Extended Attribute Grammars

David A. Watt
Computing Science Department, University of Glasgow, Glasgow G12 8QQ, UK
Ole Lehrmann 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 aims 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[1] contained hidden semantic irregularities which were revealed only by formalization of its semantics.[2] 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.[3] 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[4] 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, an 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.[5, 6]

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.

CCC-0010-4620/83/0026-0142$06.00

The empty sequence is denoted by ⟨empty⟩. 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) ⟨assignment⟩ : : = ⟨identifier⟩ ": =" ⟨expression⟩
(2) ⟨identifier⟩ : : = ⟨name⟩

2.1 Attribute grammars and affix grammars

Attribute grammars were devised by Knuth,[7] and affix grammars independently by Koster.[8] 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 (↓) and synthesized attributes by upward arrows (↑).

In our example, each of the non-terminals ⟨assignment⟩, ⟨identifier⟩ and ⟨expression⟩ will have an inherited attribute representing its 'environment' (inherited since it represents information about the context). Each of ⟨identifier⟩ and ⟨expression⟩ will also have a synthesized attribute representing its mode. The symbol ⟨name⟩ 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:

```
    ⟨assignment ↓ ENV⟩ : : =
(1)     ⟨identifier ↓ ENV1 ↑ MODE1⟩ ": =" 
            ⟨expression ↓ ENV2 ↑ MODE2⟩
            evaluate ENV1 ← ENV
            evaluate ENV2 ← ENV
            where MODE1 = ref(MODE2)
```

'*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 ⟨identifier⟩ and ⟨expression⟩ are to be made equal to the environment attribute of ⟨assignment⟩. The constraint specifies the relation which must hold between the mode attributes of ⟨identifier⟩ and ⟨expression⟩.

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

```
    ⟨identifier ↓ ENV ↑ MODE⟩ : : =
(2)     ⟨name ↑ NAME⟩
            evaluate MODE ← ENV[NAME]
```

Here we compute the mode attribute of ⟨identifier⟩ by applying the map ENV to NAME, the attribute of ⟨name⟩, where ENV is the environment attribute of ⟨identifier⟩. 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:

$$\underset{\text{def}\ \ \text{app}}{\langle X\downarrow\ldots\uparrow\ldots\rangle} : : = \langle X_1\downarrow\ldots\uparrow\ldots\rangle\ldots\underset{\text{app}\ \ \text{def}}{\langle X_m\downarrow\ldots\uparrow\ldots\rangle}$$

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:

```
    ⟨assignment ↓ ENV⟩ : : =
(1)     ⟨identifier ↓ ENV ↑ MODE1⟩ ": =" 
            ⟨expression ↓ ENV ↑ MODE2⟩
            where MODE1 = ref(MODE2)
```

The choice of ↓ and ↑ 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.

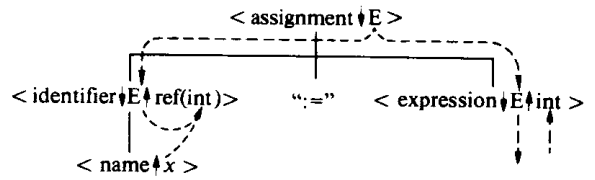

Figure 1. Fragment of an attributed syntax tree. The input string is: x := ⟨expression⟩. E stands for the attribute [$x \rightarrow$ ref(int), $y \rightarrow$ bool]. Broken arrows leading to each attribute indicate which other attributes it depends upon.

Attribute grammars have been used to define the context-sensitive syntax of several programming languages. Relative to van Wijngaarden grammars,[10] 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.

AGs are well suited to compiler construction, and have been exploited in many compiler writing systems.[11–18] We shall return to this topic in Section 5.

