ISSN 0105-8517

EXTENDED ATTRIBUTE GRAMMARS

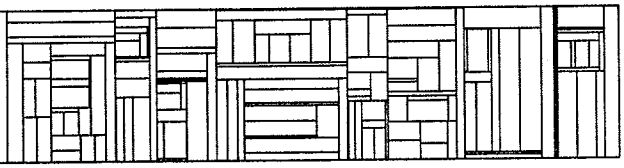
by

David Anthony Watt *
and
Ole Lehrmann Madsen

DAIMI PB-105
November 1979

* Computing Science Department, University of Glasgow,
Glasgow G12 8QQ, Scotland.

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55



EXTENDED ATTRIBUTE GRAMMARS

Abstract

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 attribute grammars and affix grammars.

Key Words and Phrases: attribute grammar, affix grammar, extended attribute grammar, extended attributed translation grammar, compiler writing system.

CR Categories: 4.12, 5.23

Revised version of: D.A. Watt, O.L. Madsen, Extended Attribute Grammars, University of Glasgow, Report no. 10, July 1977.

1. Introduction

This paper is concerned with the formalization of the syntax and semantics of programming languages. The primary aims of formalization are 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 [28] contained hidden semantic irregularities which were revealed only by formalization of its semantics [8]. 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 [26]. 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 recent survey article [19] 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 [16, 23].

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. Section 6 is a brief description 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 the reader to judge for himself the suitability of the 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 <empty>. Terminal symbols without attributes are enclosed in quotes.

Throughout this section we use the following running example: assignments in an ALGOL68-like language, in which 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 [10], and affix grammars independently by Koster [11]. 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 refer to either attribute grammars or affix grammars.

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 S and a phrase p derived from S. Each inherited attribute of S is supposed to convey information about the context of p, and will be prefixed by a downward arrow (↓). Each synthesized attribute of S is supposed to convey information about the phrase p itself in the given context, and will be prefixed by an upward arrow (↑).

In our example, each of the nonterminals <assignment>, <identifier> and <expression> will have an inherited attribute representing its "environment" (these attributes are inherited since they represent 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 "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:

```
<v↓...↑...>  ::=  <v1↓...↑...>  .....  <vm↓...↑...>
   def app            app def              app def
```

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 [10], in which the evaluation rules may use attributes from any positions. As [2] points out, however, the restriction effectively excludes only grammars containing circularities. See also Section 5.1.)

In practice, many evaluation rules turn out to be simple copies; following [27], we shall 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, Figure 1 shows a fragment of a syntax tree, based on our example.

An attribute grammar has been used to define the programming language Simula [27]. Relative to van Wijngaarden grammars [22], 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 a language definition by an AG is that it is unmistakeably algorithmic.

AGs are well suited to compiler construction, and have been exploited in compiler writing systems [6, 12, 13, 15]. 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, and like van Wijngaarden grammars [22] to be generative rather than algorithmic in nature.

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 both become unnecessary. The attribute variables become somewhat akin to the "metanotions" of a van Wijngaarden grammar.

Our example could be 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 and affix 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, and then evaluate all the attribute expressions.

