 gets true,

EXP_6 gets 2.

ST_6 is (redundantly) enqueued (no change in the queue).

ST_6 is removed from the queue.

ST_7.pool gets $\{(a,1),(b,2),(c,0),(d,3),(e,2)\}$.

EXP_7 and ST_7 are enqueued.

EXP_7 is removed from the queue.

EXP_7.isconstant and EXP_7.value are unchanged.

ST_7 is removed from the queue.

$$ST_3.\text{pool gets } \{(a,1),(c,0)\} \wedge \{(a,1),(b,2),(c,4),(d,3),(e,2)\} = \{(a,1)\}$$

A change of value has occurred,

so <while head> and ST_3 are enqueued,

since they use ST_3.pool.

<while head> is removed from the queue.

<while head> uses the new pool,

but doesn't effect any other pools.

ST_3 is removed from the queue.

ST_4.pool is reevaluated and gets $\{(a,1)\}$.

EXP_4 and ST_4 are enqueued.

EXP_4 is removed from the queue.

EXP_4.isconstant and EXP_4.value are unchanged.

ST_4 is removed from the queue.

ST_5.pool is reevaluated and gets {(a,1),(b,2)}.

EXP_5 and ST_5 are enqueued.

EXP_5 is removed from the queue.

EXP_5.isconstant and EXP_5.value are unchanged.

ST_5 is removed from the queue.

ST_6.pool is reevaluated and gets {(a,1),(b,2),(d,3)}.

EXP_6 and ST_6 are enqueued.

EXP_6 is removed from the queue.

EXP_6.isconstant is changed to false.

ST_6 is redundantly enqueued.

ST_6 is removed from the queue.

ST_7.pool is reevaluated and gets {(a,1),(b,2),(d,3)}.

EXP_7 and ST_7 are enqueued.

EXP_7 is removed from the queue.

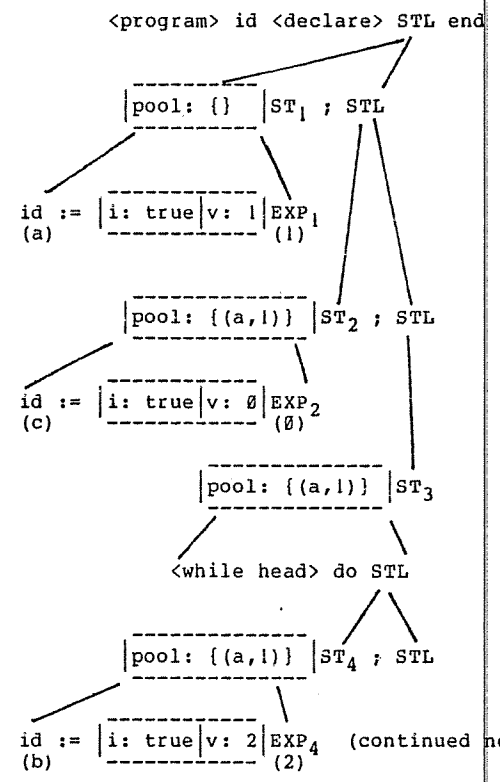
EXP_7.isconstant and EXP_7.value are unchanged.

ST_7 is removed from the queue.

ST_3.pool is reevaluated with no change.

Nothing remains in the queue; reevaluation is over.

The result (omitting unchanged attributes) is:



(continued next page)

Figure 6.6: Constant Propagation Results (part 1)

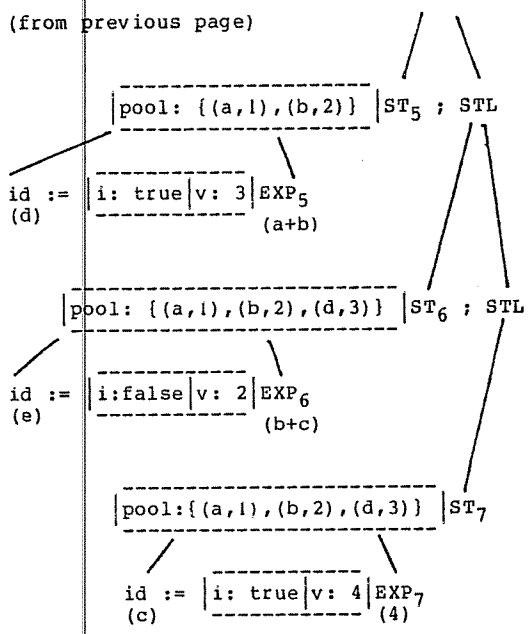


Figure 6.7: Constant Propagation Results (part 2)

6.5 Live Variable Analysis

Another iterative algorithm suggested by Kildall is the analysis of live and dead variables, also called busy variable analysis.

