Programming Languages

B. Wegbreit Editor

Semantic Evaluation from Left to Right

Gregor V. Bochmann
University of Montreal

This paper describes attribute grammars and their use for the definition of programming languages and compilers; a formal definition of attribute grammars and a discussion of some of its important aspects are included. The paper concentrates on the evaluation of semantic attributes in a few passes from left to right over the derivation tree of a program. A condition for an attribute grammar is given which assures that the semantics of any program can be evaluated in a single pass over the derivation tree, and an algorithm is discussed which decides how many passes from left to right are in general necessary, given the attribute grammar. These notions are explained in terms of an example grammar which describes the scope rules of Algol 60. Practical questions, such as the relative efficiency of different evaluation schemes, and the ease of adapting the attribute grammar of a given programming language to the left-to-right evaluation scheme are discussed.

Key Words and Phrases: attribute grammars, semantics of programming languages, semantic attributes, left-to-right parsing, multipass compilers, semantic evaluation, semantic conditions

CR Categories: 4.10, 4.20, 5.23, 5.24

This work was supported in part by the National Research Council of Canada. Author's address: Department d'Informatique, Université de Montréal, Case Postale 6128, Montreal 101, Quebec, Canada.

1. Introduction

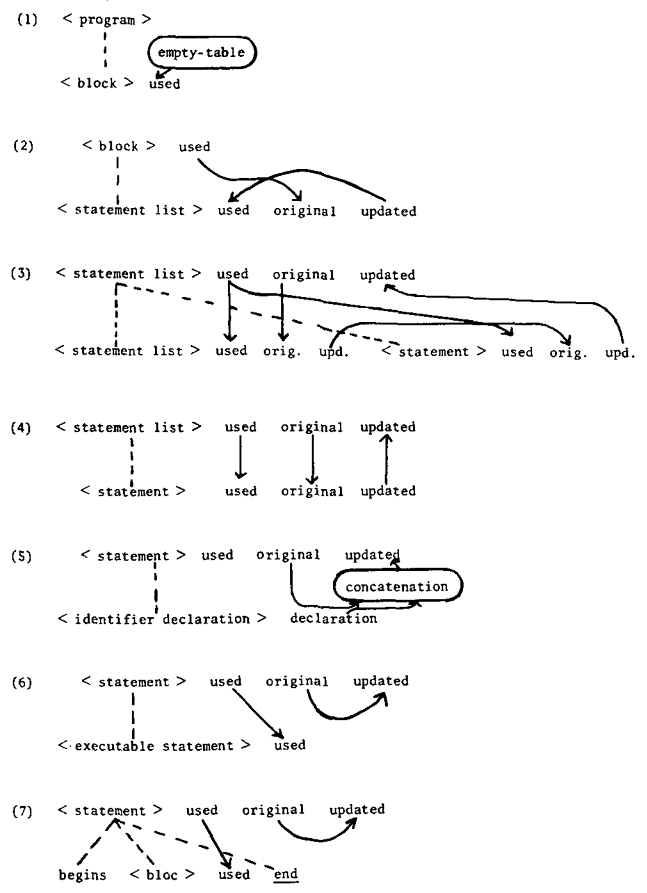
The definition of the semantics of a programming language can be formulated in different ways. Knuth [1, 2] has proposed attribute grammars for this purpose. This kind of language definition consists of two parts: the syntax defined by a context-free grammar and the semantics defined in terms of attributes, associated with the syntactic symbols, and semantic functions which determine the evaluation of the attributes on the derivation tree of a program. This method has several advantages: (a) the semantic description of a language is structured according to the syntax; (b) the context-sensitive features of a programming language can be described; (c) the description of a language can be checked for consistency and used for automatic compiler generation; and (d) different descriptions of the same language may be proven equivalent [3]. The method has been used for a definition of the programming language Simula [4]. Related concepts are property grammars [5], attributed grammars with relations [6], and affix grammars [7].

In Section 2 we give an example of using an attribute grammar for the description of a programming language construct. The given attribute grammar, which describes the Algol scope rules, turns out not to be adequate for an evaluation from left to right. Appropriate changes to the grammar are discussed in the following sections. Section 3 contains a formal definition of attribute grammars and a discussion of some of its important aspects. This section is essentially a review of known results.

In Section 4, we introduce the concept of left-to-right evaluation, and give conditions which allow a semantic evaluation of the derivation tree of any program in a single pass from left to right. In Section 5, several passes from left to right are considered, and an algorithm is given which determines for a given attribute grammar the number of passes necessary. The concept of left-to-right evaluation in several passes, as described in this paper, is related to the practice of writing compilers which do several passes over the internal representation of a program in order to obtain its translation. By formalizing the concept of multipass compilation within the framework of attribute grammars, it is possible to determine the attributes that can be evaluated in each individual pass.