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,[10] 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:

⟨identifier ↓ ENV ↑ ENV[NAME]⟩ : : =
(2) ⟨name ↑ NAME⟩

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:

⟨assignment ↓ ENV⟩ : : =
(1) ⟨identifier ↓ ENV ↑ ref(MODE)⟩ ": = "
⟨expression ↓ ENV ↑ MODE⟩

⟨identifier ↓ ENV ↑ ENV[NAME]⟩ : : =
(2) ⟨name ↑ NAME⟩

In rule (1) we have specified the relation which must hold between the second attribute, MODE, of ⟨expression⟩ and the second attribute of ⟨identifier⟩ simply by writing 'ref(MODE)' in the latter position. Similarly, in rule (2) we have specified that the second attribute of ⟨identifier⟩ is obtained by applying ENV to NAME simply by writing '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 $[x \rightarrow$ ref(int), $y \rightarrow$ bool] for ENV and x for NAME in rule (2), and evaluating ENV[NAME], we get the production rule

⟨identifier ↓ [$x \rightarrow$ ref(int), $y \rightarrow$ bool] ↑ ref(int)⟩ : : =
⟨name ↑ x⟩

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 NAME, we find that the value of ENV[NAME] is not defined; therefore no production rule can be generated.

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

⟨assignment ↓ [$x \rightarrow$ ref(int), $y \rightarrow$ bool]⟩ : : =
⟨identifier ↓ [$x \rightarrow$ ref(int), $y \rightarrow$ bool] ↑ ref(int)⟩
": = "⟨expression ↓ [$x \rightarrow$ ref(int), $y \rightarrow$ bool] ↑ int⟩

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

3. FORMAL DEFINITION OF EXTENDED ATTRIBUTE GRAMMARS

An *extended attribute grammar* is a 5-tuple

$$G = \langle D, V, Z, B, R \rangle$$

whose elements are defined in the following paragraphs.

$D = (D1, D2, \ldots, f1, f2, \ldots)$ is an algebraic structure with domains $D1$, $D2$, . . ., and (partial) functions $f1$, $f2$, . . . 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:

$$F ::= F_1 \ldots F_m$$

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 $F ::= F_1 \ldots F_m$ 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:

$$A ::= A_1 \ldots A_m$$

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 $A ::= A_1 \ldots A_m$ 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-sensitiveness 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 example 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:

⟨*LOAD* ↓Register ↓Integer⟩
⟨*ADD* ↓Register ↓Integer⟩
⟨*STORE* ↓Register ↓Integer⟩
⟨*MPY* ↓Integer⟩
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

store. We omit rules for defining this attribute since this is fully demonstrated in Appendix A. We suppose that the non-terminal ⟨evaluation⟩ 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) ⟨evaluation ↓ENV⟩ : : = ⟨expr ↓ENV⟩
(2) ⟨expr ↓ENV⟩ : : = ⟨expr ↓ENV⟩ "+" ⟨term ↓ENV⟩
(3) | ⟨term ↓ENV⟩
(4) ⟨term ↓ENV⟩ : : = ⟨term ↓ENV⟩ "*" ⟨factor ↓ENV⟩
(5) | ⟨factor ↓ENV⟩
(6) ⟨factor ↓ENV⟩ : : = "(" ⟨expr ↓ENV⟩ ")"
(7) | ⟨name ↑NAME⟩

Output rules

(1) ⟨evaluation⟩ : : = ⟨expr ↓a ↑H⟩
(2) ⟨expr ↓REG ↑max(H1, H2) + 1⟩ : : = ⟨term ↓a ↑H1⟩ ⟨*STORE* ↓a ↓H2⟩ ⟨expr ↓REG ↑H2⟩ ⟨*ADD* ↓REG ↓H2⟩
(3) ⟨expr ↓REG ↑H⟩ : : = ⟨term ↓REG ↑H⟩
(4) ⟨term ↓REG ↑max(H1, H2) + 1⟩ : : = ⟨factor ↓a ↑H1⟩ ⟨*STORE* ↓a ↓H2⟩ ⟨term ↓b ↑H2⟩ ⟨*MPY* ↓H2⟩ ⟨move ↓REG⟩
(5) ⟨term ↓REG ↑H⟩ : : = ⟨factor ↓REG ↑H⟩
(6) ⟨factor ↓REG ↑H⟩ : : = ⟨expr ↓REG ↑H⟩
(7) ⟨factor ↓REG ↑1⟩ : : = ⟨*LOAD* ↓REG ↓ENV[NAME]⟩
⟨move ↓a⟩ : : = ⟨empty⟩
⟨move ↓b⟩ : : = *ATOB*

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.

