International Journal of Computer and Information Sciences, Vol. 8, No. 6, 1979

Semantic Equivalence of Covering Attribute Grammars[1]

Gregor V. Bochmann[2]

Received May 1977; revised June 1979

This paper investigates some methods for proving the equivalence of different language specifications that are given in terms of attribute grammars. Different specifications of the same language may be used for different purposes, such as language definition, program verification, or language implementation. The concept of syntactic coverings is extended to the semantic part of attribute grammars. Given two attribute grammars, the paper discusses several propositions that give sufficient conditions for one attribute grammar to be semantically covered by the other one. These tools are used for a comparison of two attribute grammars that specify syntax and semantics of mixed-type expressions. This example shows a trade-off between the complexity of syntactic and semantic specifications. Another example discussed is the equivalence of different attribute grammars for the translation of the *while*-statement, as used in compilers for top-down and bottom-up syntax analysis.

KEY WORDS: Semantic equivalence; attribute grammars; equivalent semantic specifications; coverings; compiler correctness; formal specification of semantics; semantics of programming languages.

1. INTRODUCTION

Since the definition of ALGOL, it has been very common to use a contextfree grammar to define the syntax of a programming language. On the other hand, there is no standard way to define programming language semantics. There are essentially three approaches: the denotational approach, considering the input-output relation represented by a program; the operational approach,

[1] This work was in part supported by the National Research Council of Canada.

[2] Université de Montréal, Département d'Informatique et de Recherche Opérationnelle, Montréal, Quebec, Canada.

0091-7036/79/1200-0523$03.00/0

828/8/6-7

considering an interpreter that executes programs; and the translational approach, considering the translation of programs into programs of a target language whose semantics is supposed to be known. The last approach seems most natural to the compiler writer.

Often for a given language several different specifications exist as definition of the language. Often only the syntax is formally defined in each specification, but it must be complemented with semantic specifications for obtaining a complete definition of the language. For a programming language, at least the following specifications are important:

1. Specification used during the design of the language.
2. Specification that describes the implementation of the language in terms of a compiler, often used in conjunction with a compiler writing system.
3. Specification for the user of the language, as written in the programming language manual.

These specifications are normally all different, but it is important that they be equivalent. Unfortunately, it is very hard to find a specification method that is suitable for all the purposes.[1]

In this paper we assume that the concept of attribute grammars (2–4) is used as metalanguage for the different language specifications. An attribute grammar consists of a contextfree syntax, semantic attributes that are associated with the nonterminal symbols of the grammar, and evaluation rules that specify the attribute values on the derivation tree for any program. We discuss here some possibilities for proving the equivalence of different attribute grammars.

The proof of the equivalence of different language specifications is certainly not easy, since the question is already undecidable if one considers the syntax alone. However, in the cases already mentioned the different language specifications of the same language are in some sense similar to one another. This simplifies the equivalence proof. If we consider attribute grammars, the similarity lies in the contextfree syntax, the attributes, and the evaluation rules of the grammars. Consequently, Sec. 2 gives some conditions for the equivalence of two grammars which apply when either the syntax or the semantic attributes are identical or very similar in both grammars.

For any useful equivalence proof for two grammars G and G', it seems to be necessary to construct intermediate grammars $G^{(i)}$ such that the equivalence proofs between G and $G^{(1)}$, $G^{(i)}$ and $G^{(i+1)}$ for $1 \leqslant i \leqslant n$, and $G^{(n+1)}$ and G' are straightforward. As a simple example, two different attribute grammars that specify mixed arithmetic expressions are compared in Sec. 3. We mention also the more complex example of attribute grammars

for lambda expressions discussed elsewhere.[9] A similar approach has also been taken by McGowan[10] for the equivalence proof of interpreters.

Since the contextfree syntax of a grammar for a given programming language is closely related to the syntactic analysis of programs by the compiler, and each syntax analysis method used by a compiler works only if the syntax of the language satisfies certain conditions, it is quite usual that different compilers for the same language are based on different contextfree syntaxes. One purpose of this paper is to show that different equivalent contextfree syntaxes, adopted for the design of the language and by different compilers, can be extended to equivalent attribute grammars that specify the syntax and a large part of the semantics of the language. A small example is given in Sec. 4, where the syntax and semantics of the *while*-statement is specified by two different attribute grammars that are suitable for bottom-up and top-down syntax analysis, respectively.

2. EQUIVALENCE OF ATTRIBUTE GRAMMARS

We begin this section by describing some notions that are essential for the definition of semantic equivalence which follows.

