Extended Attribute Grammars

David A. Watt
Computing Science Department, University of Glasgow, Glasgow G12 8QQ, UK

Ole Lehmann Madsen
Computer Science Department, Aarhus University, Ny Munkegade, DK 8000 Aarhus C, Denmark

Two new formalisms are introduced: extended attribute grammars, which are capable of defining completely the syntax of programming languages, and extended attributed translation grammars, which are additionally capable of defining their semantics by translation. These grammars are concise and readable, and their suitability for language definition is demonstrated by a realistic example. The suitability of a large class of these grammars for compiler construction is also established, by borrowing the techniques already developed for attributable grammars and affix grammars.

1. INTRODUCTION

This paper is concerned with the formalization of the syntax and semantics of programming languages. The primary aim of formalization are preciseness, completeness and unambiguity of language definition. Given these basic properties, the value of a formalism depends critically on its clarity, without which its use will be restricted to a tight circle of theologians. Another important property of a formalism is its suitability for automatic compiler construction, since this greatly facilitates the correct implementation of the defined language.

Experience with context-free grammars (CFGs) illustrates our points well. Although not capable of defining completely the syntax of programming languages (which are context-sensitive), CFGs have all the other desirable properties, and their undoubted success has been due both to their comprehensibility to ordinary programmers and to their value as a tool for compiler writers. Indeed, it is likely that any more powerful formalism, if it is to match the success of CFGs, will have to be a clean extension of CFGs which retains all their advantages.

We firmly believe in the advantages of formalization of a programming language at its design stage. Even such a clear and well-designed language as Pascal contained hidden semantic irregularities which were revealed only by formalization of its semantics. Similarly, certain ill-defined features of the context-sensitive syntax of Pascal (such as the exact scope of each identifier) are thrown into sharp relief by an attempt at formalization. It is well known that issues not resolved at the design stage of a programming language tend to become resolved de facto by its first implementations, not necessarily in accordance with the intentions of its designers.

A survey article has assessed four well-known formalisms, van Wijngaarden grammars, production systems, Vienna definition language and attribute grammars, comparing them primarily for completeness and clarity. None of these formalisms is fully satisfactory, even from this limited viewpoint. The first three formalisms tend to produce language definitions which are, in our opinion, difficult to read. Attribute grammars are easier to understand because of their explicit attribute structure and distinction between ‘inherited’ and ‘synthesized’ attributes. These same properties make attribute grammars the only one of these formalisms which is suitable for automatic compiler construction, a important application which was not considered in the survey article.

In this paper we introduce a new formalism, the extended attribute grammars (EAGs), which we believe will compare favourably with these well-known formalisms from every point of view. EAGs are based on attribute grammars and affix grammars, and retain the more desirable properties of these formalisms, but are designed to be more elegant, readable and generative in nature. They represent a refinement of earlier work by the authors.

Section 2 of this paper is an informal introduction to EAGs via attribute grammars and affix grammars, and Section 3 is a more formal definition of EAGs. In Section 4 we discuss the possibilities of using EAGs to specify the semantics as well as the syntax of programming languages, and we introduce an enhanced formalism, the extended attributed translation grammars (EATGs), which are designed to do so by translation into some target language. Section 5 demonstrates the suitability of a large class of EAGs for automatic compiler construction, and contains a brief description of a compiler writing system based on EATGs which has been implemented at Aarhus.

In the appendices we give a complete definition by an EAG of the syntax of a small but realistic programming language, and by an EATG of its translation into an intermediate language. These examples should allow readers to judge for themselves the suitability of these formalisms for language definition.

2. ATTRIBUTE GRAMMARS AND EXTENDED ATTRIBUTE GRAMMARS

In this section we briefly describe attribute grammars and affix grammars, and introduce extended attribute grammars. We use a notation which is based on BNF.
The empty sequence is denoted by \( \langle \text{empty} \rangle \). Terminal symbols without attributes are enclosed in quotes.

Assignments in an ALGOL68-like language are used as a running example throughout this section. The LHS of each assignment must be an identifier of mode ref(MODE), where MODE is the mode of the RHS; each identifier must be declared (elsewhere), and its mode is determined by its declaration. We shall use the term 'environment' for the set of declared identifiers together with their modes, and we shall view this environment as a partial map from names to modes. We shall assume the following context-free syntax:

(1) \( \langle \text{assignment} \rangle :: = \langle \text{identifier} \rangle \text{":=" } \langle \text{expression} \rangle \)
(2) \( \langle \text{identifier} \rangle :: = \langle \text{name} \rangle \)

2.1 Attribute grammars and affix grammars

Attribute grammars were devised by Knuth, and affix grammars independently by Koster. The two formalisms are essentially equivalent, and we shall attempt to abstract their common properties by a unified notation. We use the abbreviation AG to mean either attribute grammar or affix grammar.

The basic idea of AGs is to associate, with each symbol of a CFG, a fixed number of attributes, with fixed domains. Different instances of the same symbol in a syntax tree may have different attribute values, and the attributes can be used to convey information obtained from other parts of the tree. A distinction is made between synthesized and inherited attributes. Consider a symbol \( X \) and a phrase \( p \) derived from \( X \). Each inherited attribute of \( X \) is supposed to convey information about the context of \( p \), and each synthesized attribute of \( X \) is supposed to convey information about \( p \) itself. We shall prefix inherited attributes by downward arrows (\( \downarrow \)) and synthesized attributes by upward arrows (\( \uparrow \)).

In our example, each of the non-terminals \( \langle \text{assignment} \rangle \), \( \langle \text{identifier} \rangle \) and \( \langle \text{expression} \rangle \) will have an inherited attribute representing its 'environment' (inherited since it represents information about the context). Each of \( \langle \text{identifier} \rangle \) and \( \langle \text{expression} \rangle \) will also have a synthesized attribute representing its mode. The symbol \( \langle \text{name} \rangle \) will have a single synthesized attribute, its spelling.

The attributes can be used to specify context-sensitive constraints on a language with a context-free phrase structure. Each AG rule is basically a context-free production rule augmented by:

(a) evaluation rules, specifying the evaluation of certain attributes in terms of others, and
(b) constraints, or predicates which must be satisfied by the attributes in each application of this rule.

In our example, assignments could be specified by the following rule:

\[
\langle \text{assignment} \rangle \text{ENV} :: =
\begin{align*}
(1) & \langle \text{identifier} \rangle \text{ENV1} \text{MODE1} \text{":="} \\
& \langle \text{expression} \rangle \text{ENV2} \text{MODE2} \\
& \text{evaluate ENV1 } \rightarrow \text{ENV} \\
& \text{evaluate ENV2 } \rightarrow \text{ENV} \\
& \text{MODE1 } = \text{ref(MODE2)}
\end{align*}
\]

Where 'introduces a constraint, and 'evaluate' introduces an evaluation rule. Here we have used some attribute variables, ENV, ENV1, ENV2, MODE1 and MODE2, to stand for the various attribute occurrences in this rule.

The evaluation rules specify that the environment attributes of both \( \langle \text{identifier} \rangle \) and \( \langle \text{expression} \rangle \) are to be made equal to the environment attribute of \( \langle \text{assignment} \rangle \). The constraint specifies the relation which must hold between the mode attributes of \( \langle \text{identifier} \rangle \) and \( \langle \text{expression} \rangle \).

An 'identifier' is a name for which a mode is defined in the environment. We could specify this by the following rule:

\[
\langle \text{identifier} \rangle \text{ENV } \text{MODE} :: =
\begin{align*}
(2) & \langle \text{name} \rangle \text{NAME} \\
& \text{evaluate MODE } = \text{ENV[NAME]}
\end{align*}
\]

Here we compute the mode attribute of \( \langle \text{identifier} \rangle \) by applying the map ENV to NAME, the attribute of \( \langle \text{name} \rangle \), where ENV is the environment attribute of \( \langle \text{identifier} \rangle \). There is an implicit constraint here, that the map ENV is in fact defined at the point NAME.