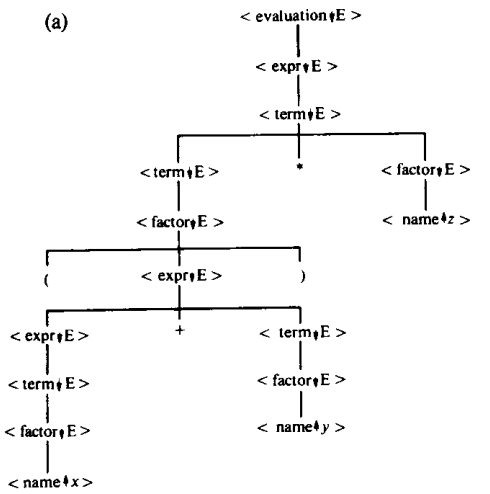

Figure 2(a). Attributed syntax tree of the input string $(x + y) * z$. E is the value of ENV in the given context.

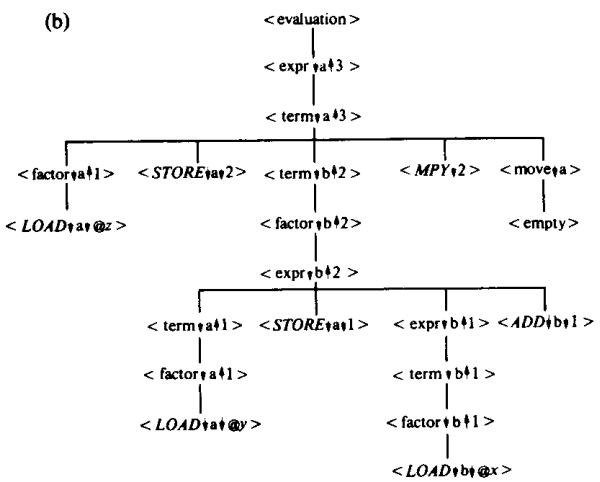

Figure 2(b). Attributed syntax tree of the output string corresponding to Fig. 2(a). @x, @y, @z denote the addresses of x, y, z.

(c) Defining a translation into some lambda-notation that may be 'executed' by a lambda reducer.[22] An example of this is the language LAMB of SIS,[23] which is a compiler generator based upon denotational semantics;[24] 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.[22]

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.[7]

A decade of research has produced a variety of attribute evaluators for non-circular AGs. These include

one-pass evaluators,[8,15,25,26] multi-pass left-to-right evaluators,[9] multi-pass alternating evaluators,[27] and multi-sweep evaluators.[28] 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[16] and NEATS[18]) 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

⟨identifier ↓ ENV ↑ ENV[NAME].mode⟩ : : =
(19) ⟨name ↑ NAME⟩

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

⟨identifier ↓ ENV ↑ MODE⟩ : : =
⟨name ↑ NAME⟩
evaluate MODE ← ENV[NAME].mode

Example 2

⟨assignment ↓ ENV⟩ : : =
(3) ⟨variable ↓ ENV ↑ TYPE⟩ ":=" ⟨expression ↓ ENV ↑ TYPE⟩

Here the variable 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 TYPE by a new variable, say TYPE1, and insert the constraint 'TYPE = TYPE1':

⟨assignment ↓ ENV⟩ : : =
⟨variable ↓ ENV ↑ TYPE⟩ ":=" ⟨expression ↓ ENV ↑ TYPE1⟩
where TYPE = TYPE1

Example 3

⟨variable ↓ ENV ↑ TYPE⟩ : : =
(18d) ⟨variable ↓ ENV ↑ array(LB, UB, TYPE)⟩ "[" ⟨expression ↓ ENV ↑ integer⟩ "]"

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 TYPE1, and the constraint 'TYPE1 = integer' inserted.