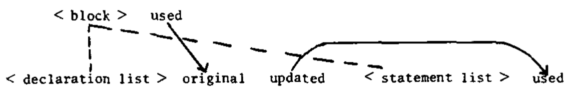
In Section 6, we relate these results to the practice of compiling programming languages. After pointing out the relative efficiency of the left-to-right evaluation scheme compared to more general ones, we discuss the possibility of recasting the semantic definition of a given

Fig. 1(a). Syntactic rules and semantic functions of a simple grammar, discussed in Section 2.



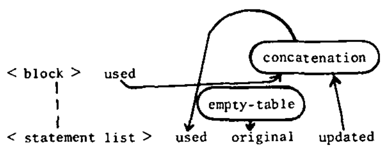
(1) < program >
empty-table
< block > used
(2) < block > used
< statement list > used original updated
(3) < statement list > used original updated
< statement list > used orig. upd. < statement > used orig. upd.
(4) < statement list > used original updated
< statement > used original updated
(5) < statement > used original updated
concatenation
< identifier declaration > declaration
(6) < statement > used original updated
< executable statement > used
(7) < statement > used original updated
begins < bloc > used end

Fig. 1(b). The second production modified in view of a semantic evaluation in a single pass from left to right.



< block > used
< declaration list > original updated < statement list > used

Fig. 1(c). The second production modified such as to allow a semantic evaluation in two passes.



language into a form which allows an evaluation in a few passes from left to right, and the question how difficult (or easy) this may be for the constructs found in real programming languages.

2. Example

We show as an example how an attribute grammar can be used to describe the scope rules of Algol. We consider the following simplified syntax:

(1)	⟨program⟩	→ ⟨block⟩
(2)	⟨block⟩	→ ⟨statement list⟩
(3)	⟨statement list⟩	→ ⟨statement list⟩ ⟨statement⟩
(4)		→ ⟨statement⟩
(5)	⟨statement⟩	→ ⟨identifier declaration⟩
(6)		→ ⟨executable statement⟩
(7)		→ **begin** ⟨block⟩ **end**

Each syntactic symbol may have one or more associated attributes. Each attribute represents a "symbol table," i.e. a list of identifier declarations. The following attribute names are used:

used: attribute of ⟨block⟩, ⟨statement list⟩, ⟨statement⟩, and ⟨executable statement⟩. The value is the symbol table containing all identifier declarations whose scope includes the syntactic symbol concerned, i.e. it contains all identifiers which are valid for the syntactic symbol. The declaration of an identifier is searched starting at the end of the table.

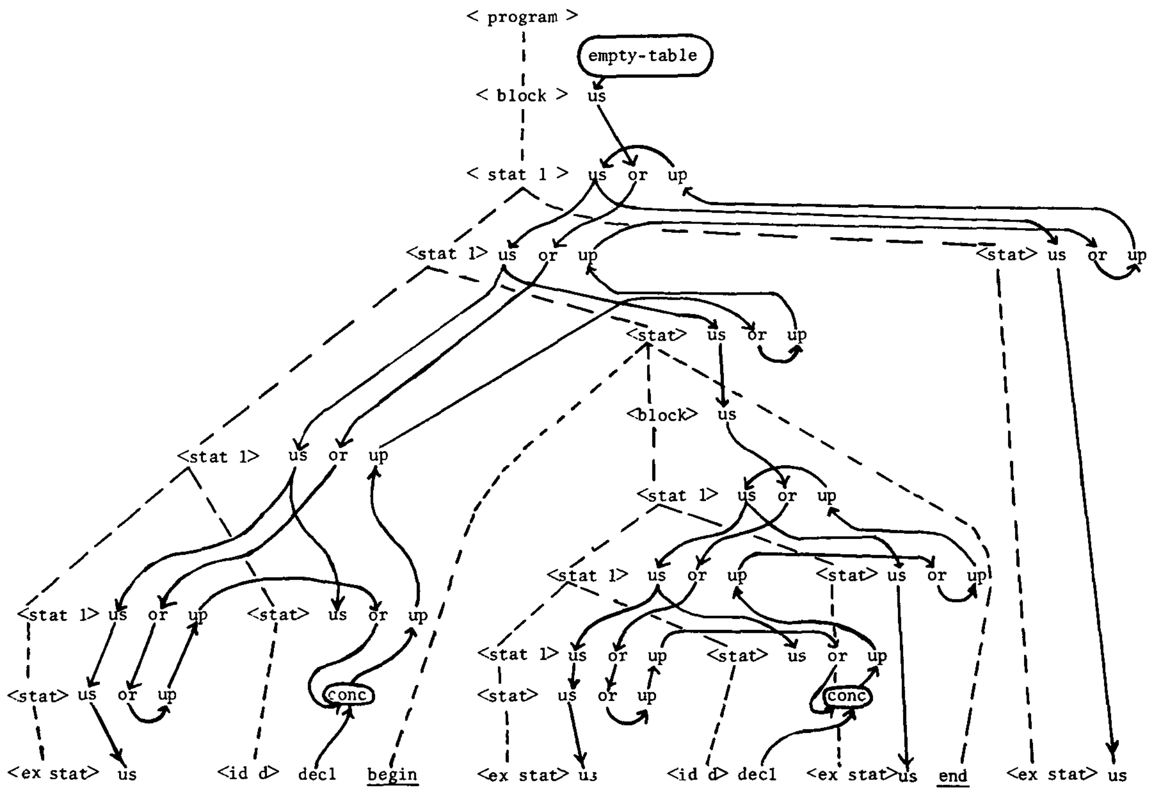
declaration: attribute of ⟨identifier declaration⟩. The value is a new identifier declaration to be appended to the symbol table.