Inherited attribute-positions on the left-side and synthesized attribute-positions on the right-side of a rule are called defining positions. Synthesized attribute-positions on the left-side and inherited attribute-positions on the right-side of a rule are called applied positions. This classification is illustrated below:

\[
\langle X \rangle \uparrow \ldots \uparrow \downarrow \ldots \uparrow \downarrow \ldots \uparrow \downarrow \ldots \uparrow \downarrow \ldots : = \langle X_1 \rangle \uparrow \ldots \uparrow \downarrow \ldots \uparrow \downarrow \ldots \uparrow \downarrow \ldots \uparrow \downarrow \ldots \uparrow \downarrow \ldots
\]

In general, there must be exactly one attribute variable for each defining position in a rule. The evaluation rules specify how to compute all attributes in applied positions from those in defining positions. The constraints relate some of the attributes in defining positions. (This definition is actually more restrictive than that of Ref. 7, in which the evaluation rules may use attributes from any positions. As Ref. 9 points out, however, the restriction effectively excludes only grammars containing circularities.)

In practice, many evaluation rules turn out to be simple copies; we can eliminate these by allowing any variable which occupies a defining position also to occupy any number of applied positions, and for each such position a simple copy is implied. This allows rule (1) to be simplified as follows:

\[
\langle \text{assignment} \rangle \text{ENV} :: =
\begin{align*}
(1) & \langle \text{identifier} \rangle \text{ENV1 MODE1} \text{":="} \\
& \langle \text{expression} \rangle \text{ENV2 MODE2} \\
& \text{where MODE1 } = \text{ref(MODE2)}
\end{align*}
\]

The choice of \( \downarrow \) and \( \uparrow \) to distinguish inherited and synthesized attributes is motivated by the tendency of inherited attributes to move downwards, and synthesized attributes to move upwards, in a syntax tree. To illustrate this, Fig. 1 shows a fragment of a syntax tree, based on our example.

![Figure 1. Fragment of an attributed syntax tree. The input string is: \( x := \langle \text{expression} \rangle \). E stands for the attribute \( x \rightarrow \text{ref(int)} \), \( y \rightarrow \text{bool} \). Broken arrows leading to each attribute indicate which other attributes it depends upon.](https://academic.oup.com/comjnl/article/26/2/142/310019)
Attribute grammars have been used to define the context-sensitive syntax of several programming languages. Relative to van Wijngaarden grammars,\footnote{Language definitions by AGs are easy to understand, because of the explicit attribute structure and the distinction between inherited and synthesized attributes. It is quite easy to detect the underlying context-free syntax, although this does tend to be obscured by a profusion of evaluation rules and constraints. Another disadvantage of AGs is that they are not generative grammars. AGs are well suited to compiler construction, and have been exploited in many compiler writing systems.\footnote{We shall return to this topic in Section 5.}} for example, language definitions by AGs are easy to understand, because of the explicit attribute structure and the distinction between inherited and synthesized attributes. It is quite easy to detect the underlying context-free syntax, although this does tend to be obscured by a profusion of evaluation rules and constraints. Another disadvantage of AGs is that they are not generative grammars.

2.2 Extended attribute grammars

EAGs are intended to preserve all the desirable properties of AGs, but at the same time to be more concise and readable. Like van Wijngaarden grammars,\footnote{EAGs are generative grammars.} EAGs are generative grammars.

A straightforward notational improvement on AGs is to allow attribute expressions, rather than just attribute variables, in applied positions; for each such attribute expression an evaluation rule is implied. For example, rule (2) in our example could be expressed as follows:

\[
\langle \text{identifier} \uparrow \text{ENV} \uparrow \text{ENV}[\text{NAME}] \rangle ::=
\text{(2) } \langle \text{name} \uparrow \text{NAME} \rangle
\]

This relaxation makes explicit evaluation rules unnecessary.

In EAGs we go much further, however, and allow any attribute position, applied or defining, to be occupied by an attribute expression. Moreover, we withdraw the restriction that each attribute variable must occur in only one defining position in a rule. These relaxations allow all relationships among the attributes in each rule to be expressed implicitly, so that explicit evaluation rules and constraints become unnecessary. The attribute variables become somewhat akin to the ‘metanotions’ of a van Wijngaarden grammar.

Our example could be expressed in an EAG as follows:

\[
\langle \text{assignment} \uparrow \text{ENV} \rangle ::=
\text{(1) } \langle \text{identifier} \uparrow \text{ENV} \uparrow \text{ref}(\text{MODE}) \rangle \\
\langle \text{expression} \uparrow \text{ENV} \uparrow \text{MODE} \rangle
\text{(2) } \langle \text{identifier} \uparrow \text{ENV} \uparrow \text{ENV}[\text{NAME}] \rangle ::=
\text{[expression [x -> reft(int), y -> bool] ref(int)] [name -> x]}
\]

In rule (1) we have specified the relation which must hold between the second attribute, \text{MODE}, of \text{expression}, and the second attribute of \text{identifier} simply by writing \text{ref(MODE)} in the latter position. Similarly, in rule (2) we have specified that the second attribute of \text{identifier} is obtained by applying \text{ENV} to \text{NAME} simply by writing \text{ENV[NAME]} in the appropriate position.

It may be seen that the EAG rules are rather more concise than the corresponding AG rules, and the underlying context-free syntax is consequently more visible.

Context-sensitive errors are treated by EAGs in the same implicit manner as context-free syntax errors are by CFGs. A CFG can generate only (context-free) error-free strings. Similarly, an EAG can generate only (context-sensitive) error-free strings.

Each EAG rule acts as a generator for a (possibly infinite) set of context-free production rules, using a systematic substitution mechanism similar to that of van Wijngaarden grammars. In detail, this works as follows. To generate a production rule, we must systematically substitute some suitable attribute for each attribute variable occurring in the rule; then we must evaluate all the attribute expressions.

For example, after systematically substituting \([x \rightarrow \text{ref(int)}, y \rightarrow \text{bool}]\) for \text{ENV} and \(x\) for \text{NAME} in rule (2), and evaluating \text{ENV[NAME]}, we get the production rule

\[
\langle \text{identifier} \uparrow [x \rightarrow \text{ref(int)}, y \rightarrow \text{bool}] \uparrow \text{ref(int)} \rangle ::=
\langle \text{name} \uparrow x \rangle
\]

This production rule may be applied at some node of a syntax tree (just as in Fig. 1).

If, instead, we try to substitute \(z\) for \text{NAME}, we find that the value of \text{ENV[NAME]} is not defined; therefore no production rule can be generated.

The rest of Fig. 1 can be filled in by substituting \([x \rightarrow \text{ref(int)}, y \rightarrow \text{bool}]\) for \text{ENV} and \text{int} for \text{MODE} in rule (1), giving the production rule

\[
\langle \text{assignment} \uparrow [x \rightarrow \text{ref(int)}, y \rightarrow \text{bool}] \rangle ::=
\langle \text{identifier} \uparrow [x \rightarrow \text{ref(int)}, y \rightarrow \text{bool}] \uparrow \text{ref(int)} \rangle \\
\langle \text{expression} \uparrow [x \rightarrow \text{ref(int)}, y \rightarrow \text{bool}] \rangle \\
\langle \text{name} \uparrow [x \rightarrow \text{ref(int)}, y \rightarrow \text{bool}] \rangle
\]

The systematic substitution rule makes it impossible to generate from rule (1) a production rule in which the mode attributes of \text{identifier} and \text{expression} are, for instance, \text{ref(int)} and \text{bool}, respectively.