The synthesized attribute of ⟨variable⟩ (on the right-side of the rule) is more difficult. We know that this attribute must be in the domain

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

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

Now the composition function

array: Integer × Integer × Type → Type

has a partial inverse function:

array^{-1}: Type → Integer × Integer × Type

$\text{array}^{-1}(T) \equiv$ if $(\exists L, U, T')(T = \text{array}(L, U, T'))$
then (L, U, T')
else undefined

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

⟨variable ↓ ENV ↑ TYPE⟩ : : =
⟨variable ↓ ENV ↑ TYPE2⟩,
"[" ⟨expression ↓ ENV ↑ TYPE1⟩ "]"
where TYPE1 = integer
evaluate (LB, UB, TYPE) ← array^{-1}(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 ($n > 0$), 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.[18] 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.[30] 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[10] or extended affix grammars,[6] 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[4] 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

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.[12]

A very interesting recent development of EAGs is the work of Paulson.[31] 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.[22] Finally it translates the resulting lambda-expression into code for the SECD machine, which is subsequently interpreted. Both the compilation and interpretation phases are much more efficient than SIS.[23] 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.[32]

Acknowledgements

We are grateful for many helpful comments and encouragement from Frank DeRemer, Mehdi Jazayeri, Steve Muchnick, Bob Tennent, the referee, 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