original and updated: attributes of ⟨statement list⟩, and ⟨statement⟩. These attributes can be understood in the following terms: The "*updated*" symbol table of a ⟨statement⟩ is identical to the "*original*" one, except that it contains the new "*declaration*" in the case where the ⟨statement⟩ is an ⟨identifier declaration⟩. Similarly, the "*updated*" symbol table of a ⟨statement list⟩ contains the "*original*" one plus the "*declarations*" which are part of the ⟨statement list⟩.

For each production rule, there are certain evaluation rules that specify how the values of the attribute occurrences are obtained. The evaluation rules for this example are shown in Figure 1(a). The dashed lines indicate the syntactic relation of the symbols, and an arrow between two attribute occurrences indicates that the second attribute will have the same value as the first one. In production rule (1) the "*original*" symbol table is initialized, in production rule (5) a new "*declaration*" is appended to the symbol table, and the information in the "*used*" symbol table is used in the ⟨executable statement⟩ of production rule (6). Figure 2 shows the derivation tree of a sample program and its semantic evaluation according to the rules of Figure 1(a). The arrows indicate how the values of certain attributes depend on the values of the other attributes in the derivation tree. The order of the evaluation of attributes is only indirectly determined by the rules of Figure 1(a). In the case of the derivation tree of Figure 2, one can start by assigning to the "*used*" symbol table of the topmost ⟨block⟩ the value "*emptytable*," and then one can transfer this value to other attributes in the tree by following the arrows. Each occurrence of production (5) within the tree adds a new identifier declaration to the value of the symbol table. The attributes in the subtree of the embedded ⟨block⟩ can only be evaluated after its "*used*" symbol table has obtained its value in the surrounding subtree of the ⟨program⟩.

For this derivation tree, one sees immediately that the "*used*" symbol table conforms with the Algol scope rule for embedded blocks, and it is easy to verify that

Fig. 2. The derivation tree of a sample program showing the evaluation of the attributes. The names of the syntactic symbols and the attributes are abbreviated.



this holds for any derivation tree which is formed according to the rules of Figure 1(a).

This example illustrates the use of an attribute grammar for describing a programming language construct. We come back to this example in the following sections, and show how this grammar can be modified so that the attributes can be evaluated in a few passes from left to right over the derivation tree of a program.

3. Attribute Grammars

Definition

In this section we give a definition of attribute grammars, which is similar to the one in [1], and discuss some points which are important when using such grammars for the definition of programming languages.

An attribute grammar is a context-free grammar augmented with attributes and semantic rules. More precisely, an attribute grammar consists of:

1. A reduced context-free grammar $G_0 = (V_T, V_N, P, S_0)$. The sets V_T of terminal and V_N of nonterminal symbols form the vocabulary $V = V_T \cup V_N$; P is the set of production rules, and $S_0 \in V_N$ is the start symbol, which does not appear on the right side of any production rule. A production rule $p \in P$ is written in the form

p: $X_0 \rightarrow X_1X_2 \cdots X_{n_p}$

where $n_p \geqslant 1, X_0 \in V_N$, and $X_k \in V$ for $1 \leqslant k \leqslant n_p$.

2. A set of attributes. Each attribute can be understood as a data type [8]. For each symbol $X \in V$, there are the disjoint (sub-)sets $I(X)$ of inherited and $S(X)$ of synthesized attributes. We have $I(X) = \emptyset$ for the start symbol and for all terminal symbols. We write $A(X)$ for the union $I(X) \cup S(X)$.

3. The evaluation of the attributes is defined within the scope of a single production: a production p is said to have the *attribute occurrence* (a, k) if $a \in A(X_k)$ where X_k is the kth symbol of p $(k = 0, \cdots, n_p)$. An attribute occurrence (a, k) can be understood as a variable of type a associated with the symbol X_k which can take on *attribute values* according to its type.

For each occurrence (i, k) of an inherited attribute i on the right side of a production p $(k = 1, \cdots, n_p)$, there is an associated *semantic function* $f^{(p)}_{(i,k)}$, and for each occurrence $(s, 0)$ of a synthesized attribute s on the left side of p, there is an associated *semantic function* $f^{(p)}_{(s,0)}$. These semantic functions determine the value for the attribute occurrence as a function of values of certain other attribute occurrences in the same production.