3. FORMAL DEFINITION OF EXTENDED ATTRIBUTE GRAMMARS

An extended attribute grammar is a 5-tuple

\[
G = (D, V, Z, B, R)
\]

whose elements are defined in the following paragraphs. \(D = (D_1, D_2, \ldots, f_1, f_2, \ldots)\) is an algebraic structure with domains \(D_1, D_2, \ldots, \) and (partial) functions \(f_1, f_2, \ldots\) operating on Cartesian products of these domains. Each object in one of these domains is called an attribute.

\(V\) is the vocabulary of \(G\), a finite set of symbols which is partitioned into the non-terminal vocabulary \(V_N\) and the terminal vocabulary \(V_T\). Associated with each symbol in \(V\) is a fixed number of attribute-positions. Each attribute-position has a fixed domain chosen from \(D\), and is classified as either inherited or synthesized.

\(Z\), a member of \(V_N\), is the distinguished non-terminal of \(G\).

We shall assume, without loss of generality, that \(Z\) has no attribute-positions, and that no terminal symbol has any inherited attribute-positions.

\(B\) is a finite collection of attribute variables (or simply variables). Each variable has a fixed domain chosen from \(D\).

An attribute expression is one of the following:

(a) a constant attribute, or
(b) an attribute variable, or
(c) a function application \(f(e_1, \ldots, e_m)\), where \(e_1, \ldots, e_m\) are attribute expressions and \(f\) is an appropriate (partial) function chosen from \(D\).

In practice, when writing down attribute expressions
we use not only functional notation but also other
conventional notations such as infix operators.

Let \( v \) be any symbol in \( V \), and let \( v \) have \( p \) attribute-
positions whose domains are \( D_1, \ldots, D_p \), respectively. If
\( a_1, \ldots, a_p \) are attributes in the domains \( D_1, \ldots, D_p \),
respectively, then
\[
\langle v + a_1 \cdots + a_p \rangle
\]
is an attributed symbol. In particular, it is an attributed
non-terminal (terminal) if \( v \) is a non-terminal (terminal).
Each \( + \) stands for either \( \downarrow \) or \( \uparrow \), prefixing an inherited or
synthesized attribute-position as the case may be.

If \( e_1, \ldots, e_p \) are attribute expressions whose ranges are
included in \( D_1, \ldots, D_p \), respectively, then
\[
\langle v + e_1 \cdots + e_p \rangle
\]
is an attributed symbol form.

\( R \) is a finite set of production rule forms (or simply
rules), each of the form:
\[
F : \leftarrow F_1 \ldots F_m
\]
where \( m \geq 0 \), and \( F, F_1, \ldots, F_m \) are attributed symbol
forms, \( F \) being non-terminal.

The language generated by \( G \) is defined as follows.

Let \( F : \leftarrow F_1 \ldots F_m \) be a rule. Take a variable \( x \) which
occurs in this rule, select any attribute \( a \) in the domain of
\( x \), and systematically substitute \( a \) for \( x \) throughout the
rule. Repeat such substitutions until no variables remain,
then evaluate all the attribute expressions. *Provided all the
attribute expressions have defined values*, this yields a
production rule, which will be of the form:
\[
A : \leftarrow A_1 \ldots A_m
\]
where \( m \geq 0 \), and \( A, A_1, \ldots, A_m \) are attributed symbols,
\( A \) being an attributed non-terminal.

A direct production of an attributed non-terminal \( A \) is
a sequence \( A_1, \ldots, A_m \) of attributed symbols such that \( A
: \leftarrow A_1 \ldots A_m \) is a production rule.

A production of \( A \) is either:

(a) a direct production of \( A \), or
(b) the sequence of attributed symbols obtained by
replacing, in some production of \( A \), some attributed
non-terminal \( A' \) by a direct production of \( A' \).

A terminal production of \( A \) is a production of \( A \) which
consists entirely of (attributed) terminals.

A sentence of \( G \) is a terminal production of the
distinguished non-terminal \( Z \). (Recall that \( Z \) has no
attributes.)

The language generated by \( G \) is the set of all sentences
of \( G \).

Observe that the distinction between inherited and
synthesized attributes makes no difference to the language
generated by the EAG. Nevertheless, we believe that
this distinction makes a language definition easier to
understand. It is also essential to make EAGs suitable
for automatic compiler construction.

Complete examples of EAGs may be found in
Appendix A and in Ref. 3.

4. EXTENDED ATTRIBUTED TRANSLATION
GRAMMARS

We have seen that a CFG can be enhanced with
attributes to define context-sensitive syntax. In a similar
manner, a syntax-directed translation schema (SDTS)\(^{19}\)
can be enhanced with attributes and thus express context-
sensitivities of both an input grammar and an output
grammar. The attributed translation grammars of Ref. 20
are in fact an enhancement of simple SDTSs with
attributes, in the style of ordinary AGs.

By analogy with the previous sections, it is straightforward
to generalize SDTSs in the style of EAGs. The resulting
extended attributed translation grammars (EATGs) are a powerful tool for specifying the analysis
phase of compilers. A major example of this can be found in
Ref. 21.

An EATG is an EAG where the terminal vocabulary
is partitioned into two disjoint sets, the (attributed) input
symbols and the (attributed) output symbols. We shall assume
that no input symbol has any inherited attribute-
positions and that no output symbol has any synthesized
attribute-positions. Like an STDS rule, an EATG rule
consists of an input rule and an output rule. The input and
output rules are ordinary EAG rules. The input rules
consist of input symbols and non-terminals; the output rules
consist of output symbols and non-terminals. The attributes are partitioned into two disjoint sets, one for
the input rules and one for the output rules. The two
attribute sets express context-sensitivities of the input language and the output language, respectively.

In general, we allow each output rule to make use of
any attribute variables from the corresponding input
rule, but not vice versa. Notwithstanding their separation,
the input rule and corresponding output rule are taken together when applying the EAG systematic
substitution rule.

It is straightforward to generalize the formal definition of
EAGs in Section 3 to EATGs and we shall not do so here.
The main advantage of EATGs relative to EAGs is that
EATGs are better suited for expressing modularity in
language definitions.

To demonstrate the advantages of EATGs we show how
easy 9.19 of Ref. 19 may be written using an
EATG. The example is code generation for arithmetic
expressions to a machine with two fast registers, A and
B. The terminals of the output EAG correspond to
instructions of this machine. Most of these symbols have
an inherited register-valued attribute \((a:b)\) and an
inherited attribute representing a storage address of the
machine. The multiply instruction, \( MPY \), takes one
operand from B and the other operand from store, and
delivers its result in A. The other instructions should be
obvious. The corresponding output terminals are:
\[
\langle LOAD \downarrow Register \downarrow Integer \rangle
\]
\[
\langle ADD \uparrow Register \downarrow Integer \rangle
\]
\[
\langle STORE \downarrow Register \downarrow Integer \rangle
\]
\[
\langle MPY \downarrow Integer \rangle
\]
\(ATOB\) ('move contents of A to B')

The non-terminals of the output EAG have two attributes
each: an inherited register-valued attribute which specifies
where the corresponding subexpression should be
evaluated, and a synthesized integer attribute representing
the height of the corresponding syntax subtree. The
latter attribute is used to keep track of safe temporary
locations. More details about the example and the code-
generation strategy adopted may be found in Ref. 19.

We have extended the input EAG with a map-valued
attribute which for each identifier gives its address in

THE COMPUTER JOURNAL, VOL. 26, NO. 2, 1983 145
store. We omit rules for defining this attribute since this
is fully demonstrated in Appendix A. We suppose that
the non-terminal \langle evaluation \rangle is part of a larger
grammar.

We have taken the liberty of adding a non-terminal to
the output EAG which is not present in the input EAG.
This should cause no conceptual difficulty.