1. N. Wirth, The programming language Pascal. *Acta Informatica* **1**, 35–63 (1971).
2. C. A. R. Hoare and N. Wirth, An axiomatic definition of the programming language Pascal. *Acta Informatica* **2**, 335–355 (1973).
3. D. A. Watt, An extended attribute grammar for Pascal. *SIGPLAN Notices* **14** (2), 60–74 (1979).
4. M. Marcotty, H. F. Ledgard and G. V. Bochmann, A sampler of formal definitions. *Computing Surveys* **8**, 191–276 (1976).
5. O. L. Madsen, On the use of attribute grammars in a practical translator writing system. Master Thesis, Aarhus University (July 1975).
6. D. A. Watt, Analysis-oriented two-level grammars. PhD Thesis, University of Glasgow (January 1974).
7. D. E. Knuth, Semantics of context-free languages. *Mathematical Systems Theory* **2**, 127–145 (1968).
8. C. H. A. Koster, Affix grammars. In: *ALGOL 68 Implementation* ed. by J. E. Peck, pp. 95–109. North-Holland, Amsterdam (1971).
9. G. V. Bochmann, Semantic evaluation from left to right. *Communications of the ACM* **19**, 55–62 (1976).
10. A. Van Wijngaarden, B. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens and R. G. Fisker, Revised Report on the Algorithmic Language ALGOL 68. *Acta Informatica* **5**, 1–236 (1975); Springer, Berlin (1976).
11. R. Farrow, LINGUIST-86: yet another translator writing system based on attribute grammars. *Proceedings of the SIGPLAN 82 Symposium on Compiler Construction, SIGPLAN Notices* **17** (6), 160–171 (1982).
12. H. Ganzinger, K. Ripken and R. Wilhelm, Automatic generation of optimizing multipass compilers, in *Proceedings of the IFIP 77 Congress*, pp. 535–540, North-Holland, Amsterdam (1977).
13. U. Kastens, E. Zimmerman and B. Hutt, GAG—a Practical Compiler Generator. *Lecture Notes in Computer Science*, Vol. 141. Springer, Berlin (1982).
14. C. H. A. Koster, *A Compiler Compiler*, Mathematisch Centrum, Amsterdam, Report MR127 (November 1971). Also: Using the CDL compiler compiler. In *Compiler Construction, an Advanced Course* ed. by F. L. Bauer and J. Eickel, *Lecture Notes in Computer Science*, Vol. 21, pp. 366–426, Springer, Berlin (1974).
15. O. Lecarme and G. V. Bochmann, A (truly) usable and portable compiler writing system, in *Proceedings of the IFIP 74 Congress*, pp. 218–221. North-Holland, Amsterdam (1974).
16. B. Lorho, Semantic attributes processing in the system DELTA, in *Methods of algorithmic language implementation* ed. by C. H. A. Koster, *Lecture Notes in Computer Science*, Vol. 47, pp. 21–40, Springer, Berlin (1977).
17. K.-J. Räihä, M. Saarinen, E. Soisalon-Soininen and M. Tienari, *The Compiler Writing System HLP*. Department of Computer Science, University of Helsinki, Report A-1978-2 (March 1978).
18. H. Riis and M. Madsen, *The NEATS System*. Aarhus University, Report DAIMI MD-44 (2nd Edn) (May 1982).
19. A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation and Compiling: Vol. 1: Parsing; Vol. 2: Compiling*, Prentice-Hall, Englewood Cliffs, New Jersey (1972, 1973).
20. P. M. Lewis, D. J. Rosenkrantz and R. E. Stearns, Attributed translations. *J. Computer and System Sciences* **9**, 279–307 (1974).
21. O. L. Madsen, B. B. Kristensen and J. Staunstrup, *Use of Design Criteria for Intermediate Languages*, Aarhus University, Report DAIMI PB-59 (August 1976).
22. O. L. Madsen, On defining semantics by means of extended attribute grammars, in *Semantic-Directed Compiler Generation* ed. by N. D. Jones, *Lecture Notes in Computer Science*, Vol. 94, pp. 259–299, Springer, Berlin (1980).
23. P. Mosses, *SIS, Semantics Implementation System*, Aarhus University, Report DAIMI MD-30 (1979).
24. R. D. Tennent, The denotational semantics of programming languages. *Communications of the ACM* **19**, 437–453 (1976).
25. G. V. Bochmann and P. Ward, *Compiler Writing Systems for Attribute Grammars*, Publication #199, Département d'Informatique, Université de Montréal (July 1975).
26. D. A. Watt, The parsing problem for affix grammars. *Acta Informatica* **8**, 1–20 (1977).
27. M. Jazayeri and K. G. Walter, Alternating semantic evaluators. *Proceedings of the ACM 75 Annual Conference*, 230–234 (1975).
28. J. Engelfriet and G. File, Passes, Sweeps, and Visits. *Lecture Notes in Computer Science*, Vol. 115, pp. 193–207. Springer, Berlin (1981).
29. D. A. Watt, Rule splitting and attribute-directed parsing, in *Semantics-Directed Compiler Generation* ed. by N. D. Jones, *Lecture Notes in Computer Science*, Vol. 94, Springer, Berlin (1980).
30. S. H. Eriksen, B. B. Kristensen and O. L. Madsen, *The BOBS-system*, Aarhus University, Report DAIMI PB-71 (revised version) (1979).
31. L. Paulson, A compiler generator for semantic grammars, PhD Dissertation, Stanford University (December 1981).
32. D. A. Watt, *Modular Language Definitions*. Computing Science Department, University of Glasgow, Report CSC/82/R3 (October 1982).
33. C. A. R. Hoare, Notes on data structuring. In *Structured Programming*, ed. by O.-J. Dahl, E. W. Dijkstra and C. A. R. Hoare, pp. 83–174. Academic Press, London (1972).

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[1] 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_1 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 ith 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

$$(m_1 \cup m_2)[d] \equiv \text{if } m_1[d] \text{ is defined then } m_1[d] \text{ else } m_2[d]$$

For each m_1 and m_2 in M, $m_1 \backslash m_2$ is the map m_1 overridden by m_2; i.e.

$$(m_1 \backslash m_2)[d] \equiv \text{if } m_2[d] \text{ is undefined then } m_1[d] \text{ else } m_2[d]$$

Sequences. If D is a domain, then

$$S = D^*$$

is the domain of sequences of elements of D.

$\langle\,\rangle$ denotes the empty sequence. $\langle d \rangle$ denotes the sequence containing the single component d.