A *dependency set* $D^{(p)}_{(a,k)}$ is the set of attribute occurrences whose values are used for the evaluation of the attribute occurrence (a, k) by the semantic function $f^{(p)}_{(a,k)}$. The possibility that a given semantic function, for certain values of some attribute occurrences in the dependency set, may not depend on the values of some other attribute occurrences in the dependency set is ignored throughout this paper.

We now consider the context-free language $L(G_0)$ generated by the syntax G_0. The analysis of a program in this language can be thought of as being done in two phases, which, under certain conditions (see below), can be performed in a single pass from left to right:

(i) Syntax analysis: the construction of the derivation tree(s) of the program.

(ii) Semantic evaluation: the evaluation of the attribute occurrences at the node symbols of the tree. This evaluation is determined by the semantic functions of the attribute grammar. It is important to note that each occurrence of an inherited attribute at a certain node performs a transfer of information within the derivation tree from the top down (i.e. from the start symbol towards the terminal nodes) since its value is determined by a semantic function of the production above the node, and is (probably) used by some semantic function of the production below. Similarly a synthesized attribute occurrence performs a transfer of information from the bottom up. The synthesized attribute values at the terminal nodes of the tree are determined initially (in a compiler, this is the task of the lexical scanner).

The difference between this definition of attribute grammars and the one given in [1] lies in the fact that the terminal symbols can have synthesized but no inherited attributes, and in an attempt to distinguish clearly between an attribute (viewed as a data type) and its occurrences (viewed as variables) within a production.

In the example of Section 2, the attributes *used* and *original* are inherited, and *updated* and *declaration* are synthesized. The semantic functions are simple value transfers, except in production (5) where the value of the occurrence of *updated* is obtained by appending the attribute value "*declaration*" to the "*original.*"

Discussion

a. The order of evaluation. In general, the order of the semantic evaluation can be very complicated, and is determined by the dependency sets of the semantic functions and the form of the derivation tree. The problem is to find an order in which the semantic functions of the production rules in the derivation tree can be executed, such that at the moment when any given function f is executed, the attribute values corresponding to the dependency set of f are already evaluated. If such an order does not exist we have a derivation tree on which the semantics is defined in a circular manner. Knuth [1] has given an algorithm for testing an attribute grammar for the possibility of generating a derivation tree with such a circularity. Unfortunately, in general, the complexity of any deterministic algorithm to solve this problem is such that the execution time is an exponential function of the size of the grammar description (see [18]).

A general algorithm for the evaluation of attributes has been described and implemented by Fang [9]. He uses parallel processes, one for each semantic function in the derivation tree. A process is passivated when it tries to use an attribute occurrence which is not yet evaluated, and it is reactivated when that attribute occurrence has been evaluated by some other process. Such an algorithm tends to be not very efficient.

b. Using attribute grammars for language definition. Attribute grammars have been used [4] to define a programming language in terms of its compilation into a more simple language which is well defined and implemented. Since the semantics is specified in a local manner, i.e. the attribute values of a syntactic symbol within the derivation tree of a program depend only on the values of the immediate neighbors in the tree, this gives rise to a simple and comprehensible semantic specification which is structured according to the syntax of the language.

Compiler writing systems have been constructed which allow a specification of the semantics in terms of semantic attributes and functions [9–12]. The semantic functions, which specify the evaluation of attribute occurrences, are the semantic actions of the generated compiler and are generally expressed in some convenient programming language.

To describe the code generation of a compiler in the framework of attribute grammars, several approaches have been made. Knuth [1] has originally proposed to use a particular synthesized attribute which represents, at each node X of the derivation tree, the translation of the subtree of X. Then the root node contains the translation of the whole program. Another approach is to introduce particular semantic rules for output generation, as has been proposed with translation grammars [13]. In [12], on the other hand, the aspects of syntax, semantic actions, and code generation are all described in the framework of *one* language. Semantic attributes have also been proposed for the specification of code optimization [19].

c. Semantic conditions. Most practical languages are not context-free, although many programming languages allow for a context-free syntax. In many cases the following approach has been used when defining a programming language L: by means of a context-free grammar G_0 the language $L(G_0)$ of *syntactically correct* programs is defined, and then additional restrictions are given that have to be fulfilled by each program of L. This approach can be formalized in different ways [6, 7, 14]. In the framework of attribute grammars, additional restrictions can be introduced by semantic conditions as follows:

We define a language L (not necessarily context-free) in two steps:

(i) We give an attribute grammar with a context-free syntax G_0 such that $L(G_0)$ is an envelope for L, i.e. $L \subset L(G_0)$.

(ii) For each production p of G_0 we give a set of *semantic conditions*. Each condition is a relation between the values of attribute occurrences in the production p which must be satisfied for each occurrence of the production p within the derivation tree of a program of L.