Input rules

(1) \langle evaluation \downarrow ENV \rangle \::= \langle expr \downarrow ENV \rangle
(2) \langle expr \downarrow ENV \rangle \::= \langle expr \downarrow ENV \rangle + " + "
\langle term \downarrow ENV \rangle
(3) \langle term \downarrow ENV \rangle \::= \langle term \downarrow ENV \rangle \cdot " \cdot "
\langle factor \downarrow ENV \rangle
(4) \langle factor \downarrow ENV \rangle \::= \langle factor \downarrow ENV \rangle " "
\langle name \uparrow NAME \rangle

Output rules

(1) \langle evaluation \rangle \::= \langle expr \uparrow H \rangle
(2) \langle expr \uparrow REG \rangle \uparrow \langle max(H1, H2) + 1 \rangle \::= 
\langle term \uparrow H1 \rangle \langle STORE \downarrow a \downarrow H2 \rangle
\langle expr \downarrow REG \rangle \langle H2 \rangle
\langle ADD \downarrow REG \rangle \langle H2 \rangle
(3) \langle expr \downarrow REG \rangle \uparrow \langle max(H1, H2) + 1 \rangle \::= 
\langle factor \uparrow H1 \rangle \langle STORE \downarrow a \downarrow H2 \rangle
\langle term \downarrow H2 \rangle \langle MPY \downarrow H2 \rangle
\langle move \rangle \langle REG \rangle
(4) \langle term \downarrow REG \rangle \langle H1 \rangle \::= \langle factor \downarrow REG \rangle \langle H1 \rangle
(5) \langle factor \downarrow REG \rangle \langle H1 \rangle \::= \langle expr \downarrow REG \rangle \langle H1 \rangle
(6) \langle factor \downarrow REG \rangle \langle H1 \rangle \::= \langle expr \downarrow REG \rangle \langle H1 \rangle
(7) \langle move \rangle \langle a \rangle \::= \langle empty \rangle
\langle move \rangle \langle b \rangle \::= \langle ATOB \rangle

Figure 2 shows attributed syntax trees for an example
translation.

The generalization of SDTSs to EATGs in the style of
EAGs is, as mentioned, straightforward. Our reason for
treating EATGs in this paper is to demonstrate their
practical use when defining semantics. (In this paper we
take the liberty of using 'semantics' in the narrow sense
of defining a translation.) The use of EATGs allows a
high degree of modularity in defining semantics. The
input EAG may be used to define the (context-sensitive)
syntax of a language, and the output EAG its semantics.
This makes it possible to separate the two parts and to
have a clean interface consisting of corresponding rules
interconnected with attributes. Furthermore, it is possible
to have more than one output EAG corresponding to
the same input EAG, and in this way to define different
semantics. Examples of different semantics are:

(a) Defining a translation into an intermediate language
suitable for code generation. In Appendix B, the
EAG of Appendix A is enhanced to an EATG
defining such a translation.
(b) Defining a translation into code for a hypothetical
machine (perhaps a real machine if it has a simple
structure) intended for interpretation.

(c) Defining a translation into some lambda-notation
that may be 'executed' by a lambda reducer. An
example of this is the language LAMB of SIS, which is
a compiler generator based upon denotational
semantics; SIS also provides a reducer for
LAMB.

(d) Defining a verification generator by means of an
output EAG which has predicates as attributes and
generates a series of verification conditions.

5. IMPLEMENTATION ISSUES

5.1 Parsing and attribute evaluation with AGs

Some AGs contain circularities, i.e. situations in which a
set of attributes (not necessarily all occurring in one rule)
depend upon one another circularly. Circularity implies
that there is no order in which all the attributes can be
evaluated. Fortunately, circularities can be detected
automatically from the grammar.

A decade of research has produced a variety of
attribute evaluators for non-circular AGs. These include
one-pass evaluators, multi-pass left-to-right evaluators, multi-pass alternating evaluators, and multi-sweep evaluators. In all these cases the order of evaluation is fixed by the constructor, independently of any particular program. By contrast, there are some systems (such as DELTA and NEATS) which choose an evaluation order dependent on the particular program. These are general enough to accept any non-circular AG.

5.2 Extension to extended attribute grammars

All the attribute evaluators mentioned in the previous section can be used for EAGs as well. The simplest way to establish this is to show how, and in what circumstances, an EAG can be converted automatically into an equivalent AG.

The following examples, all taken from Appendix A, illustrate the necessary transformations.

Example 1

\[
\langle \text{identifier} \downarrow \text{ENV} \uparrow \text{ENV}[\text{NAME}]. \text{mode} \rangle \equiv \langle \text{name} \uparrow \text{NAME} \rangle
\]

Here we have an attribute expression, `\text{ENV}[\text{NAME}].\text{mode}`, in an applied position. This causes no problem: we just replace the expression by a new variable, say \text{MODE}, and insert an evaluation rule which makes \text{MODE} equal to \text{ENV}[\text{NAME}].\text{mode}:

\[
\langle \text{identifier} \downarrow \text{ENV} \uparrow \text{MODE} \rangle \equiv \langle \text{name} \uparrow \text{NAME} \rangle
\]

\[
\text{evaluate} \text{MODE} \leftarrow \text{ENV}[\text{NAME}].\text{mode}
\]

Example 2

\[
\langle \text{assignment} \downarrow \text{ENV} \rangle \equiv \langle \text{variable} \downarrow \text{ENV} \uparrow \text{TYPE} \rangle \equiv \langle \text{expression} \downarrow \text{ENV} \uparrow \text{TYPE} \rangle
\]

Here the \text{variable} \text{TYPE} occurs in two defining positions. To ensure that the variable receives a unique value, in accordance with the systematic substitution rule, we replace one occurrence of \text{TYPE} by a new variable, say \text{TYPE1}, and insert the constraint `\text{TYPE} = \text{TYPE1}':

\[
\langle \text{assignment} \downarrow \text{ENV} \rangle \equiv \langle \text{variable} \downarrow \text{ENV} \uparrow \text{TYPE} \rangle \equiv \langle \text{expression} \downarrow \text{ENV} \uparrow \text{TYPE1} \rangle
\]

\[
\text{where} \text{TYPE} = \text{TYPE1}
\]

Example 3

\[
\langle \text{variable} \downarrow \text{ENV} \uparrow \text{TYPE} \rangle \equiv \langle \text{variable} \downarrow \text{ENV} \uparrow \text{array}(\text{LB}, \text{UB}, \text{TYPE}) \rangle
\]

\[
\langle \text{variable} \downarrow \text{ENV} \uparrow \text{integer} \rangle \equiv \langle \text{variable} \downarrow \text{ENV} \uparrow \text{TYPE} \rangle
\]

Here we have two defining positions occupied by attribute expressions which are not simple variables. The constant attribute `integer' can be replaced by a new variable, say \text{TYPE1}, and the constraint `\text{TYPE1} = \text{integer}' inserted.

The synthesized attribute of \langle \text{variable} \rangle (on the right side of the rule) is more difficult. We know that this attribute must be in the domain

\[
\text{Type} = (\text{boolean} | \text{integer} | \text{array}((\text{Integer}, \text{Integer}, \text{Type}))
\]

but it will be necessary at evaluation-time to check that the attribute is indeed of the form \text{array}(\text{LB}, \text{UB}, \text{TYPE}), and thereby deduce the values of LB, UB and TYPE.