If s is in S and d is in D, then $d \hat{} s$ denotes the sequence obtained by prepending d to s.

A.2 Domain definitions

```
Environment = Name → (declarationdepth:Level;
                      mode:Mode)
Mode        = ( variable(Type) |
                formal(Parameter) |
                procedure(Plan) )
Plan        = Parameter*
Parameter   = ( value(Type)|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.)

⟨name ↑ Name⟩
⟨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.

⟨actual parameter ↓ Environment ↓ Parameter⟩	10
⟨actual parameter list ↓ Environment ↓ Plan⟩	9
⟨adding operator ↑ Operator⟩	16
⟨assignment ↓ Environment⟩	3
⟨block ↓ Level ↓ Environment ↓ Environment⟩	20
⟨compound statement ↓ Environment⟩	5
⟨constant ↑ Type⟩	14
⟨expression ↓ Environment ↑ Type⟩	11
⟨formal parameter ↓ Level ↑ Parameter ↑ Environment⟩	26
⟨formal parameter list ↓ Level ↑ Plan ↑ Environment⟩	25
⟨identifier ↓ Environment ↑ Mode⟩	19
⟨if statement ↓ Environment⟩	7

The distinguished non-terminal is ⟨program⟩.

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, PROCS	:Environment
MODE	:Mode
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⟩ ";" ⟨statement ↓ 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 paramater ↓ ENV ↓ PARM⟩ ","
⟨actual parameter list ↓ ENV ↓ PLAN⟩

⟨actual parameter ↓ ENV ↓ value(TYPE)⟩ : :=
(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⟩

⟨simple expression ↓ ENV ↑ TYPE⟩ : :=
(12a) ⟨term ↓ ENV ↑ TYPE⟩
⟨simple expression ↓ ENV ↑ integer⟩ : :=
(12b) ⟨simple expression ↓ ENV ↑ integer⟩
⟨adding 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) "="
⟨relational operator ↑ unequal⟩ : : =
(15b) "< >"

⟨adding operator ↑ plus⟩ : : =
(16a) "+"
⟨adding operator ↑ minus⟩ : : =
(16b) "−"

⟨where comparable ↓ integer ↓ integer⟩ : : =
(17a) ⟨empty⟩
⟨where comparable ↓ boolean ↓ boolean⟩ : : =
(17b) ⟨empty⟩

(* The non-terminal ⟨where comparable⟩ acts as a predicate, since all its terminal productions are empty; it serves to enforce type compatibility. *)

(* VARIABLES AND IDENTIFIERS *)

⟨variable ↓ ENV ↑ TYPE⟩ : : =
(18a) ⟨identifier ↓ ENV ↑ variable(TYPE)⟩ |
(18b) ⟨identifier ↓ ENV ↑ formal(value(TYPE))⟩ |
(18c) ⟨identifier ↓ ENV ↑ formal(var(TYPE))⟩ |
(18d) ⟨variable ↓ ENV ↑ array(LB, UB, TYPE)⟩
"[" ⟨expression ↓ ENV ↑ integer⟩ "]"

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

⟨identifier ↓ ENV ↑ ENV[NAME].mode⟩ : : =
(19) ⟨name ↑ NAME⟩

(* ⟨identifier⟩ has a synthesized attribute representing its mode, which is determined by looking up the name of the identifier in the "environment". *)

(* DECLARATIONS *)