The second step (ii) is the selection of the semantically correct programs, which constitute the language L, out of the syntactically correct programs of $L(G_0)$. One can use a synthesized "error" attribute [4] associated with all nonterminal symbols in order to indicate whether a derivation tree corresponds to a semantically correct program.

Semantic conditions can also be useful for the parsing of programs. For a given programming language, the adopted parsing algorithm may be unable, on certain occasions, to choose among several parsing possibilities on the basis of the syntax alone. Then it is generally not difficult to find some semantic conditions which allow the choice. We note, however, that in this case the semantic attributes involved in the semantic conditions must be evaluated during the same pass from left to right together with the syntax analyisis.

4. Evaluation from Left to Right

Under certain conditions the attributes of a derivation tree can be evaluated in a single pass from left to right. The following algorithm defines what we mean by an evaluation pass from left to right over a derivation tree.

ALGORITHM 1
(Attribute evaluation from left to right)

The algorithm consists of stepping through the derivation tree and evaluating attributes locally according to the semantic functions of the productions in the tree, in the order of a recursive descent from left to right. This is realized by calling a recursive procedure *evaluate-subtree* (*node*) using as parameter the root of the tree.

The action of the procedure *evaluate-subtree* (*node*) can be described in the following terms, where we suppose that at the "*node*" the production $p : X_0 \rightarrow X_1 \cdots X_{n_p}$ applies:

```
for k := 1 to n_p (i.e. for each descendant of "node" from left to right) do
    if X_k is nonterminal then
        evaluate occurrences of inherited attributes at the k-th
            descendant of "node" using the appropriate
            semantic functions of p;
        call evaluate-subtree (k-th descendent of "node");
            (we note that occurrences of synthesized attributes at the kth descendant are thereby evaluated) fi;
evaluate occurrences of synthesized attributes at the "node" using the
    appropriate semantic functions of p.
```

THEOREM 1. *Given an attribute grammar, the attributes of any derivation tree can be evaluated in a single pass from left to right if the dependency sets of the semantic functions of any production* $p{:}X_0 \rightarrow X_1 \ldots X_{n_p}$ *of the grammar satisfy the conditions*

$$D^{(p)}_{(s,0)} \cap S_0 = \varnothing \tag{1}$$

for all synthesized attributes $s \in S(X_0)$ *and*

$$D^{(p)}_{(i,k)} \cap \left\{ S_0 \cup \bigcup_{k'=k}^{n_p} (I_{k'} \cup S_{k'}) \right\} = \varnothing \tag{2}$$

for all $k = 1, \cdots, n_p$ *and* $i \in I(X_k)$, *where* I_k *and* S_k *are respectively the sets of inherited and synthesized attribute occurrences at the k-th symbol of the production.*

One can show that if these conditions are satisfied then all attribute occurrences can be evaluated by following the algorithm given above, since at each point of the algorithm, when an attribute occurrence must be evaluated, the dependency set of the corresponding semantic function contains only attribute occurrences which have been evaluated previously.

It is to be noted that condition (1) represents no restriction of generality, if circularity of attribute definitions is excluded. In particular, circularity can be due to a local circular definition, i.e. a situation where the semantic functions of a *single* production, independently of the surrounding context in the derivation tree, imply a circular definition. In the absence of this kind of local circularity (which can be easily checked, but does not exclude global circularity [1]) it is easy to find, for each production rule, an equivalent set of semantic functions which use only those attribute values which are furnished by the surrounding context of the production within the derivation tree, i.e. the equivalent semantic functions satisfy

$$D^{(p)}_{(a,k)} \subset I_0 \cup \bigcup_{k'=1}^{n_p} S_{k'} \tag{3}$$

for $k = 0$ and $a \in S(X_0)$ as well as $k = 1, \cdots, n_p$ and $a \in I(X_k)$. This implies condition (1) above, but not condition (2). For an attribute grammar which satisfies (3), the above theorem can be stated with an "if and only if":

THEOREM 2. *Given an attribute grammar such that condition* (3) *is satisfied, the attributes of any derivation tree can be evaluated in a single pass from left to right if and only if the condition*

$$D^{(p)}_{(i,k)} \cap \bigcup_{k'=k}^{n_p} S_{k'} = \varnothing$$

is satisfied for all $k = 1, \cdots, n_p$ *and* $i \in I(X_k)$.

In fact, if this condition is not satisfied for the dependency set of some attribute occurrence in some production then the algorithm given above cannot be followed on a derivation tree which contains this production.