Now the composition function

\[
\text{array}: \text{Integer} \times \text{Integer} \times \text{Type} \rightarrow \text{Type}
\]

has a partial inverse function:

\[
\text{array}^{-1}: \text{Type} \rightarrow \text{Integer} \times \text{Integer} \times \text{Type}
\]

\[
\text{array}^{-1}(T) = \begin{cases} (3L, U, T) & (T = \text{array}(L, U, T)) \\ (L, U, T') & \text{else undefined} \end{cases}
\]

Thus we can replace the attribute expression `\text{array}(\text{LB}, \text{UB}, \text{TYPE})' by a new variable, say \text{TYPE2}, and insert an evaluation rule invoking the inverse function array\(^{-1} \)1:

\[
\langle \text{variable} \downarrow \text{ENV} \uparrow \text{TYPE} \rangle \equiv \langle \text{variable} \downarrow \text{ENV} \uparrow \text{TYPE2} \rangle
\]

\[
\langle \text{expression} \downarrow \text{ENV} \uparrow \text{TYPE1} \rangle \equiv \langle \text{variable} \downarrow \text{ENV} \uparrow \text{TYPE2} \rangle
\]

\[
\text{where} \text{TYPE} = \text{integer}
\]

\[
\text{evaluate} (\text{LB}, \text{UB}, \text{TYPE}) \leftarrow \text{array}^{-1} (\text{TYPE2})
\]

Clearly the last transformation will work only if the attribute expression in the defining position is composed only of invertible functions. Among the useful functions which do have (partial) inverses are the composition functions for Cartesian products, discriminated unions and sequences.

An EAG is well-formed if and only if:

(a) every variable occurs in at least one defining position in each rule in which it is used; and

(b) every function used in the composition of an attribute expression in a defining position has a (partial) inverse function.

These conditions do not seem to be too restrictive in practice. For example, the EAG in Appendix A is well-formed.

Any well-formed EAG can be converted into an equivalent AG by repeatedly applying the following transformations to each rule of the EAG.

(T1) Wherever an applied position contains an attribute expression e which is not a simple variable, choose some new variable x (i.e. one which is not already used in the rule) whose domain is the same as that of the applied position, replace e by x, and insert the evaluation rule `evaluate x \leftarrow e'.

(T2) Wherever a variable x occurs in n + 1 defining positions (n \geq 0), choose some new variables x_1, \ldots, x_n whose domains are the same as that of x, use them to replace all but one defining occurrence of x, and insert the constraint `where x = x_1 = \cdots = x_n'.

(T3) Wherever a constant attribute c occurs in a defining position, choose a new variable x, replace c by x, and insert the constraint `where x = c'.

(T4) Wherever a function application f(x_1, \ldots, x_n) occurs in a defining position, where x_1, \ldots, x_n are all variables, choose some new variable x, whose domain is the same as the range of f, replace f(x_1, \ldots, x_n) by x, and insert the evaluation rule `evaluate (x_1, \ldots, x_n) \leftarrow f^{-1}(x)'. (Such a function f^{-1} must exist, by condition (b) for well-formedness of an EAG.)

Now any evaluator for AGs can be adapted to well-formed EAGs as well. For example, the EAG in Appendix A is capable of being handled by a two-pass left-to-right evaluator.
5.3 Attribute-directed parsing

Most evaluators for AGs assume that the underlying CFG is deterministic (e.g. LL, LALR or LR). However, most "natural" grammars for programming languages contain ambiguities which are resolved by context. An EAG is a natural tool for expressing such ambiguities. A typical example of this is rule-group (10) in Appendix A. Here the underlying CFG is ambiguous, but the EAG is not.

References 9 and 26 mention the possibility of making the attributes influence the parsing. This would allow some AGs and EAGs with ambiguous underlying CFGs to be handled. This problem has not yet found a satisfactory general solution; the main difficulty is that it is undecidable whether the attributes do indeed resolve the ambiguity. For a further discussion of attribute-directed parsing, see Ref. 29.

5.4 The Aarhus compiler writing system

An experimental compiler writing system, NEATS, has been designed and implemented at Aarhus. NEATS accepts an EATG consisting of one input EAG and one output EAG, and constructs a translator according to this EATG.

The attribute domains available in NEATS are essentially those defined in Appendix A.

The constructed translator translates an input string into an output string, and if this is sufficient for the application then the user need supply no more than the EATG.

For most practical purposes, however, the user may wish to do more. Instead of generating an output string, the translator may be made to call a procedure each time an output symbol is to be generated. The output symbol and its associated attributes will then be passed as parameters to the procedure. This will be the situation when, for example, the EATG defines the analysis phase of a compiler, and the user himself programs the synthesis (code generation).

NEATS is programmed in Pascal and is an extension of the BOBS-system, which is an LALR(1) parser generator. Consequently, the CFG underlying the EATG must be LALR(1).

NEATS will accept any non-circular EATG. During parsing, the translator builds a directed acyclic graph defining the order of evaluation of the attributes. After parsing, a recursive scan of this graph will evaluate all the attributes. The parse tree itself is not stored. The reader is referred to Refs 18 and 22 for details of NEATS and the AG constructor algorithm adopted.

The practical value of this algorithm has to be investigated further; it is reasonably fast but uses a lot of store. The algorithm adopted is not essential for the use of EATGs; any other AG constructor algorithm could equally well have been adopted. However, the system is intended for experiments, so it was decided to have an implementation accepting all non-circular AGs rather than some more limited subclass.

The experiments to be done include the following:
(a) to test the system with some large grammars to measure its usefulness in generating parts of a production compiler
(b) to use the system in teaching
(c) to modify the CF parser constructor (the BOBS-system) to accept all LR(1) grammars, and certain ambiguous ones in order to experiment with attribute-directing parsing
(d) to make it possible to define a sequence of translations
(e) to investigate the possibilities and requirements for adding new domains and thus extend the fixed set of domains available in NEATS.

So far the results have been very promising.

6. CONCLUSIONS

We have introduced two new formalisms, the EAGs and the EATGs, which we believe come close to reconciling two conflicting ideals. On the one hand, these grammars are concise and readable, and therefore may be capable of making formal language definitions more widely acceptable than hitherto. On the other hand, they are also well suited to automatic compiler construction.

The advantages of EAGs and EATGs stem from their combination of the best features of other formalisms with some new ideas:
(a) the explicit attribute structure and the distinction between inherited and synthesized attributes
(b) the visibility of the underlying context-free syntax
(c) generative definition of languages (like context-free and van Wijngaarden grammars)
(d) the implicit and concise specification of context-sensitivities by means of attribute expressions in applied and defining positions
(e) the free choice of domain types.

We have found in practice that EAGs and EATGs are straightforward to write. Complete definitions of real programming languages can be found in Refs 3 and 21.

The abstract data types (partial maps, discriminated unions, etc.) used in the example are very well suited to describing attributes, in particular the 'environment' attributes in a programming language. Certainly, the same attributes can be represented by strings, as in van Wijngaarden grammars or extended affix grammars, but this leads to some artificiality; compare, for example, rule (19) in Appendix A with the corresponding syntax in Ref. 10. Likewise, the tree structure of 'objects' in Vienna definition language is not always the most natural structure.

Evidently, the definitive power of EAGs and EATGs rests largely on the power of the functions used to compose attribute expressions. These functions may be arbitrarily powerful, and their definition is not part of the formalism itself. One could abuse this power by making the functions do most of the work of language definition—in the extreme case, using a single function which accepts or rejects a complete program—but obviously this would help no-one. We have avoided any such cheating, in our examples, by using only well-known abstract domain types and functions; grammatically defined predicates (e.g. rule-group (17) in Appendix A) can be used to avoid inventing special-purpose functions.

We have briefly described an experimental compiler writing system which has been implemented at Aarhus. This system accepts a large subclass of EATGs, and it
EXTENDED ATTRIBUTE GRAMMARS
demonstrates the feasibility of using an EATG to automate the construction of the analysis phase of a compiler. It is being used to investigate the practicality of this approach and some other open problems.
The automation of the synthesis (code-generation) phase of a compiler has not been treated in this paper, but AGs and EAGs have an application here too.  
A very interesting recent development of EAGs is the work of Paulson. Paulson’s ‘semantic grammars’ are EAGs in which some of the attributes are semantic denotations. Thus a semantic grammar can provide a complete (syntactic and semantic) definition of a programming language. Paulson has implemented a compiler writing system whose input is a semantic grammar. The generated compiler parses the source program and evaluates the semantic attributes, using an evaluator very similar to that of Madsen. Finally it translates the resulting lambda-expression into code for the SECOS machine, which is subsequently interpreted. Both the compilation and interpretation phases are much more efficient than SIS. Paulson has used his system to generate compilers for large subsets of Pascal and FORTRAN.