2.1. Attribute Grammars

Contextfree grammars with attributes have been proposed for the description of the semantics of programming languages.[2,3] In this paper we employ the notation used in Ref. 4, where a more detailed discussion of attribute grammars can be found.

An attribute grammar G consists of the following parts:

1. *Syntax.* A reduced contextfree grammar $G_0 = (V_t, V_n, S, P)$.

2. *Attributes.* For each nonterminal symbol X in V_n, there is a set $A(X)$ of attributes that is partitioned into two subsets containing the inherited and synthesized attributes, respectively. There are no inherited attributes for the starting symbol $S \in V_n$. Each attribute a takes possible values in a set Val_a.

3. *Evaluation rules.* For each production rule p of the syntax there is a *semantic function* $f^{(p)}_{(s,0)}$ for each synthesized attribute s of the left-side symbol of the rule, and a *semantic function* $f^{(p)}_{(i,k)}$ for each inherited attribute i of the kth symbol of the right side of the rule. Each semantic function specifies how to compute a value for the attribute, given the values of certain other occurrences of attributes of symbols in the same production.

Given an attribute grammar G, with syntax G_0, the semantics of each syntactically correct program $y \in L(G_0)$ is given by the values of the

synthesized attributes of the starting symbol S, i.e., the semantics of a program is an element of the product set $Semantics_G = Val_{a_1} \times Val_{a_2} \times \cdots \times Val_{a_k}$ where $a_1, a_2, \ldots, a_k$ are the synthesized attributes of S. These values are obtained by evaluating all attributes on the derivation tree of the program, using for each production of the tree the semantic functions of the attribute grammar. The absence of circular evaluation rules[2] and the possibility of left-to-right evaluation[4] can be verified. If the syntax of the grammar is nonambiguous, then the semantics specified for each program is unique.

2.2. Structurally Related Grammars

Several notions of structural similarity have been proposed for the comparison of contextfree grammars, lying somewhere between the notion of weak equivalence (generation of the same terminal language) and strong equivalence (generation of the same derivation trees, up to an isomorphism). An isomorphism between two contextfree grammars G_0 and G_0' is a one-to-one correspondance between the symbols and the productions of the two grammars such that for any pair, $p: X_0 \to X_1 X_2 \cdots X_n$ and $p': X_0' \to X_1' X_2' \cdots X_n'$, of corresponding productions in G_0 and G_0' respectively, we have $n = n'$ and the pairs (X_i, X_i') for $i \leqslant 0 \leqslant n$ are pairs of corresponding symbols in the two grammars. For the simplicity of our exposition we do not make any distinction between two contexfree grammars that are isomorphic.

Given two contextfree grammars G_0 and G_0' over the same terminal alphabet, we say that $G_0 = (V_t, V_n, S, P)$ is *finer than* $G_0' = (V_t, V_n', S', P')$ (or G_0' is *coarser than* G_0) if (1) there is a correspondence between the nonterminal symbols V_n and V_n', given by a mapping from V_n' into V_n, and (2) for each production $p: X_0' \to X_1' X_2' \cdots X_n'$ of G_0', there is a derivation $X_0 \overset{*}{\underset{G_0}{\Rightarrow}} X_1 X_2 \cdots X_n$ of corresponding symbols in G_0 using one or more production rules of G_0. This definition is closely related to Reynolds' and Haskell's notion of weak coverings.[5]

We note that, if G_0 is finer than G_0', then the language generated by G_0 contains the one generated by G_0'; i.e., $L(G_0') \subseteq L(G_0)$.

2.3. Semantic Equivalence

As discussed, each attribute grammar G specifies a function $s_G : L(G_0) \to \mathscr{P}(Semantics_G)$ that specifies for each syntactically correct program $y \in L(G_0)$ the set of possible semantics (given for a set Z, we write $\mathscr{P}(Z)$ for the set of all subsets of Z). If G_0 is nonambiguous, then the image of each y is a single element of $Semantics_G$.

Definition 1. Given two attribute grammars G and G' over the same set of terminal symbols, we say that G is *semantically finer* than G' if there exists a mapping ϕ: $Semantics_G \to Semantics_{G'}$, such that $s_{G'}(y) \subseteq \hat{\phi}(s_G(y))$ for all $y \in L(G_0) \cap L(G_0')$, where $\hat{\phi}$ is the natural extension of ϕ to $\mathcal{P}(Semantics_G)$. We say that G and G' are *semantically equivalent* if G is semantically finer than G' and G' is semantically finer than G.