Coming back to the example of Section 2, we see that an evaluation from left to right is impossible because of the semantic function for the attribute *used* in production (2). At least two passes from left to right are necessary for this grammar, since the identifier declarations can be placed among executable statements and embedded blocks in any order. If we change the syntax such that all identifier declarations of a block are located at its beginning then we can obtain an attribute grammar which allows a single pass from left to right for the semantic evaluation. Such a grammar is, for instance, the following:

```
⟨program⟩          → ⟨block⟩
⟨block⟩            → ⟨declaration list⟩ ⟨statement list⟩
⟨declaration list⟩ → ⟨declaration list⟩ ⟨identifier declaration⟩
                   → ⟨identifier declaration⟩
⟨statement list⟩   → ⟨statement list⟩ ⟨statement⟩
                   → ⟨statement⟩
⟨statement⟩        → ⟨executable statement⟩
                   → begin ⟨block⟩ end
```

The attributes are the same as in the example of Section 2, except that the attributes *original* and *updated* are only associated with the symbol ⟨declaration list⟩. The semantic functions for the second production are shown in Figure 1(b), the others are similar to those in the example of Section 2.

It is obvious that the construction of the derivation tree (the syntax analysis) for a program and the evaluation of the semantic attributes can be done during one single pass, reading the terminal symbols of the program from left to right, if the language allows a top-down syntax analysis without backup ($LL(k)$) and the semantic functions satisfy conditions (1) and (2). In fact, independently, Lewis et al. [13] have defined an "attributed pushdown machine," and find that there exists a deterministic attributed pushdown machine which performs the syntax analysis and the evaluation of the attributes for any program of a given attribute grammar

(a) if the syntax of the grammar is $LL(k)$ and the semantic functions satisfy conditions (1) and (2), or
(b) if the syntax of the grammar is $LR(k)$ and there are only synthesized attributes, the semantic functions satisfying condition (1).

5. Several Passes from Left to Right

In the preceding section we showed that the semantic attributes within a derivation tree of a program can be evaluated in a single pass from left to right if certain conditions are satisfied. In this section we consider the case that several passes from left to right are necessary to evaluate all attributes. Each pass is executed by following Algorithm 1, given in Section 4. We now describe an algorithm which decides, for a given attribute grammar, whether all occurrences of attributes within the derivation tree of any program can be evaluated by doing a fixed number of passes over the derivation tree from left to right and how many passes are necessary.

There are attribute grammars without circularity such that the attributes of an arbitrary derivation tree cannot be evaluated in a limited number of passes. The grammar of Section 2 is an example. For the derivation tree of Figure 2, the attributes can be evaluated in three passes, but each additional embedding of blocks in the program necessitates an additional pass for the evaluation of attributes.

In order to determine for a given attribute grammar the number of passes which are necessary to evaluate all attributes on any derivation tree, we consider for each nonterminal X the subset $A_m'(X) \subset A(X)$ of those attributes the occurrences of which are evaluated during the mth pass. Each semantic function applied during the mth pass can use the values of all attribute occurrences evaluated during previous passes, and the values of those attribute occurrences evaluated during the same pass subject to conditions similar to (1) and (2). We use the notation

$$NL_{(a,0)}^{(p)} = S_0$$

for $(a, 0) \in S_0$ and

$$NL_{(a,k)}^{(p)} = S_0 \cup \bigcup_{k'=k}^{n_p} (I_{k'} \cup S_{k'})$$

for $k = 1, \cdots, n_p$ and $(a, k) \in I_k$. Then the following algorithm determines for each consecutive pass from left to right which attributes can be evaluated.

ALGORITHM 2

(The idea of this algorithm is to assume initially that, during each pass, it is possible to evaluate all remaining undefined attributes. In the inner loop, it is verified that this is indeed possible.)

Variables used

m: the number of the present pass;

$B'(X)$ *and* $\hat{B}(X)$ *for all* $X \in V$: which signify respectively the subsets of attributes of X which may be evaluated during the present pass and those which have been evaluated in the previous passes;

The algorithm

```
$m := 0$;
$\hat{B}(X) := A(X)$ for all $X \in V_T$;
$\hat{B}(X) := \emptyset$ for all $X \in V_N$;
For each pass do the following:
$m := m + 1$;
$B'(X) := A(X) - \hat{B}(X)$ for all $X \in V$;
Repeat the following:
  Test for all $p \in P$, all $k = 1, \ldots, n_p$,
    for all inherited attributes $a \in B'(X_{pk})$, where $X_{pk}$ is the symbol
    at the k-th position in the production p, and all synthesized attri-
    butes $a \in B'(X_{po})$
  whether the conditions
  (a) $a' \in \hat{B}(X_{pk'}) \cup B'(X_{pk'})$ and
  (b) if $(a', k') \in NL_{(a,k)}^{(p)}$ then $a' \in \hat{B}(X_{pk'})$
  are satisfied for all $(a', k') \in D_{(a,k)}^{(p)}$.
  (We note that the condition (b) means that if the attribute oc-
    currence $(a', k')$ used for the evaluation of $(a, k)$ is "not to the
    left" of the occurrence $(a, k)$, then $a'$ should have been evalu-
    ated during a previous pass.)
  Eliminate the attributes for which these conditions are not satisfied
    from the corresponding $B'$
Until these conditions are satisfied for all remaining attributes.
Obtain $A_m'(X) := B'(X)$ and $\hat{B}(X) := \hat{B}(X) \cup B'(X)$ for all $X \in V_N$.
```

Termination

(a) If no attribute was eliminated during the pass, one obtains

$\hat{B}(X) = A(X)$ for all $X \in V_N$.