Definition 6.2

A value of the variable is live, or busy, at a given node, if the variable could be referenced in a subsequent expression before it is redefined.

Live/dead analysis can be useful when deciding which of several registers that are in use should be freed; one can always free a register that contains a dead variable. Informally, Kildall defines the optimizing function as follows:

1. If an expression at a node N involves an assignment to a variable, let d be the destination of the assignment, and let P be the pool of live variables that we are creating to propagate to predecessor nodes. Set P to P - {d} (d becomes a dead variable).

2. Consider each variable v used in an expression at node N. Set P to P u {v} (v becomes a live variable).

In this algorithm we are "looking ahead" to find the uses of a variable. To do so we pass information backwards through the program graph; successor nodes report the

expressions that they are "interested in" by merging sets of live variables. Thus, we will use the predecessor attributes to propagate the live expression sets. These pools will be initially empty, so that the initial reevaluation will copy an exit set of busy variables from successor to predecessor. The algorithm states that all exit nodes receive an empty set of busy variables, so we do not need to add an attribute definition to production 1 to initialize <program>.exit.busy. The new attributes are as follows:

ST, <if head>, <while head>, and <until part> have an attribute busy, which is the set of live variables known to be busy at the exit of the node (initially empty).

EXP, <if head>, <while head> and <until part> have the attribute used, the set of all variables used in this node.

ST, <if head>, <while head> and <until part> have the attribute outbusy, the set of all live variables after the effects of this node have been calculated. These results are passed forward to predecessors.

The definitions for this algorithm are:

```
2.11 <if head>.outbusy =
        <if head>.busy u <if head>.used
2.12 <if head>.predecessor.busy =
        <if head>.predecessor.busy u <if head>.outbusy

3.8 <while head>.outbusy =
        <while head>.busy u <while head>.used
3.9 <while head>.predecessor.busy =
        <while head>.predecessor.busy
        u <while head>.outbusy
```



```
4.8 <until part>.outbusy =
        <until part>.busy u <until part>.used
4.9 <until part>.predecessor.busy =
        <until part>.predecessor.busy u
        <until part>.outbusy

5.3 ST.outbusy = (ST.busy - id) u EXP.used
5.4 ST.predecessor.busy =
        ST.predecessor.busy u ST.outbusy

I(ST) = {busy=Ø,predecessor,successor}
S(ST) = {last,newbusy}

I(<if head>) = {busy=Ø,predecessor,successor}
S(<if head>) = {last,newbusy}

I(<while head>) = {busy=Ø,predecessor,successor}
S(<while head>) = {last,newbusy}

I(<until part>) = {busy=Ø,predecessor,successor}
S(<until part>) = {last,newbusy}

I(EXP) = Ø
S(EXP) = {used}
```

Figure 6.8: Live Variable Specification

New live expressions are generated by the EXP, <if head>, <while head> and <until part> nodes, so productions two through five need a definition that adds elements to busyset. These sets are merged with the predecessor's set, giving a set of variables used on any path. Only production five can redefine (kill) a variable, so its definition also removes elements from its contribution to its predecessor's busy set. The other productions are unchanged since they do not generate or use variables or expressions.

6.6 Very Busy Variable Analysis

We can define another class of variables that are similar to busy variables, called very busy variables. A variable is very busy if it is used along all paths from the node in question before it is redefined.

Not surprisingly, the definitions needed to compute sets of very busy expressions are nearly the same as those for busy variable analysis. The only change is to make sure that the variable is referenced along all successor paths. We will need two new attributes:

ST, <if head>, <while head>, and <until part> have an attribute verybusy, a set of very busy variables available at this node. They also have an attribute called outverybusy, which is the set of very busy variables to be merged with the predecessor's set.