G being semantically finer than G' means that for all programs y that are syntactically valid in both grammars the semantics of y according to G' can be obtained by applying the function ϕ to its semantics according to G. In the case of nonambiguous syntax, this means that the diagram shown in Fig. 1 commutes.

If the starting symbols of the two grammars have the same attributes, the sets $Semantics_G$ and $Semantics_{G'}$ are identical. In this case we say that G is *strongly semantically equivalent* to G' if

$$s_G(y) = s_{G'}(y) \text{ for all } y \in L(G_0) \cap L(G_0').$$

Definition 2. We say that two attribute grammars G and G' are *equivalent* if (1) they are syntactically equivalent, i.e., $L(G_0) = L(G_0')$, and (2) they are semantically equivalent.

To make these definitions useful, we need to find methods for deciding whether one grammar is semantically finer than another one, or whether both grammars are equivalent.

Clearly, this is undecidable in general. But in most practical cases one is interested in verifying the semantic equivalence of two grammars that are likely to be equivalent, and there is normally some kind of similarity between them. In these cases one can possibly use one of the following propositions which specify conditions that are sufficient for proving that a grammar G is semantically finer than another grammar G'.

Proposition 1. Given two attribute grammars G and G' over the same alphabet such that G_0 is (syntactically) finer than G_0' and the attributes for a nonterminal $X' \in V_n'$ are identical to the attributes of the corresponding nonterminal $X \in V_n$, then the following condition is sufficient for G to be semantically finer than G' (with ϕ being the identity mapping), and if G_0 is

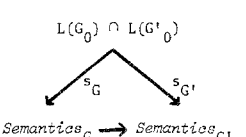

Fig. 1

nonambiguous, this condition implies strong semantic equivalence between G and G'.

Considering any production p': $X_0' \to X_1' X_2' \cdots X_n'$ of G' and any corresponding simulation of it by a derivation $X_0 \overset{*}{\Rightarrow} X_1 X_2 \cdots X_n$ with productions of G, then each semantic function $f'^{(p')}_{(a,k)}$ of G' for evaluating the attribute a of the nonterminal symbol X_k' in terms of other attribute values of the symbols $X_i'(0 \leqslant i, k \leqslant n)$ must be identical to the function obtained by the composition of the semantic functions of the productions used in the derivation $X_0 \overset{*}{\underset{G}{\Rightarrow}} X_1 X_2 \cdots X_n$ according to this derivation.

We consider, for example, the production p': $A' \to B'D'$, which is simulated, as shown in Fig. 2, by the production p_1: $A \to BC$ and p_2: $C \to D$, where the attribute evaluation rules are

$$s_{A'} = f_1'(s_{B'}, s_{D'}) \quad \text{and} \quad i_{D'} = f_2'(s_{B'}) \quad \text{for } p'$$
$$s_A = f_1(s_B, s_C) \quad \text{and} \quad i_C = f_2(s_B) \text{ for } p_1$$

and

$$s_C = f_3(s_D, i_C) \quad \text{and} \quad i_D = f_4(i_C) \text{ for } p_2$$

In this case the above condition has the form

$$f_1'(s_B, s_D) = f_1(s_B, f_3(s_D, f_2(s_B)))$$

and

$$f_2'(s_B) = f_4(f_2(s_B))$$

We note that the identity between the semantic function $f'^{(p')}_{(a,k)}$ of G' and the function obtained by composition of functions of G is only required for those values of the function domain that may actually occur for some occurrence of p' in some derivation tree of a program. It is often possible to derive an assertion, associated with a production or nonterminal of G', specifying a predicate that must be satisfied by the attribute values for all occurrences of the production or nonterminal, respectively, in all derivation trees.[6,7] An example is given in the next section.

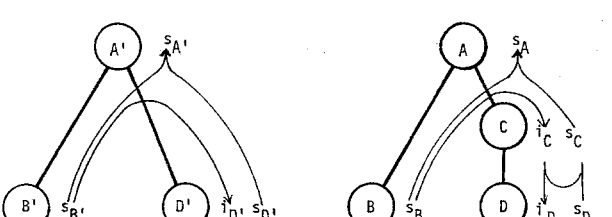

Fig. 2

Proposition 2. We consider two attribute grammars G and G' with the same contextfree syntax $G_0 = (V_t, V_n, S, P)$ but different attributes and evaluation rules. Without restricting the generality the semantic functions of a production may be written in such a form that each function depends only on the inherited attributes of the left-side symbol and the synthesized attributes of the right-side symbols.[4] Therefore we may write the semantic functions of G for a given production p: $X_0 \to X_1 X_2 \cdots X_n$ in vector notation as

$$\bar{s}_{X_0} = \bar{f}_0^{(p)}(\bar{\imath}_{X_0}, \bar{s}_{X_1}, \bar{s}_{X_2}, \ldots, \bar{s}_{X_n})$$

and

$$\bar{\imath}_{X_k} = \bar{f}_k^{(p)}(\bar{\imath}_{X_0}, \bar{s}_{X_1}, \bar{s}_{X_2}, \ldots, \bar{s}_{X_n}) \quad \text{for } 1 \leqslant k \leqslant n$$