⟨block ↓ DEPTH ↓ NONLOCALS ↓ FORMALS⟩ : : =
(20) ⟨variable declarations ↓ DEPTH ↑ VARS⟩
⟨procedure declarations ↓ DEPTH
↓ NONLOCALS\(FORMALS ∪ VARS ∪ PROCS)
↑ PROCS⟩
⟨compound statement ↓ ENV
↓ NONLOCALS\(FORMALS ∪ VARS ∪ PROCS)⟩

(* The first attribute of ⟨block⟩ is its depth of nesting. ⟨block⟩ also has two inherited 'environment' attributes, representing non-local identifiers and local formal parameters, respectively. The latter attribute (FORMALS) is disjointly united with the local variable identifiers (VARS) and local procedure identifiers (PROCS) to form the set of local identifiers: FORMALS ∪ VARS ∪ PROCS; this then overrides the non-local identifiers to form the 'environment' inside the block: NONLOCALS\(FORMALS ∪ VARS ∪ PROCS). The use of the disjoint-union operator ∪ ensures that no identifier may be declared more than once in the same block. (20) makes this inner 'environment' apply to the local procedure declarations as well as to the compound statement, allowing each procedure to be called by any procedure declared in the same block; it is this rule which implies a minimum of two passes for attribute evaluation in this EAG. *)

⟨variable declarations ↓ DEPTH ↑ DECL⟩ : : =
(21a) "var"
⟨variable declaration ↓ DEPTH ↑ DECL⟩ ";"

⟨variable declarations ↓ DEPTH ↑ DECLS ∪ DECL⟩ : : =
(21b) ⟨variable declarations ↓ DEPTH ↑ DECLS⟩
⟨variable declaration ↓ DEPTH ↑ DECL⟩ ";"

⟨variable declaration ↓ DEPTH
↑ [NAME → (DEPTH, variable(TYPE))]⟩ : : =
(22) ⟨name ↑ NAME⟩ ":" ⟨type ↑ TYPE⟩

(* PROCEDURES *)

⟨procedure declarations ↓ DEPTH ↓ ENV ↑ [ ]⟩ : : =
(23a) ⟨empty⟩
⟨procedure declarations ↓ DEPTH ↓ ENV ↑ DECLS ∪ DECL⟩ : : =
(23b) ⟨procedure declarations ↓ DEPTH ↓ ENV ↑ DECLS⟩
⟨procedure declaration ↓ DEPTH ↓ ENV ↑ DECL⟩ ";"

⟨procedure declaration ↓ DEPTH ↓ ENV
↑ [NAME → (DEPTH, procedure(PLAN))]⟩ : : =
(24) "procedure" ⟨name ↑ NAME⟩ "("
⟨formal parameter list ↓ DEPTH + 1 ↑ PLAN ↑ FORMALS⟩
")" ";"
⟨block ↓ DEPTH + 1 ↓ ENV ↓ FORMALS⟩

⟨formal parameter list ↓ DEPTH ↑ ⟨PARM⟩ ↑ DECL⟩ : : =
(25a) ⟨formal parameter ↓ DEPTH ↑ PARM ↑ DECL⟩

⟨formal parameter list ↓ DEPTH
↑ PARM ^ PLAN ↑ DECLS ∪ DECL⟩ : : =
(25b) ⟨formal parameter ↓ DEPTH ↑ PARM ↑ DECL⟩ ";"
⟨formal parameter list ↓ DEPTH ↑ PLAN ↑ DECLS⟩

(* The second attribute of ⟨formal parameter list⟩ is a sequence of the modes of the formal parameters, to be used in checking actual parameter lists. Its third attribute is the partial 'environment' established by the formal parameter list. *)

⟨formal parameter ↓ DEPTH ↑ value(TYPE)
↑ [NAME → (DEPTH, formal(value(TYPE)))]⟩ : : =
(26a) ⟨name ↑ NAME⟩ ":" ⟨type ↑ TYPE⟩