This was the last pass and all attributes can be evaluated.

(b) If some attribute was eliminated during the pass, and

$B'(X) = \varnothing$ for all $X \in V_N$,

then the remaining attributes cannot be evaluated in a limited number of left-to-right passes. (This case occurs in particular if the semantic rules of the grammar are circular).

(c) If some attribute was eliminated during the pass, and

$B'(X) \neq \varnothing$ for some $X \in V_N$,

then another pass must be tried.

It is easy to see that the algorithm terminates. The inner loop for a given pass terminates because for each $X \in V_N$ the number of attributes initially in the set $B'(X)$ is finite. Similarly, the total number of passes considered will be finite, since during each pass, except the last one, the number of attributes in the set $\hat{B}(X)$ increases for at least one $X \in V_N$, and we have $\hat{B}(X) \subset A(X)$, the latter being finite sets.

If the attribute grammar satisfies condition (3) (see Section 4) then the number of passes found by the algorithm is the *minimum* number of passes necessary for the evaluation of all attributes on an arbitrary derivation tree. This can be shown by applying Theorem 2 (Section 4), for each given pass, to the set of attributes A_m' that are evaluated. If the condition (3) is not satisfied then the algorithm gives an upper bound for the number of left-to-right passes necessary. We also note that, independently of condition (3), if we consider the attribute evaluation on a particular derivation tree, certain attribute occurrences may be evaluated in an earlier pass than determined by this algorithm.

Applying this algorithm to the example of Section 2 shows that the attribute *used* cannot be evaluated during the first pass because of the semantic function of production (2). Therefore *original* cannot be evaluated (because of production (2)), and neither can *updated* (because of production (6) and (7)). We conclude that there is no attribute such that all of its occurrences in an arbitrary derivation tree can be evaluated during the first pass. The same holds for the following passes.

By making a small change in the semantic functions of production (2) we obtain a grammar which allows a semantic evaluation of any derivation tree in two passes from left to right. The new production (2) is shown in Figure 1(c). During the first pass, the occurrences of the attributes *original* and *updated* are evaluated, and during the second those of *used*. (The occurrences of *declaration* are evaluated initially, because they are associated with a terminal symbol). We note that the changed grammar discussed here is semantically equivalent [3] to the original one; this is not the case for the change discussed in Section 4.

6. Applications

In the last two sections we have described the evaluation of semantic attributes in several passes from left to right over the derivation tree, which is in contrast to Knuth's approach [1] of evaluating the attribute occurrences in any order possible. In this section we discuss how the concept of left-to-right evaluation can be used for the compilation and definition of programming languages.

The results of the previous sections lead us to designing multipass compilers of the following form: apart from the lexical and syntactic analysis of the program, each compiler pass reads over the internal representation of the derivation tree of the program from left to right (see Algorithm 1, Section 4), and evaluates *all* occurrences of certain attributes in the derivation tree.

Given the attribute grammar of the language, Algorithm 2 (Section 5) determines, independent of the program, which attributes are evaluated during any given pass. In particular, Algorithm 2 determines the number of evaluation passes (typically two or three) that are necessary for the programming language, and detects circular attribute definitions. We note that Algorithm 2 is less complex, and more efficient for large grammars than the circularity test of Knuth [1, 18]. In addition, the considered multipass compilers are more efficient than the compilers described by Fang [9] which allow any order of attribute evaluation.

The internal representation of the derivation tree poses some problems if the central memory of the computer is not large enough to contain the whole tree of the program, and complex attributes, such as symbol tables, must be represented. For the compilation of Algol 60, for example, Naur [15] has used a scheme of several passes from left to right and from right to left. His scheme allows writing the internal representation of a ⟨block⟩ and its "*updated*" symbol table [see Figure 1(c)] on auxiliary storage during a pass from right to left. During another pass, from left to right, this symbol table can be read and combined with the inherited symbol table [see Figure 1(c)] before being used in the analysis of the ⟨block⟩, which is subsequently read from auxiliary storage. Using complex attributes, such as symbol tables, also brings up the problem that it is inefficient to use separate copies for the value of such an attribute at different nodes of the derivation tree. Instead pointers could be used. In the case of the grammar of Figures 1(a) and 1(c), which allows an evaluation in two passes, it is sufficient to have only one copy of a symbol table per ⟨block⟩ to represent the "*original*" and "*updated*" attributes during the first pass. During the second pass, a single copy of a "*used*" symbol table is sufficient which is used like a stack throughout the whole program.