Let $\Phi = \{\psi_X \mid X \in V_n\}$ and $\chi = \{\xi_X \mid X \in V_n\}$ be two sets of functions such that

$$\psi_X\colon Val_{s_1} \times \cdots \times Val_{s_m} \to Val_{s_1'} \times Val_{s_2'} \times \cdots \times Val_{s_{m'}'}$$

where $s_1, \ldots, s_m$ and $s_1', \ldots, s_{m'}'$ are the synthesized attributes of X in G and in G', respectively. Similarly ξ_X maps the inherited attribute values of X in G' into the inherited values in G.

Then the conditions

$$\bar{f}_0'^{(p)}(\bar{\imath}'_{X_0}, \psi_{X_1}(\bar{s}_{X_1}), \ldots, \psi_{X_n}(\bar{s}_n)) = \psi_{X_0}(\bar{f}_0^{(p)}(\xi_{X_0}(\bar{\imath}'_{X_0}), \bar{s}_{X_1}, \ldots, \bar{s}_{X_n}))$$

and

$$\xi_{X_k}(\bar{f}_k'^{(p)}(\bar{\imath}'_{X_0}, \psi_{X_1}(\bar{s}_{X_1}), \ldots, \psi_{X_n}(\bar{s}_n))) = \bar{f}_k^{(p)}(\xi_{X_0}(\bar{\imath}'_{X_0}), \bar{s}_{X_1}, \ldots, \bar{s}_{X_n})$$

for $1 \leqslant k \leqslant n$ are sufficient for G being semantically finer than G' with $\phi = \psi_S$.

Proposition 3. We consider two attribute grammars G and G' that are identical except that certain nonterminals of G' have some additional attributes. If the starting symbols S have the same attributes in G and G', and the additional attributes in G' are not used in any evaluation rule for an attribute that is also an attribute in G, then the two grammars are equivalent, independent of the evaluation rules for the additional attributes in G'.

All propositions can be proven by structural induction over the derivation trees of the programs in respect to grammar G'. The proofs of Propositions 1 and 3 are straightforward. The proof of Proposition 2 is complicated by the dependency relations between the inherited and synthesized attributes.

Clearly, circularity[2] must be excluded for both grammars G and G'. If both grammars allow an attribute evaluation in a single pass from left ro right,[4] then the proof of Proposition 2 is relatively simple too.

3. AN EXAMPLE: MIXED EXPRESSIONS

This example concerns mixed arithmetic expressions as they occur in most programming languages. To obtain a simple example that demonstrates the application of the ideas in the preceding section, we consider only the simplest case, namely, constants of *integer* and *real* types, the operation of addition, and the coercion of an *integer* subexpression to *real* type in the case of mixed-type expressions. A program is a simple expression, as for example "2 + 3 + 4.37." The semantics of a program is an indication of the expression type (i.e., *integer* or *real*) and the value of the expression.

We consider two different attribute grammars $G^{(1)}$ and $G^{(2)}$ that define the semantics of such programs. $G^{(1)}$ uses the straightforward syntax [omitting productions for the generation of integer (IP) and real (RP) primaries]

$$P_1^{(1)}: \quad S ::= E \qquad P_4^{(1)}: \quad P ::= IP$$

$$P_2^{(1)}: \quad E ::= E + P \qquad P_5^{(1)}: \quad P ::= RP$$

$$P_3^{(1)}: \quad E ::= P$$

The syntax of $G^{(2)}$ is

$$P_1^{(2)}: \quad S ::= IE \qquad P_5^{(2)}: \quad RE ::= RE + RP$$

$$P_2^{(2)}: \quad S ::= RE \qquad P_6^{(2)}: \quad RE ::= RP$$

$$P_3^{(2)}: \quad IE ::= IE + IP \qquad P_7^{(2)}: \quad RE ::= RE + IP$$

$$P_4^{(2)}: \quad IE ::= IP \qquad P_8^{(2)}: \quad RE ::= IE + RP$$

It is more complex and shows explicitly the coercion of *integer* subexpressions (IE) and primaries to *real* subexpressions (RE).