One flaw of EAGs is that they tend to be monolithic. EATGs possess a degree of modularity in their separation of the output grammar from the input grammar. One of us (Watt) is currently investigating how language definitions can be made even more modular, by partitioning both the input grammar and the output grammar.

Acknowledgements

We are grateful for many helpful comments and encouragement from Frank DeRemer, Mehdi Jazayeri, Steve Muchnick, Bob Tennent, and others. We are also happy to acknowledge the valuable work of Poul Jespersen, Michael Madsen and Hanne Riis in implementing the NEATS system.

REFERENCES


Received June 1982
APPENDIX A. A COMPLETE EXAMPLE OF AN EAG

To support our claim that EAGs are well suited to language definition, we give here a grammar completely defining the syntax of a small but realistic programming language. The language chosen is a subset of Pascal containing the following features:
(a) boolean, integer, and array data types
(b) variable declarations
(c) procedure declarations, with value- and variable-parameters
(d) assignments, procedure calls, compound-, if- and while-statements
(e) expressions involving integer and relational operators
(f) the usual Pascal block structure, but no requirement of declaration-before-use for procedures.

A.1 Domain types

Apart from certain base types, we shall use domains of the following types, which may be recursive. They are based on the abstract data types of Ref. 33 and the extended domain types of Scott.

Cartesian products. If \( T_1, \ldots, T_n \) are domains and \( f_1, \ldots, f_n \) are distinct names, then
\[
P = (f_1 : T_1; \ldots; f_n : T_n)
\]
is a Cartesian product with field selectors \( f_1, \ldots, f_n \).
For every \( a_i \) in \( T_1, \ldots, \) and every \( a_n \) in \( T_n \), \( (a_1, \ldots, a_n) \) is in \( P \). This is the composition function for the Cartesian product \( P \).
For every \( p \) in \( P \), and for every \( i = 1, \ldots, n \), \( p.f_i \) is in \( T_i \), and denotes the \( i \)th field of \( p \).

Discriminated unions. If \( T_1, \ldots, T_n \) are domains (or Cartesian products of domains) and \( g_1, \ldots, g_n \) are distinct names, then
\[
U = (g_1(T_1) \ldots | g_n(T_n))
\]
is a discriminated union with selectors \( g_1, \ldots, g_n \). If any \( T_i \) is void, then we abbreviate \( g_i(T_i) \) to \( g_i \).
For every \( i = 1, \ldots, n \), and for every \( a_i \) in \( T_i \), \( g_i(a_i) \) is in \( U \). These \( g_i \) are the composition functions for the discriminated union \( U \).

Maps. If \( D \) and \( R \) are domains, then
\[
M = D \rightarrow R
\]
is the domain of (partial) maps from \( D \) to \( R \).
For every \( d \) in \( D \) and \( m \) in \( M \), \( m[d] \) either is in \( R \) or is undefined. This is the application function for the map \( M \).
\([ \] \) denotes the map defined at no point in \( D \). If \( d_1, \ldots, d_n \) are distinct elements of \( D \) and \( r_1, \ldots, r_n \) are in \( R \), then \([d_1 \rightarrow r_1, \ldots, d_n \rightarrow r_n] \) is in \( M \), and denotes a map defined at points \( d_1, \ldots, d_n \) and nowhere else.
For each \( m_1 \) and \( m_2 \) in \( M \), \( m_1 \cup m_2 \) is the disjoint union of \( m_1 \) and \( m_2 \); \( m_1 \cup m_2 \) is undefined if, for any \( d \) in \( D \), both \( m_1[d] \) and \( m_2[d] \) are defined; otherwise
\[
(m_1 \cup m_2)[d] \equiv \begin{cases} m_1[d] & \text{if } m_1[d] \text{ is defined} \\ m_2[d] & \text{else} \end{cases}
\]

For each \( m_1 \) and \( m_2 \) in \( M \), \( m_1 \| m_2 \) is the map \( m_1 \) overridden by \( m_2 \); i.e.
\[
(m_1 \| m_2)[d] \equiv \begin{cases} m_1[d] & \text{if } m_1[d] \text{ is undefined} \\ m_2[d] & \text{else } m_2[d] \end{cases}
\]

Sequences. If \( D \) is a domain, then
\[
S = D^*
\]
is the domain of sequences of elements of \( D \).
\( <> \) denotes the empty sequence. \( <d> \) denotes the sequence containing the single component \( d \).
If \( s \) is in \( S \) and \( d \) is in \( D \), then \( d^s \) denotes the sequence obtained by prepending \( d \) to \( s \).

A.2 Domain definitions

Environment = Name \rightarrow (declarationdepth : Level; mode : Mode)

Mode = (variableType | formal(Parameter) | procedure(Plan))

Plan = Parameter*

Parameter = (valueType | var(Type) )

Type = (boolean | integer | array(Integer, Integer, Type))

Operator = (equal | unequal | plus | minus)

Level = Integer

Integer and Name are primitive domains of integers and names, respectively.

A.3 Vocabulary

Here is a list of those terminal symbols which have attributes, showing the types and domains of their attribute-positions. (All are synthesized and have base domains.)

\(<\text{name} >\) Name
\(<\text{integer number} >\) Integer

All other terminal symbols are written enclosed in quotes (".").

Here is a complete list of non-terminal symbols, showing the type and domain of each attribute-position, and also the number of the rule-group defining each non-terminal.

\(<\text{actual parameter} >\) Environment Parameter 10
\(<\text{actual parameter list} >\) Environment Plan 9
\(<\text{adding operator} >\) Operator 16
\(<\text{assignment} >\) Environment 3
\(<\text{block} >\) Environment Environment 20
\(<\text{compound statement} >\) Environment 5
\(<\text{constant} >\) Type 14
\(<\text{expression} >\) Environment Type 11
\(<\text{formal parameter} >\) Level Parameter 26
\(<\text{formal parameter list} >\) Level Plan 25
\(<\text{identifier} >\) Environment Mode 19
\(<\text{if statement} >\) Environment 7
EXTENDED ATTRIBUTE GRAMMARS

A.4 Attribute variables

Here is a complete list of attribute variables used in the rules, together with their domains.

- ENV, DECL, DECLS, NONLOCALS, FORMALS, VARS, PROC
- MODE : Environment
- PLAN : Plan
- PARM : Parameter
- TYPE, TYPE1, TYPE2 : Type
- OP : Operator
- DEPTH : Level
- LB, UB, VALUE : Integer
- NAME : Name

A.5 Rules

Comments are enclosed in (*...*). These are used primarily to draw attention to some of the context-sensitive constraints enforced by the grammar.

(* Most non-terminals have an inherited attribute representing their 'environment'. *)

(* PROGRAMS *)

<program> ::= (1) <block 0 [ ] [ ] ".."