We will again use the reversed program graph, and the predecessor attribute. All that changes is that we must intersect, rather than merge, the sets of predecessor verybusy sets, and that the pools must be initially **U**, rather than Ø. These changes give sets of variables that occur over all successor paths, rather than variables that occur on some path. Very busy sets are generated because if a variable is not used along some path, the intersection of verybusy and

outverybusy will remove the variable from verybusy. The exit nodes must receive a null set of very busy variables in production one, since they are initialized to **U**, and must have no very busy variables. The new definitions are:

```
1.5 <program>.exit.verybusy = Ø

2.11 <if head>.outverybusy =
        <if head>.verybusy u <if head>.used
2.12 <if head>.predecessor.verybusy =
        <if head>.predecessor.verybusy ∧
        <if head>.outverybusy

3.8 <while head>.outverybusy =
        <while head>.verybusy u <while head>.used
3.9 <while head>.predecessor.verybusy =
        <while head>.predecessor.verybusy ∧
        <while head>.outverybusy

4.8 <until part>.outverybusy =
        <until part>.verybusy u <until part>.used
4.9 <until part>.predecessor.verybusy =
        <until part>.predecessor.verybusy ∧
        <until part>.outverybusy

5.3 ST.outverybusy = (ST.verybusy - (EXP.def u id)) u
                     EXP.used
5.4 ST.predecessor.verybusy =
        ST.predecessor.verybusy ∧ ST.outverybusy

now I(ST) = {verybusy=U,predecessor,successor}
    S(ST) = {last,newbusy}

    I(<if head>) = {verybusy=U,predecessor,successor}
    S(<if head>) = {last,newbusy}

    I(<while head>) = {verybusy=U,predecessor,successor}
    S(<while head>) = {last,newbusy}

    I(<until part>) = {verybusy=U,predecessor,successor}
    S(<until part>) = {last,newbusy}

    I(EXP) = Ø
    S(EXP) = {def,used}
```

Figure 6.9: Very Busy Variable Specification

6.7 Availability Analysis

There is a problem dual to very busy variable analysis: we can propagate usage in the forward direction via available expressions.

Definition 6.3

An expression is available at a node p if every path from the initial node to p evaluates the expression, and, after the last such evaluation prior to reaching p, there are no subsequent assignments to any variable in the expression.

The new attributes are identical in purpose to those in very busy variable analysis; only the names and the direction have been changed.

ST, <if head>, <while head> and <until part> have the attribute avail, the set of available expressions at the head of this statement (initially $\mathbb{U}$).

ST, <if head>, <while head> and <until part> have the attribute outavail, the set of available expressions that is to be sent to all successors.

EXP, <if head>, <while head> and <until part> have the attribute subexp, the set of all subexpressions conputed in this node.

Propagation of available variable sets is begun by giving the entry nodes an empty available variable set. The definitions for available expressions are:

```
1.5 <program>.entry.avail = Ø

2.11 <if head>.outavail =
        <if head>.avail u <if head>.subexp
2.12 <if head>.successor.avail =
        <if head>.successor.avail ∧
        <if head>.outavail

3.8 <while head>.outavail =
        <while head>.avail u <while head>.subexp
3.9 <while head>.successor.avail =
        <while head>.successor.avail ∧
        <while head>.outavail

4.8 <until part>.outavail =
        <until part>.avail u <until part>.subexp
4.9 <until part>.successor.avail =
        <until part>.successor.avail ∧
        <until part>.outavail

5.3 ST.outavail = ST.avail -
        (all expressions involving id) u EXP.subexp
5.4 ST.successor.avail =
        ST.successor.avail ∧ ST.outavail

now I(ST) = {avail=U,predecessor,successor}
    S(ST) = {last,newavail}

    I(<if head>) = {avail=U,predecessor,successor}
    S(<if head>) = {last,newavail}

    I(<while head>) = {avail=U,predecessor,successor}
    S(<while head>) = {last,newavail}

    I(<until part>) = {avail=U,predecessor,successor}
    S(<until part>) = {last,newavail}

    I(EXP) = Ø
    S(EXP) = (subexp,def)
```

Figure 6.10: Available Expression Specification

Available expression information is used to remove redundant expression evaluation. If an expression is available and is used at a node, it need not be recalculated (since its value is still valid along all paths from the entry node). If an expression is not available along some path, it will be removed from successor nodes by the intersection of a set not containing the expression.

Chapter 7: Summary

7.1 Conclusions

A goal of this research was to extend a known technique, attributed grammars, to be more easily used by designers, implementors and users of programming languages. Attributed grammars were chosen as a starting point because there exist compiler generators that use attributed grammars as the means of specifying a compiler. Attributed grammars are extended by considering them to be a system of equations to be satisfied by the evaluator, rather than a list of assignment statements to be executed. This change in thought allows the possibility of meaningful circularity of attribute dependence, leading to time-varying attribute values and iterative algorithms.