In the case of grammar $G^{(1)}$ these coercion rules are defined by the semantic function that evaluates the *type* attribute in the production $E ::= E + P$ (see below).

The complete attribute grammar $G^{(1)}$ is given below. We use a notation similar to that in Ref. 1. Each syntactic symbol of a production is followed by the indication of its attributes. For each symbol the attributes are written

in a fixed order, inherited attributes are indicated by a "↓" sign, and synthesized ones by a "↑". (All attributes in the example are synthesized.) Those semantic functions that are simple value transfers are indicated by the use of identical attribute names. Attributes over the same value set *Val*, but generally different values, are distinguished by different indices. More complex evaluation rules are written after the syntactic part of each production.

Grammar $G^{(1)}$ for Mixed Expressions

$P_1^{(1)}$: $S \uparrow\text{type} \uparrow\text{result}$
$::= E \uparrow\text{type} \uparrow\text{result}$

$P_2^{(1)}$: $E \uparrow\text{type}_1 \uparrow\text{result}_1$
$::= E \uparrow\text{type}_2 \uparrow\text{result}_2$ "+" $P \uparrow\text{type}_3 \uparrow\text{result}_3$
with the evaluation rule
$\text{type}_1 = \text{if type}_2 = \text{type}_3$
then type_2 else *real*
and the evaluation rule
$\text{result}_1 = \text{if type}_2 = \textit{integer}$
then if $\text{type}_3 = \textit{integer}$
then $\text{result}_2 +_I \text{result}_3$
else convert $(\text{result}_2) +_R \text{result}_3$
else if $\text{type}_3 = \textit{integer}$
then $\text{result}_2 +_R$ convert (result_3)
else $\text{result}_2 +_R \text{result}_3$

$P_3^{(1)}$: $E \uparrow\text{type} \uparrow\text{result}$
$::= P \uparrow\text{type} \uparrow\text{result}$

$P_4^{(1)}$: $P \uparrow\text{type} \uparrow\text{result}$
$::=$ IP $\uparrow$result
with the evaluation rule
type $=$ *integer*

$P_5^{(1)}$: $P \uparrow\text{type} \uparrow\text{result}$
$::=$ RP $\uparrow$result
with the evaluation rule
type $=$ *real*

We note that the symbols $+_I$ and $+_R$ stand for integer and real addition, respectively. The values of the *type* and *result* attributes of the starting symbol S define the semantics of the program.

The attribute evaluation rules for the grammar $G^{(2)}$ are simpler than those of $G^{(1)}$, since the type of subexpressions is determined by the syntax of $G^{(2)}$. The complete grammar is as follows.

Grammar $G^{(2)}$ for Mixed Expressions

$P_1^{(2)}$: S ↑type ↑result
::= IE ↑result
with the evaluation rule
type = *integer*

$P_2^{(2)}$: S ↑type ↑result
::= RE ↑result
with the evaluation rule
type = *real*

$P_3^{(2)}$: IE ↑$result_1$
::= IE ↑$result_2$ "+" IP ↑$result_3$
with the evaluation rule
$result_1 = result_2 +_I result_3$

$P_4^{(2)}$: IE ↑result
::= IP ↑result

$P_5^{(2)}$: RE ↑$result_1$
::= RE ↑$result_2$ "+" RP ↑$result_3$
with the evaluation rule
$result_1 = result_2 +_R$ result 3

$P_6^{(2)}$: RE ↑result
::= RP ↑result

$P_7^{(2)}$: RE ↑$result_1$
::= RE ↑$result_2$ "+" IP ↑$result_3$
with the evaluation rule
$result_1 = result_2 +_R$ convert ($result_3$)

$P_8^{(2)}$: RE ↑$result_1$
::= IE ↑$result_2$ "+" RP ↑$result_3$
with the evaluation rule
$result_1$ = convert ($result_1$) $+_R$ $result_3$

Comparing the grammars $G^{(1)}$ and $G^{(2)}$, we first note that the syntax of $G^{(1)}$ is finer than the syntax of $G^{(2)}$. This is seen by considering the correspondance of syntactic symbols

$G^{(2)}$	$G^{(1)}$
S	S
IE	E
RE	E
IP	IP
RP	RP

and straightforward simulations, such as production $P_3^{(2)}$: IE $\rightarrow$ IE + IP of $G^{(2)}$ simulated in $G^{(1)}$ by

$$E \overset{P_2^{(1)}}{\Rightarrow} E + P \overset{P_4^{(1)}}{\Rightarrow} E + IP$$