(* STATEMENTS *)

<statement ENV> ::= (2a) <assignment ENV>
(2b) <procedure call ENV>
(2c) <compound statement ENV>
(2d) <if statement ENV>
(2e) <while statement ENV>

<assignment ENV> ::= (3) <variable ENV TYPE "="
<expression ENV TYPE>

<procedure call ENV> ::= (4) <identifier ENV procedure(PLAN)
"(" <actual parameter list ENV PLAN>
")"
<compound statement ENV> ::= (5) "begin" <serial ENV> "end"

<serial ENV> ::= (6a) <statement ENV>
(6b) <serial ENV> ";"
<if statement ENV> ::= (7) "if" <expression ENV BOOLEAN>
"then" <statement ENV>
"else" <statement ENV>

<while statement ENV> ::= (8) "while" <expression ENV BOOLEAN>
"do" <statement ENV>

(* ACTUAL PARAMETERS *)

<actual parameter list ENV PARM> ::= (9a) <actual parameter ENV PARM>
<actual parameter list ENV PARM PLAN> ::= (9b) <actual parameter ENV PARM> ";"
<actual parameter list ENV PLAN> ::= (10a) <expression ENV TYPE>
<actual parameter ENV var(TYPE) ::= (10b) <variable ENV TYPE>

(* The actual parameters in a procedure call must correspond, left to right, with the formal parameters in the procedure declaration, as summarized in the second attribute of <actual parameter list>. Corresponding to a value-parameter, the actual parameter must be an expression of the same type (10a). Corresponding to a variable-parameter, the actual parameter must be a variable of the same type (10b). *)

(* EXPRESSIONS *)

(* Each of <expression>, <simple expression>, <term>, (constant) and <variable> has a synthesized attribute representing its type. *)

<expression ENV TYPE> ::= (11a) <simple expression ENV TYPE>
<expression ENV BOOLEAN> ::= (11b) <simple expression ENV TYPE1>
<relational operator OP>
<simple expression ENV TYPE2>
<where comparable TYPE1 TYPE2> ::= (12a) <term ENV TYPE>
<simple expression ENV integer> ::= (12b) <simple expression ENV integer>
<add operator OP>
<term ENV integer>
<term ENV TYPE> ::= (13a) <constant TYPE>
(13b) <variable ENV TYPE>
(13c) "(" <expression ENV TYPE>")"
<constant BOOLEAN> ::= (14a) "false"
(14b) "true"
<constant integer> ::= (14c) <integer number VALUE>

<relational operator equal> ::=
(15a)  "="
\langle\text{relational operator } \\
\text{\[ unequal\]} \rangle ::=
(15b)  "\lt\rt"
\langle\text{adding operator } \\
\text{\[ plus\]} \rangle ::=
(16a)  "\rt"
\langle\text{adding operator } \\
\text{\[ minus\]} \rangle ::=
(16b)  "\lt"
\langle\text{where comparable } \\
\text{\[ integer\]} \rangle ::=
(17a)  \langle\text{empty} \rangle
\langle\text{where comparable } \\
\text{\[ boolean\]} \rangle ::=
(17b)  \langle\text{empty} \rangle

(* The non-terminal \langle where comparable \rangle acts as a 
\text{predicate, since all its terminal productions are empty; it 
serves to enforce type compatibility. }*)

(* VARIABLES AND IDENTIFIERS *)

(18a)  \langle\text{variable } \text{\[ ENV} \rangle ::=
(18b)  \langle\text{identifier } \text{\[ ENV} \rangle ::=
(18c)  \langle\text{identifier } \text{\[ ENV} \text{\# formal(value(TYPE))] \rangle ::=
(18d)  \langle\text{variable } \text{\[ ENV} \text{\# array(LB, UB, TYPE)]} \text{\[ "[" \]} \text{\[ expression \]} \text{\[ ENV} \text{\# integer] \text{\[ "]"} \rangle ::=

(* (18b) and (18c) allow value- and variable-
\text{parameters to be used like ordinary variables. (18d) allows 
a variable of array type to be subscripted by an 
integer expression. *)

\langle\text{identifier } \text{\[ ENV} \text{\[ NAME]\# mode] \rangle ::=
(19)  \langle\text{name } \text{\[ NAME]\# mode] \rangle ::=

(* \langle\text{identifier} \rangle has a synthesized attribute 
\text{representing its mode, which is determined by looking up 
the name of the identifier in the 
"environment". }*)

(* DECLARATIONS *)

(20)  \langle\text{block } \text{\[ DEPTH} \text{\[ NONLOCALS} \text{\[ FORMALS]} \rangle ::=
\langle\text{variable declarations } \text{\[ DEPTH} \text{\[ VARS]} \rangle ::=
\langle\text{procedure declarations } \text{\[ DEPTH} \text{\[ NONLOCALS} \text{\[ FORMALS} \rangle ::=
\text{\[ VARS} \rangle ::=
\text{\[ PROCOS] \rangle ::=
\langle\text{compound statement } \text{\[ ENV} \rangle ::=
\langle\text{NONLOCALS} \text{\[ FORMALS} \rangle ::=
\text{\[ VARS} \rangle ::=
\langle\text{PROCOS} \rangle ::=

(* The first attribute of \langle block\rangle is its depth of 
nesting. \langle block\rangle also has two inherited "environment 
\text{attributes, representing non-local identifiers and 
local formal parameters, respectively. The latter attribute 
\text{\[ FORMALS]} is disjointly united with the local variable identifiers 
\text{\[ VARS]} and local procedure identifiers \text{\[ PROCOS]} to form the set of 
local identifiers: \text{\[ FORMALS} \text{\[ VARS} \rangle ::=
this then overrides the non-local identifiers to form the 
"environment" inside the block: \text{\[ NONLOCALS} \text{\[ FORMALS} \rangle ::=
\text{\[ VARS} \rangle ::=
\text{\[ PROCOS] \rangle ::=

(* TYPES *)

(21a)  \langle\text{variable declarations } \text{\[ DEPTH} \text{\[ DECL] \rangle ::=
\langle\text{variable } \text{\[ DEPTH} \text{\[ DECL] \rangle ::=
\langle\text{variable declarations } \text{\[ DEPTH} \text{\[ DECL] \rangle ::=
\langle\text{variable declarations } \text{\[ DEPTH} \text{\[ DECL] \rangle ::=

(* PROCEDURES *)

(22a)  \langle\text{procedure declarations } \text{\[ DEPTH} \text{\[ ENV} \rangle ::=
\langle\text{procedure declarations } \text{\[ DEPTH} \text{\[ ENV} \rangle ::=
\langle\text{procedure declarations } \text{\[ DEPTH} \text{\[ ENV} \rangle ::=

(23a)  \langle\text{empty} \rangle
\langle\text{procedure declarations } \text{\[ DEPTH} \text{\[ DECL]} \rangle ::=
\langle\text{procedure declarations } \text{\[ DEPTH} \text{\[ DECL]} \rangle ::=
\langle\text{procedure declarations } \text{\[ DEPTH} \text{\[ DECL]} \rangle ::=

(24a)  \langle\text{procedure declarations } \text{\[ DEPTH} \text{\[ ENV} \rangle ::=
\langle\text{procedure declarations } \text{\[ DEPTH} \text{\[ DECL]} \rangle ::=
\langle\text{procedure declarations } \text{\[ DEPTH} \text{\[ DECL]} \rangle ::=

(25a)  \langle\text{procedure declarations } \text{\[ DEPTH} \text{\[ DECL]} \rangle ::=
\langle\text{formal parameter list } \text{\[ DEPTH} \text{\[ PLAN]} \rangle ::=
\langle\text{formal parameter list } \text{\[ DEPTH} \text{\[ PLAN]} \rangle ::=
\langle\text{formal parameter list } \text{\[ DEPTH} \text{\[ PLAN]} \rangle ::=

(26a)  \langle\text{formal parameter list } \text{\[ DEPTH} \text{\[ value(TYPE)]} \rangle ::=
\langle\text{formal parameter list } \text{\[ DEPTH} \text{\[ value(TYPE)]} \rangle ::=
\langle\text{formal parameter list } \text{\[ DEPTH} \text{\[ var(TYPE)]} \rangle ::=

(27a)  \langle\text{type } \text{\[ boolean] \rangle ::=
\langle\text{type } \text{\[ integer] \rangle ::=
\langle\text{type } \text{\[ array(LB, UB, TYPE)]} ::=
\langle\text{integer number } \text{\[ LB] \rangle ::=
\langle\text{integer number } \text{\[ UB] \rangle ::=

152 THE COMPUTER JOURNAL, VOL. 26, NO. 2, 1983
EXTENDED ATTRIBUTE GRAMMARS

APPENDIX B. AN EXAMPLE OF AN EATG

Here we enhance the EAG of Appendix A to an EATG which defines the translation of the
programming language into an intermediate language which has the following features:

(a) expressions are in postfix form
(b) each identifier is made unique by attaching to it the
depth of nesting of the block where it was declared
(c) control structures are completely bracketed, and the
level of control structure nesting is attached to each
bracket

Much more could be done, but for the sake of simplicity
we restrict ourselves to the above.