⟨formal parameter ↓ DEPTH ↑ var(TYPE)
↑ [NAME → (DEPTH, formal(var(TYPE)))]⟩ : : =
(26b) "var" ⟨name ↑ NAME⟩ ":" ⟨type ↑ TYPE⟩

(* TYPES *)

⟨type ↑ boolean⟩ : : =
(27a) "boolean"
⟨type ↑ integer⟩ : : =
(27b) "integer"
⟨type ↑ array(LB, UB, TYPE)⟩ : : =
(27c) "array" "[" ⟨integer number ↑ LB⟩ ".."
⟨integer number ↑ UB⟩ "]" "of"
⟨type ↑ TYPE⟩

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.

⟨*declare* ↓ Level ↓ Name ↓ Mode⟩
⟨*dyadic* ↓ Operator ↓ Type ↓ Type⟩
⟨*do* ↓ Level⟩
⟨*else* ↓ Level⟩
⟨*fi* ↓ Level⟩
⟨*if* ↓ Level⟩
⟨*index* ↓ Integer ↓ Integer⟩
⟨*name* ↓ Level ↓ Name⟩
⟨*number* ↓ Level⟩
⟨*od* ↓ Level⟩
⟨*procedure* ↓ Level ↓ Name⟩
⟨*store* ↓ Type⟩
⟨*then* ↓ Level⟩
⟨*while* ↓ Level⟩

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

⟨compound statement ↓ Level⟩
⟨if statement ↓ Level⟩
⟨serial ↓ Level⟩
⟨statement ↓ Level⟩
⟨while statement ↓ Level⟩

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-terminals, and in which attributes are merely copied.

(* PROGRAMS *)

⟨program⟩ : : =
(1) *program* ⟨block⟩ *endprogram*

(* STATEMENTS *)

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

⟨assignment⟩ : : =
(3) ⟨variable⟩ ⟨expression⟩ ⟨*store* ↓ TYPE⟩

⟨procedure call⟩ : : =
(4) ⟨actual parameter list⟩ ⟨identifier⟩ *call*

⟨if statement ↓ LEVEL⟩ : : =
(7) ⟨*if* ↓ LEVEL⟩ ⟨expression⟩
⟨*then* ↓ LEVEL⟩ ⟨statement ↓ LEVEL + 1⟩
⟨*else* ↓ LEVEL⟩ ⟨statement ↓ LEVEL + 1⟩
⟨*fi* ↓ LEVEL⟩

⟨while statement ↓ LEVEL⟩ : : =
(8) ⟨*while* ↓ LEVEL⟩ ⟨expression⟩
⟨*do* ↓ LEVEL⟩ ⟨statement ↓ LEVEL + 1⟩
⟨*od* ↓ LEVEL⟩

(* ACTUAL PARAMETERS *)

⟨actual parameter⟩ : : =
(10a) ⟨expression⟩ *valueparameter*
⟨actual parameter⟩ : : =
(10b) ⟨variable⟩ *varparameter*

(* EXPRESSIONS *)

⟨expression⟩ : : =
(11b) ⟨simple expression⟩ ⟨simple expression⟩
⟨*dyadic* ↓ OP ↓ TYPE1 ↓ TYPE2⟩

⟨simple expression⟩ : : =
(12b) ⟨simple expression⟩ ⟨term⟩
⟨*dyadic* ↓ OP ↓ integer ↓ integer⟩

⟨constant⟩ : : =
(14a) *false*|
(14b) *true*
⟨constant⟩ : : =
(14c) ⟨*number* ↓ VALUE⟩

(* VARIABLES AND IDENTIFIERS *)

⟨variable⟩ : : =
(18d) ⟨variable⟩ ⟨expression⟩ ⟨*index* ↓ LB ↓ UB⟩

⟨identifier⟩ : : =
(19) ⟨*name* ↓ ENV[NAME].declarationdepth ↓ NAME⟩

(* DECLARATIONS *)

⟨block⟩ : : =
(20) ⟨variable declarations⟩
⟨procedure declarations⟩
⟨compound statement ↓ 0⟩

⟨variable declaration⟩ : : =
(22) ⟨*declare* ↓ DEPTH ↓ NAME ↓ variable(TYPE)⟩

⟨procedure declaration⟩ : : =
(24) ⟨*procedure* ↓ DEPTH ↓ NAME⟩
⟨formal parameter list⟩ ⟨block⟩
endprocedure

⟨formal parameter⟩ : : =
(26a) ⟨*declare* ↓ DEPTH ↓ NAME ↓ formal(value(TYPE))⟩
⟨formal parameter⟩ : : =
(26b) ⟨*declare* ↓ DEPTH ↓ NAME ↓ formal(var(TYPE))⟩