Time-varying attributed grammars can be evaluated by methods that are based on evaluators developed by other investigators. The changes to their static evaluators are small: They include a run-time check for a changed attri-

bute value and a provision for initial values. An important feature of the evaluators for time-varying attributed grammars is that multiple passes are not made over the parse tree to correct for changed values. Only those attributes that depend on an attribute that has changed value are reevaluated.

Two evaluators for time-varying attributed grammars were discussed: the first is similar to Fang's, the second is based on Kennedy and Warren's evaluator. There is good reason to believe that the changes could be applied to other evaluation techniques, such as Rowland's [Row 77].

Replacing the idea of writing assignment statements with a non-procedural definition of the relations among attributes has a cost associated with it. It becomes possible to write attributed grammars that are not well-defined, and can have results that are dependent on the order of attribute evaluation. The latter problem is unique to time-varying attributed grammars.

A second extention to attributed grammars is the use of non-local symbols and tables. Non-local symbols lead to specifications that are easier to read than specifications written with only local symbols. Both non-local symbols and tables reduce the time and space overhead of reevaluation.

7.2 Directions for Future Research

This research presents a new view of attributed grammars, their evaluation and applications. Reference is made to work done by Rowland on an evaluator that works in parallel with a parsing algorithm. It seems likely that Rowland's evaluator could be modified to accept time-varying attributed grammars. The major unexplored area is to decide whether or not subtrees can be safely discarded, given that reevaluation may take place.

A means of detecting potential evaluation-order-dependent results in a time-varying attributed grammar needs to be developed. Such an algorithm (or set of restrictions) would make the use of time-varying attributed grammars less prone to errors. An approach that may prove fruitful is to build a dependency graph as in Kennedy and Warren and look for two conditions: multiple time-varying donors in a definition, and dependency cycles that have more than one attribute with an initial value. The latter condition insures that reevaluation will begin at a known point.

State-driven plans for reevaluation could be used to reduce the overhead of the run-time reevaluation procedure. Such plans could take into account each reevaluation path,

assuming that an attribute changes value. As soon as an attribute does not change value the current plan would be terminated and an unfinished reevaluation path could be followed.

Further investigation of techniques for minimizing the time and space requirements of evaluators (as was done with non-local variables and tables) is necessary to make attributed grammars more practical.

More code optimization algorithms could be specified. Included would be heuristics to register allocation, reordering of expressions to minimize the number of registers used, and strength reduction in FOR loops.

The usefulness of time-varying attributes will only be proven through experience. Therefore an implementation of an evaluator of time-varying attributed grammars is important to further research. Simple optimization experiments could point out weaknesses in the evaluator and the specifications of algorithms. Branch length optimization can be done without delving into the specification of a complete programming language, and would provide a good test of the capability to reevaluate attributes. Further code optimizations could be attempted on the intermediate code, in order to keep the task manageable. As experience is gained in both the use and the evaluation of time-varying attributed

grammars, one hopes that they will become a useful tool in both the study of programming languages and their practical implementations.

BIBLIOGRAPHY

[AC 72] Allen, F.E., and J. Cocke. "A Catalogue of Optimizing Transformations," in Design and Optimization of Compilers, edited by Randall Rustin. Prentice-Hall, Englewood Cliffs, New Jersey, 1972, 1-30.

[AU 73] Aho, A.V. and J.D. Ullman, The Theory of Parsing, Translation and Compiling, 1&2. Prentice Hall, Englewood Cliffs, N.J. (1973).

[AU 77] ------, Principles of Compiler Design. Addison-Wesley, Reading, Massachusetts (1977).

[Bab 77] Babich, W.A. "High Level Data Flow Analysis Using A Parse Tree Representation of the Program" PhD. thesis, University of North Carolina at Chapel Hill, May 1977.

[Bea 72] Beatty, J.C. "A Global Register Assignment Algorithm," in Design and Optimization of Compilers, edited by Randall Rustin. Prentice-Hall, Englewood Cliffs, New Jersey, 1972, 65-88.

[Boc 76] Bochmann, Gregor V. "Semantic Evaluation from Left to Right," Communications of the ACM, 2 (February 1976), 55-62.

[CM 76] Chirica, L. M. and D. F. Martin. "An Algebraic Formulation of Knuthian Semantics." 17th Annual Symposium of Foundations of Computer Science, IEEE CH1133-8, Houston, TX (October 1976) 127-136.