To show the semantic equivalence between $G^{(1)}$ and $G^{(2)}$, we first consider an intermediate grammar $\bar{G}^{(2)}$ that is identical to $G^{(2)}$, except that the expression nonterminals IE and RE have an additional *type* attribute. Since the additional attributes are not used by the semantic functions for the other attributes, which are also present in $G^{(2)}$, Proposition 3 applies and shows that $G^{(2)}$ and $\bar{G}^{(2)}$ are equivalent whatever semantic functions are chosen for the additional attributes.

We note that the attributes of corresponding symbols are the same in $G^{(1)}$ and $\bar{G}^{(2)}$. Therefore Proposition 1 may be used to prove their equivalence. For this purpose we assume the following additional evaluation rules in $\bar{G}^{(2)}$:

1. For $P_4^{(2)}$: type = *integer*.
2. For $P_6^{(2)}$: type = *real*.
3. For $P_3^{(2)}$ and $P_7^{(2)}$: type_1 = if type_2 = *integer*
   then type_2
   else *real*.
4. For $P_3^{(2)}$ and $P_8^{(2)}$: type_1 = *real*.

A straightforward substitution shows that the corresponding simulations in $G^{(1)}$ yield the same semantic functions for both attributes of the left-side symbol of the productions $P_4^{(2)}$ and $P_6^{(2)}$, for the attribute *result* of $P_1^{(2)}$ and $P_2^{(2)}$, and for the attribute *type* of $P_3^{(2)}$, $P_5^{(2)}$, $P_7^{(2)}$, and $P_8^{(2)}$.[3] But for the attribute *result* of the latter productions and the attribute *type* of $P_1^{(2)}$ and $P_2^{(2)}$, the semantic functions are not identical. For example, the function for the attribute *result* of production $P_3^{(2)}$ in $\bar{G}^{(2)}$ is

$$\text{result}_1 = \text{result}_2 +_I \text{result}_3$$

whereas the function obtained by substitution according to the simulation in $G^{(1)}$, given above, is

$$\begin{aligned}\text{result}_1 = {} & \text{if } \text{type}_2 = \textit{integer} \\ & \text{then } \text{result}_2 +_I \text{result}_3 \\ & \text{else } \text{result}_2 +_R \text{convert } (\text{result}_3)\end{aligned}$$

[3] This would not be true for the simpler evaluation rules "type_1 = *integer*" and "type_1 = *real*" for the productions $P_3^{(2)}$ and $P_7^{(2)}$, respectively.

However, these two functions are identical on the domain of values that may actually occur. It is, in fact, easy to show that the assertions

"type = *integer*" and "type = *real*"

hold for all occurrences, within a derivation tree, of the nonterminals IE and RE, respectively. Therefore the attribute $type_2$, in the function above, always has the value *integer*, in which case the two functions are identical. This shows that the production $P_3^{(2)}$ of $\bar{G}^{(2)}$ is correctly simulated by the productions of $G^{(1)}$. Similar considerations apply to the other productions of $\bar{G}^{(2)}$.

We conclude that Proposition 1 may be applied, and the nonambiguity of $G^{(1)}$ implies the strong semantic equivalence between $G^{(1)}$ and $\bar{G}^{(2)}$, and therefore between $G^{(1)}$ and $G^{(2)}$. This example illustrates the fact that certain complexities of languages can be expressed either by the contextfree syntax of the grammar or by its "semantics" in the form of attribute evaluation rules. $G^{(2)}$ has a more complex syntax but simpler semantic evaluation rules than $G^{(1)}$.

4. ANOTHER EXAMPLE: *WHILE*-STATEMENTS

This example illustrates the translation performed by a compiler that translates a high level programming language by generating code for some target language. We consider the *while*-statement and give two possibilities for its translation. We suppose that the remaining rules for the translation are identical.

We consider two attribute grammars for the *while*-statement. The grammar G contains the productions

P_1: ⟨statement⟩ ::= ⟨while clause⟩ ⟨statement⟩

P_2: ⟨while clause⟩ ::= ⟨keyword while⟩ ⟨expression⟩ *do*

P_3: ⟨keyword while⟩ ::= *while*

The grammar G' contains only the production

P': ⟨statement⟩ ::= *while* ⟨expression⟩ *do* ⟨statement⟩

These two grammars are typical for compilers that analyze the programs in a single pass from left to right, constructing the derivation tree of a program from the bottom up or from the top down, respectively.