B.1 Additional vocabulary

Here is a list of those output terminal symbols which
have attributes.

\[ \langle \text{declare} \downarrow \text{LEVEL} \downarrow \text{Name} \downarrow \text{Mode} \rangle \]
\[ \langle \text{dyadic} \downarrow \text{Operator} \downarrow \text{Type} \downarrow \text{Type} \rangle \]
\[ \langle \text{do} \downarrow \text{LEVEL} \rangle \]
\[ \langle \text{else} \downarrow \text{LEVEL} \rangle \]
\[ \langle \text{fi} \downarrow \text{LEVEL} \rangle \]
\[ \langle \text{if} \downarrow \text{LEVEL} \rangle \]
\[ \langle \text{index} \downarrow \text{Integer} \downarrow \text{Integer} \rangle \]
\[ \langle \text{name} \downarrow \text{LEVEL} \downarrow \text{Name} \rangle \]
\[ \langle \text{number} \downarrow \text{LEVEL} \rangle \]
\[ \langle \text{od} \downarrow \text{LEVEL} \rangle \]
\[ \langle \text{procedure} \downarrow \text{LEVEL} \downarrow \text{Name} \rangle \]
\[ \langle \text{store} \downarrow \text{Type} \rangle \]
\[ \langle \text{then} \downarrow \text{LEVEL} \rangle \]
\[ \langle \text{while} \downarrow \text{LEVEL} \rangle \]

Here is a list of non-terminal symbols, showing the
type and domain of each attribute-position used in the
output grammar. Non-terms which have no such
attribute-position are omitted.

\[ \langle \text{compound statement} \downarrow \text{LEVEL} \rangle \]
\[ \langle \text{if statement} \downarrow \text{LEVEL} \rangle \]
\[ \langle \text{serial} \downarrow \text{LEVEL} \rangle \]
\[ \langle \text{statement} \downarrow \text{LEVEL} \rangle \]
\[ \langle \text{while statement} \downarrow \text{LEVEL} \rangle \]

B.2 Additional attribute variables

LEVEL:Level

B.3 Output rules

For each input rule in Appendix A we give here only the
corresponding output rule. For the sake of brevity, we
omit output rules which contain no output symbols, and
in which there is no reordering of the non-terms, and
in which attributes are merely copied.

(* PROGRAMS *)

\[ \langle \text{program} \rangle \::= \]
(1) \quad \text{program} \langle \text{block} \rangle \text{endprogram} \]

(* STATEMENTS *)

(* Each of \langle \text{statement} \rangle, \langle \text{compound statement} \rangle, \langle \text{if}
statement \rangle \text{ and } \langle \text{while statement} \rangle \text{ has an inherited}
attribute which is its level of control structure
nesting. The level of nesting starts at 0 in each
block—rule (20). *)

\[ \langle \text{assignment} \rangle \::= \]
(3) \quad \langle \text{variable} \rangle \langle \text{expression} \rangle \langle \text{store} \downarrow \text{TYPE} \rangle \]
\[ \langle \text{procedure call} \rangle \::= \]
(4) \quad \langle \text{actual parameter list} \rangle \langle \text{identifier} \rangle \text{ call} \]
\[ \langle \text{if statement} \downarrow \text{LEVEL} \rangle \::= \]
(7) \quad \langle \text{if} \downarrow \text{LEVEL} \rangle \langle \text{expression} \rangle \langle \text{then} \downarrow \text{LEVEL} \rangle \langle \text{statement} \downarrow \text{LEVEL} + 1 \rangle \]
\[ \langle \text{else} \downarrow \text{LEVEL} \rangle \langle \text{statement} \downarrow \text{LEVEL} + 1 \rangle \]
\[ \langle \text{fi} \downarrow \text{LEVEL} \rangle \]
\[ \langle \text{while statement} \downarrow \text{LEVEL} \rangle \::= \]
(8) \quad \langle \text{while} \downarrow \text{LEVEL} \rangle \langle \text{expression} \rangle \langle \text{do} \downarrow \text{LEVEL} \rangle \langle \text{statement} \downarrow \text{LEVEL} + 1 \rangle \langle \text{od} \downarrow \text{LEVEL} \rangle \]

(* ACTUAL PARAMETERS *)

\[ \langle \text{actual parameter} \rangle \::= \]
(10a) \quad \langle \text{expression} \rangle \text{ valueparameter} \]
\[ \langle \text{actual parameter} \rangle \::= \]
(10b) \quad \langle \text{variable} \rangle \text{ varparameter} \]

(* EXPRESSIONS *)

\[ \langle \text{expression} \rangle \::= \]
(11b) \quad \langle \text{simple expression} \rangle \langle \text{simple expression} \rangle \langle \text{dyadic} \downarrow \text{OP} \downarrow \text{TYPE1} \downarrow \text{TYPE2} \rangle \]
\[ \langle \text{simple expression} \rangle \::= \]
(12b) \quad \langle \text{simple expression} \rangle \langle \text{term} \rangle \langle \text{dyadic} \downarrow \text{OP} \downarrow \text{integer} \downarrow \text{integer} \rangle \]
\[ \langle \text{constant} \rangle \::= \]
(14a) \quad \text{false} \]
(14b) \quad \text{true} \]
\[ \langle \text{constant} \rangle \::= \]
(14c) \quad \langle \text{number} \downarrow \text{VALUE} \rangle \]

(* VARIABLES AND IDENTIFIERS *)

\[ \langle \text{variable} \rangle \::= \]
(18d) \quad \langle \text{variable} \rangle \langle \text{expression} \rangle \langle \text{index} \downarrow \text{LB} \downarrow \text{UB} \rangle \]
\[ \langle \text{identifier} \rangle \::= \]
(19) \quad \langle \text{name} \downarrow \text{ENV[NNAME].declarationdepth} \downarrow \text{NAME} \rangle \]

(* DECLARATIONS *)

\[ \langle \text{block} \rangle \::= \]
(20) \quad \langle \text{variable declarations} \rangle \langle \text{compound statement} \downarrow 0 \rangle \langle \text{procedure declarations} \rangle \]
\[ \langle \text{variable declaration} \rangle \::= \]
(22) \quad \langle \text{declare} \downarrow \text{DEPTH} \downarrow \text{NAME} \downarrow \text{variable(TYPE)} \rangle \]
\[ \langle \text{procedure declaration} \rangle \::= \]
(24) \quad \langle \text{procedure} \downarrow \text{DEPTH} \downarrow \text{NAME} \rangle \langle \text{formal parameter list} \rangle \langle \text{block} \rangle \endprocedure \]
\[ \langle \text{formal parameter} \rangle \::= \]
(26a) \quad \langle \text{declare} \downarrow \text{DEPTH} \downarrow \text{NAME} \downarrow \text{formal(value(TYPE))} \rangle \]
\[ \langle \text{formal parameter} \rangle \::= \]
(26b) \quad \langle \text{declare} \downarrow \text{DEPTH} \downarrow \text{NAME} \downarrow \text{formal(var(TYPE))} \rangle \]

THE COMPUTER JOURNAL, VOL. 26, NO. 2. 1983 153