Many constructs typically found in programming languages pose no problems for a semantic evaluation in a single pass from left to right. However, some other constructs necessitate several passes. As an example, we have already shown the problems related to the symbol table attribute in a language with Algol block structure. An evaluation of the attributes in two passes could be obtained without changing the original language, whereas an evaluation in one pass was obtained by changing the syntax and semantics of the language in an essential manner. Another example are constructs with forward references, such as **if**, **while** [3], or **goto** *statements*. Realized in a straightforward way, by evaluating the label attributes in one pass and using them for the generation of branch instructions in another, these statements need two passes from left to right. However, simple methods are known [16] to describe the same semantics for an evaluation in a single pass.

It is normally not difficult to satisfy the conditions such that an attribute grammar allows a semantic evaluation in a few passes. In the case that some change of the grammar is necessary, it is often possible to prove that the resultant grammar is semantically equivalent [3] to the original one. Changes of a grammar may also be considered in order to obtain a syntax which allows an efficient parsing of the language. Programming languages are often designed such that syntax analysis and semantic evaluation can be efficiently performed in a single pass from left to right [17].

The idea of using several passes from left to right in the translation of programs is not new. It has been used for a long time in multipass assemblers and compilers. However, the concept of a pass had not been formalized. We have shown here how this concept can be formalized in the framework of attribute grammars. We think that it is useful as a conceptual tool in the design of programming languages. It also can be used for a compiler writing system which is able to generate multipass compilers.

Acknowledgments. I would like to thank Bill Armstrong and Olivier Lecarme for many fruitful discussions.

Received June 1974; revised February 1975

References

1. Knuth, D.E. Semantics of context-free languages. *Math Systems Th. 2* (1968), 127–145. Correction appears in *Math. Systems Th. 5* (1971), 95.
2. Knuth, D.E. Examples of formal semantics, In *Lecture Notes in Mathemetics No. 188*, Springer-Verlag, Berlin 1971.
3. Bochmann, G.V. Semantic equivalence of syntactically related attribute grammars. Publ. No. 148, Département d'Informatique, U. de Montréal, Nov. 1973.
4. Wilner, W.T. Declarative semantic definition. Rep. STAN-CS-233-71, Computer Science Dep., Stanford U., 1971. Also Wilner, W.T. Formal semantic definition using synthesized and inherited attributes. In *Formal Semantics of Programming Languages*, R. Rustin (Ed.), Prentice-Hall, Englewood Cliffs, N.J., 1972.
5. Stearns, R.E., and Lewis, P.M. Property grammars and table machines. *Inform. and Control 14* (1969), 524–549.
6. Culik, K. II, Attributed grammars and languages. Publication No. 3, Département d'Informatique, U. de Montréal, May 1969.
7. Koster, C.H.A. Affix grammars. In *Algol 68 Implementation*, North-Holland Pub. Co., Amsterdam, 1971. See also: Crowe, D. Generating parsers for affix grammars. *Comm. ACM 15* (1972), 728–732.
8. Morris, J.H. Jr. Types are not sets. In Proceedings of the ACM Symp. on Principles of Programming Languages, Boston, 1973, pp. 120–124.
9. Fang, I. FOLDS, a declarative formal language definition system. Rep. STAN-CS-72-329, Computer Science Dep., Stanford U., 1972.
10. Lecarme, O., and Bochmann, G.V. A (truly) usable and portable compiler writing system. In Proc. IFIP Congress 74, North-Holland Pub. Co., Amsterdam, 1974, pp. 218–221.
11. Bouckaert, M., Pirotte, A., and Snelling, M. SOFT: a tool for writing software. Report R212, Laboratoire de Recherche, M.B.L.E., Brussels, Jan. 1973.
12. Bosch, R., Grune, D., and Meertens, L. ALEPH, a language encouraging program hierarchy. In Proc. International Computing Symp. 1973, North-Holland Pub. Co., Amsterdam, 1974, p. 73.
13. Lewis, P.M., Rosenkrantz, D.J., and Stearns, R.E. Attributed translations. To be published in *J. Computer and Systems Sci.*
14. Van Wijngaarden, A., et al. Revised report on the algorithmic language Algol 68. IFIP, 1973.
15. Naur, P. The design of the GIER ALGOL compiler. *BIT, 3* (1963), 124–140 and 145–166, and *Annual Review 4* (1965), 49–85.
16. Gries, D. Compiler Construction for Digital Computers. Wiley, New York, 1971.
17. Wirth, N. The design of a PASCAL compiler. *Software-Practice and Experience 1* (1971), 309–333.
18. Jazayeri, M., Ogden, W.F., and Rounds, W.C. On the Complexity of the circularity test for attribute grammars. In Conf. Record, ACM Symp. on Principles of Programming Languages, Palo Alto, Calif., Jan. 20–22, 1975, pp. 119–129.
19. Neel, D., and Amirchahy, M. Semantic attributes and improvement of generated code. In Proc. ACM Congress 1974, San Diego, Calif., Nov. 1974, pp. 1–10.