[Dem 77] Demers, Alan J. "Generalized Left-Corner Parsing," Conference Record of the Fourth ACM Symposium on the Principles of Programming Languages SIGACT-SIGPLAN, Los Angeles, California, January 17-19, 1977, 170-193.

[Fan 72] Fang, Isu. "FOLDS, a Declarative Formal Language Definition System," Technical Report STAN-1972-329, Stanford Jr. University, December 1972, 286pp.

[HU 69] Hopcroft, J.E. and J.D. Ullman, Formal Languages and Their Relation to Automata. Addison Wesley, Reading, Massachusetts (1969).

[JOR 75] Jazayeri, M., W.F. Ogden, and W.C. Rounds. "On the Complexity of the Circularity Test for Attribute

Grammars," Conference Record of the Second ACM Symposium on the Principles of Programming Languages, SIGACT-SIGPLAN, Palo Alto, California, January 20-22, 1975, 119-129.

[JW 75] Jazayeri, M. and Walter, K.G. "Alternating Semantic Evaluator," in ACM '75, Proceedings of the Annual Conference, Edited by J.D.White. Minneapolis, Minnesota, October 20-22, 1975, 230-234.

[Joh 75] Johnsson, R.K. "An Approach to Global Register Allocation," Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University, December, 1975, 133pp.

[Ken 72] Kennedy, Ken. "Index Register Allocation in Straight Line Code and Simple Loops," in Design and Optimization of Compilers, edited by Randall Rustin. Prentice-Hall, Englewood Cliffs, New Jersey, 1972, 1-30.

[KW 76] Kennedy, Ken, and Scott K. Warren. "Automatic Generation of Efficient Evaluators for Attribute Grammars," Conference Record of the Third ACM Symposium on the Principles of Programming Languages, SIGACT-SIGPLAN, Atlanta, Georgia, January 19-21, 1976, 32-49.

[Kil 73] Kildall, Gary A. "A Unified Approach to Global Program Optimization," Conference Record of the ACM Symposium on the Principles of Programming Languages, SIGACT-SIGPLAN, Boston, Massachusetts, October 1-3, 1973, 194-206.

[Knu 68] Knuth, Donald E. "Semantics of Context-free Languages," Math. Systems Theory 2 , 2 (1968), 127-145. [Correction in Math. Systems Theory 5, 1 (1971), 95-96]

[LRS 74] Lewis, P.M., D.J. Rosenkrantz, and R.E. Stearns. "Attributed Translations." Journal of Computer and System Sciences 9, (1974), 279-307.

[Mil 77] Milton, Donn. "Syntactic Specification and Analysis with Attributed Grammars," Ph.D. Thesis. Computer Sciences Department, University of Wisconsin, July 1977.

[Naur 63] Naur, P. (ed.). "Revised Report on the Algorithmic Language ALGOL 60," Communications of the

ACM, Vol. 6, No. 1, Jan. 1963 pp. 1-17.

[NAM 74] Neel, D., M. Amirchahy, and M. Mazaud. "Optimization of Generated Code by Means of Attributes," Gesellschaft für Informatik - 4. Jahrestagung. Lecture Notes in Computer Science, Vol. 26. Springer-Verlag, Berlin, 1974, 247-256.

[Rai 77] Räihä, K-J. "On Attribute Grammars and Their Use in a Complier Writing System," Report A-1977-4, Department of Computer Science, University of Helsinki, Finland, August 1977.

[Rob 77] Robertson, E.L., "Code Generation for Short/Long Address Machines", Technical Summary Report 1779, Mathematics Research Center, University of Wisconsin, August 1977.

[Row 77] Rowland, Bruce R. "Combining Parsing and the Evaluation of Attributed Grammars," Ph.D. Thesis. Computer Sciences Department, University of Wisconsin, June 1977.

[Sch 73] Schaefer, Marvin. A Mathematical Theory of Global Program Optimization. Prentice-Hall, Englewood Cliffs, New Jersey, 1973, 198pp.

[Sch 76] Schulz, W.R. "Semantic Analysis and Target Language Synthesis in a Translator," Ph.D. Thesis. Computer Science Department, University of Colorado, July 1976, 220pp.

[Szy 76] Szymanski, T.G., "Assembling Code for Machines with Span-dependent Instructions", Technical Report 224, Department of Electrical Engineering, Princeton University, December 1976.

[Wil 71] Wilner, W.T., "Declarative Semantic Definition as Illustrated by a Definition of Simula 67". Ph.D. Thesis, Computer Science Department, Stanford University, Stanford, California, June 1971.