Both grammars use the following attribute types:

Attribute type	Set of possible values	Significance
valid	{*true*, *false*}	boolean value indicating whether all semantic conditions are satisfied in the subtree of the symbol
type	{*boolean*, etc.}	indicating the data type of an ⟨expression⟩
label	{...}	indicating a possible branch point in the generated object program

In this example we use the notion of action symbols[8] for specifying the translation of a source program of the language. The code generated for a given program is the sequence of action symbols in the derivation tree of the program. Action symbols may have attributes. Each action symbol evaluates its synthesized attributes as functions of its inherited ones. Using the action symbols

locate-label ↑*label*
generate-branch ↓*label*
generate-conditional-branch ↓*label*

we can write the production p' of G', including semantics, as

P': ⟨statement⟩ ↑$valid_1$
::= *while*
locate-label ↑$label_1$
⟨expression⟩ ↑$valid_2$ ↑*type*
generate-conditional-branch ↓$label_2$
⟨statement⟩ ↑$valid_3$
generate-branch ↓$label_1$
locate-label ↑$label_2$
with the evaluation rule
$valid_1 = valid_2 \wedge valid_3 \wedge (type = boolean)$

Similarly, the rules for the *while*-statement in the grammar G are

P_1: ⟨statement⟩ ↑$valid_1$
::= ⟨while clause⟩ ↓$label_2$ ↑$label_1$ ↑$valid_2$
⟨statement⟩ ↑$valid_3$
generate-branch ↓$label_1$
locate-label ↑$label_2$

with the evaluation rule
$valid_1 = valid_2 \wedge valid_3$

P_2: ⟨while clause⟩ $\downarrow label_2 \uparrow label_1 \uparrow valid_1$
::= ⟨keyword while⟩ $\uparrow label_1$
⟨expression⟩ $\uparrow valid_2 \uparrow type$
generate-conditional-branch $\downarrow label_2$
do
with the evaluation rule
$valid_1 = valid_2 \wedge (type = \text{boolean})$

P_3: ⟨keyword while⟩ $\uparrow label$
::= *while*
locate-label $\uparrow label$

Similarly, as in Sec. 3, the equivalence of the grammars G and G' relies on their syntactic equivalence, the fact that G is syntactically finer than G' and nonambiguous, and that the attribute evaluation rules of P' are simulated by the appropriate composition of the evaluation rules of G. Figure 3 shows

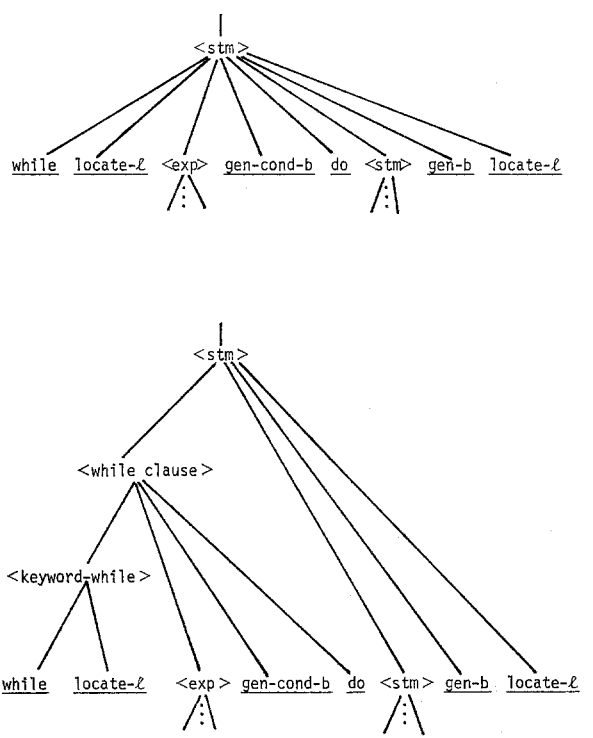

Fig. 3

the production rule P' of G' as a local part of the derivation tree of some program, and the corresponding part of the derivation tree according to the syntax of G, which shows how the production of P' is simulated by the productions P_1, P_2, and P_3 of G.

This example shows a trade-off between grammar forms that are efficient for processing and forms that are easy to understand. The form of grammar G is adapted to efficient bottom-up parsing algorithms, while grammar G' is much simpler to understand.

5. CONCLUSIONS

In Sec. 2 we discussed several propositions that may be used for proving the equivalence of attribute grammars. They use the notion of one grammar being semantically finer than another one, which means that the semantics of the latter can be expressed in terms of the semantics of the former. This situation is similar to the case of syntactic coverings, where the derivation tree of a program in respect to one grammar can be obtained by the derivation tree of the program in respect to the covering grammar. In fact, one of the propositions implies syntactic coverings. Therefore one could call the semantically finer attribute grammar a covering grammar. Under certain conditions the covering grammar is syntactically and semantically equivalent to the covered grammar.

Throughout this paper we have assumed that the metalanguage in which the semantic functions are expressed is well understood. The methods for equivalence proofs, discussed in this paper, assume that methods are known to decide the equivalence between the semantic functions of a given attribute within two different attribute grammars. For example, in the example given in Sec. 3, several semantic functions are specified by a "definition by cases," the meaning of which is used for the equivalence proof.

We have not addressed, in this paper, the problem of deciding the equivalence of different semantic functions, because we feel that the methods discussed in the paper are related to the attribute-passing mechanism of attribute grammars, and independent of any particular approach to the problem of deciding the equivalence of semantic functions. In practice, the latter problem is closely related to the (quite complex) problem of proving properties of programs. In fact, most compiler writing systems for attribute grammars use a particular programming language for the specification of the semantic functions.

The examples given in Sec. 3 and 4 show that certain trade-offs exist between different equivalent language definitions. Among the virtues of a language definition we mention

1. Simplicity of contextfree syntax (ease of understanding).
2. Simplicity of the semantics (ease of understanding).
3. Efficiency of implementation:
   (a) Efficiency of syntax analysis of the compiler.
   (b) Efficiency of semantic analysis and code generation of the compiler.
   (c) Efficiency of run-time organization.

The example of mixed expressions in Sec. 3 shows a trade-off between points 1 and 2; the example of the *while*-statements in Sec. 4, between points 1, 2, and 3a. In a different context Hoare[11] discusses several aspects of programming languages that are important in the practice of programming language design. It is difficult to combine all advantages into one language definition. It seems that certain trade-offs between the different aspects of the definition cannot be avoided.

Since one of the applications of the methods discussed in this paper is proving the equivalence between the definition of a programming language, as written down during its design, and its implementation by a compiler, we hope that these methods, together with other necessary tools, will actually be used in the correctness proof of compilers.

ACKNOWLEDGMENT

I am grateful to Bill Armstrong for many fruitful discussions on the subject of this paper.

REFERENCES

1. M. Marcotty, H. F. Ledgard, and G. V. Bochmann, "A sampler of formal definitions," *ACM Comput. Surv.* **8**(2):191–276 (June 1976).
2. D. E. Knuth, "Semantics of context-free languages," *Math. Syst. Theory* **2**:127–145 (1968); **5**:95 (1971).
3. C. H. A. Koster, "Affix Grammars," in J. E. L. Peck, ed., *Algol 68 Implementation* (North Holland, Amsterdam, 1971), pp. 95–109.
4. G. V. Bochmann, "Semantic evaluation from left to right," *Commun. ACM* **19**(2): 55–62 (February 1976).
5. J. C. Reynolds and R. Haskell, "Grammatical Coverings," unpublished manuscript, 1970. See also A. V. Aho and J. P. Ullman, *The Theory of Parsing, Translation and Compiling*, Vol. I, Sec. 3.4.5 (Prentice Hall, Englewood Cliffs, New Jersey, 1972).
6. G. Godbout, "Définition d'un langage intermédiaire pour un système d'écriture de compilateurs," Master's thesis, Département d'Informatique et de Recherche Opérationnelle, Université de Montréal (1976).

7. C. Pair, M. Amirchahy, and D. Neel, *Correstness Proofs of Text-Processing Descriptions by Attributes* (IRIA-Laboria, France, 1976).
8. P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns, "Attributed Translations," *Proceedings of the ACM Symposium on Theory of Computing*, Austin (May 1973), pp. 160–171.
9. G. V. Bochmann, "Semantic Equivalence of Covering Attribute Grammars," Publication #218, Département d'I.R.O., Université de Montréal (1975).
10. C. L. McGowan, "An Inductive Proof Technique for Interpreter Equivalence," in R. Austin, ed., *Formal Semantics of Programming Languages* (Prentice Hall, Englewood Cliffs, New Jersey, 1972), pp. 139–147.
11. C. A. R. Hoare, "Hints on Programming Language Design," invited address at SIGACT/SIGPLAN Symposium on Principles of Programming Languages, Boston (October 1973), and Stanford CS Report.