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

<identifier ↓ {x->ref(int),y->bool} ↑ ref(int)> ::=

<name ↑ x>

This production rule may be applied at some node of a syntax tree (just as in Figure 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 Figure 1 can be filled in by substituting {x->ref(int),y->bool} for ENV and int for MODE in rule (1), giving the production rule

```
<assignment ↓ {x->ref(int),y->bool}>  ::=
      <identifier ↓ {x->ref(int),y->bool} ↑ ref(int)>  ":="
         <expression ↓ {x->ref(int),y->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.

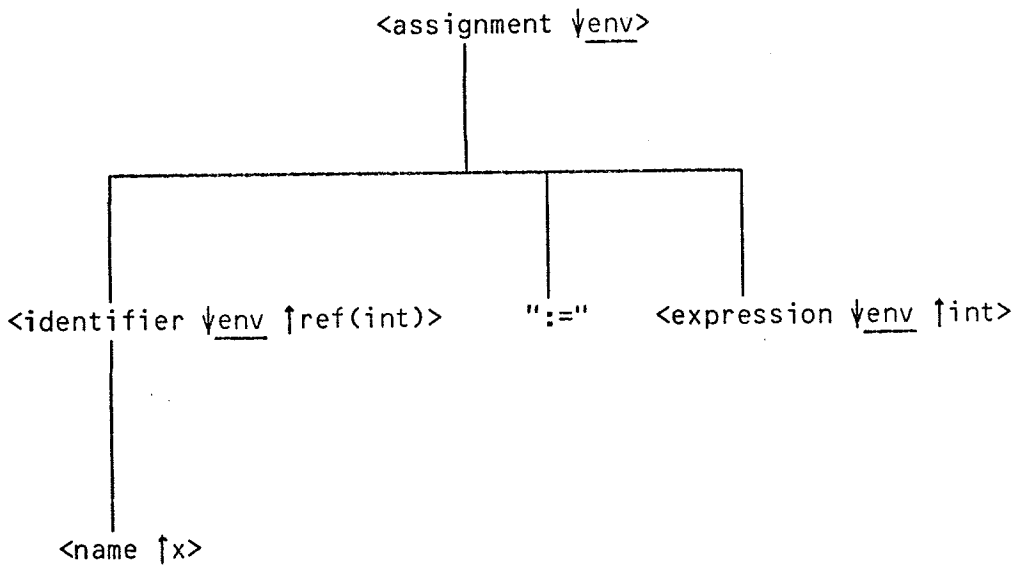


Figure 1. Fragment of a syntax tree in an AG (or EAG).

env stands for the attribute {x->ref(int),y->bool}.

Broken lines leading to each attribute indicate which other attributes it depends upon.

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.

$D = (D1, D2, \ldots, f1, f2, \ldots)$ 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 nonterminal 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 nonterminal 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, ..., e_m are attribute expressions and f is an appropriate (partial) function chosen from D.

In the examples, we shall use not only functional notation for attribute expressions but also infix operators, etc., where convenient.

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

$$\langle v \Updownarrow a_1 \ldots \Updownarrow a_p \rangle$$

is an attributed symbol. In particular, it is an attributed nonterminal (terminal) if v is a nonterminal (terminal). Each $\Updownarrow$ 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

$$\langle v \Updownarrow e_1 \ldots \Updownarrow e_p \rangle$$

is an attributed symbol form.

R is a finite set of production rule forms (or simply rules), each of the form

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

where $m \geq 0$, and $F, F_1, \ldots, F_m$ are attributed symbol forms, F being nonterminal.

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

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

where $m \geq 0$, and A, A_1, ..., A_m are attributed symbols, A being an attributed nonterminal.

A direct production of an attributed nonterminal 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 nonterminal 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 nonterminal 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 [26].

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) [1] can be enhanced with attributes and thus express context-sensitivities of both an input grammar and an output grammar. The attributed translation grammars of [14] 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. (The analysis phase includes lexical analysis, context-free syntax analysis, context-sensitive syntax analysis, and translation into some intermediate language.) A major example of this can be found in [17].

Consider the while statement of Appendix A. Suppose that we wish to specify its translation into some intermediate form which preserves its structure. In an SDTS this could be specified by the following rule:

<while statement> ::=

"while" <expression> "do" <statement> ⟹

while <expression> do <statement> od

The notation follows that given in Section 2, with the addition that ⟹ is used to separate the output part from the input part of each rule. Underlined symbols are terminals in the output alphabet.

Suppose now that we wish to link while, do and od by the nesting depth of the <while statement>. This can be done by giving each of the nonterminals <statement> and <while statement>, and each of the output symbols, an inherited attribute which is its level of nesting.

Generalizing the above SDTS rule with attributes, we obtain:

<while statement ↓ ENV ↓ LEVEL> ::=
"while" <expression ↓ ENV ↑ boolean>
"do" <statement ↓ ENV ↓ LEVEL+1> ⟹
<while ↓ LEVEL> <expression>
<do ↓ LEVEL> <statement> <od ↓ LEVEL>

In the interests of modularity, however, we prefer to split each EATG rule into an input rule (an ordinary EAG rule) and an output rule, in order to separate the attributes defining context-sensitivities of the input grammar from those defining context-sensitivities of the output grammar. Our EATG rule will therefore be expressed as follows:

Input rule: <while statement ↓ ENV> ::=
"while" <expression ↓ ENV ↑ boolean>
"do" <statement ↓ ENV>

Output rule: <while statement ↓ LEVEL> ::=
<while ↓ LEVEL> <expression>
<do ↓ LEVEL> <statement ↓ LEVEL+1>
<od ↓ LEVEL>

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 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.

In generalized SDTSs [11], it is possible to associate with each nonterminal a number of translation elements, strings which may be

constructed from translation elements of descendants in the derivation tree. Aho and Ullman also consider generalized SDTSs where the translation elements are not string-valued. In this case there is no difference between a translation element and a synthesized attribute. Furthermore, they consider inherited translations. We think, however, that one should use translation elements only when it is to be stressed that a translation is actually performed, and use attributes for moving other information around.

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 [18]. An example of this is the language LAMB of SIS [20], which is a compiler generator based upon denotational semantics [21]; 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 [18].

To demonstrate the advantages of EATGs we conclude this section by showing how example 9.19 of [1] 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 nonterminals of the output EAG have two attributes each: an inherited register-valued attribute which specifies where the corresponding sub-expression 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 [1].

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 have taken the liberty of adding a nonterminal to the output EAG

which is not present in the input EAG. This should cause no conceptual difficulty.

Rules of input EAG

(1) <prog> ::= <expr ↓{}>

(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>

Rules of output EAG

(1) <prog> ::= <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

5. Parsing and attribute evaluation

5.1. Parsing and attribute evaluation with AGs

Since AGs are a straightforward extension of CFGs, a corresponding extension of context-free parsing techniques to handle AGs is quite feasible. The only new problem concerns the order of evaluation of the attributes.

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 [10]. We must restrict our attention to grammars containing no circularities.

Consider an AG rule of the following form, stripped of its constraints and evaluation rules:

$$\langle v\downarrow\ldots\uparrow\ldots\rangle \;::=\; \langle v_1\downarrow\ldots\uparrow\ldots\rangle \;\ldots\ldots\ldots\; \langle v_m\downarrow\ldots\uparrow\ldots\rangle$$

An AG is L-attributed if and only if, in every such rule, and for every $i=1,\ldots,m$, no inherited attribute of v_i depends on a synthesized attribute of any of $v_i, \ldots, v_m$.

In an L-attributed AG, all the attributes can be evaluated in a single left-to-right pass over the syntax tree of a sentence [2, 14]. For each v_i, first its inherited attributes are evaluated (using the inherited attributes of v and the synthesized attributes of $v_1, \ldots, v_{i-1}$), then the subtree under v_i is traversed, resulting in the evaluation of the synthesized attributes of v_i. Finally any constraints are checked, and the

synthesized attributes of $\underline{v}$ are evaluated (using the inherited attributes of $\underline{v}$ and the synthesized attributes of $\underline{v}_1, \ldots, \underline{v}_m$), thus completing the traverse of the subtree under $\underline{v}$.

If the CFG underlying the L-attributed AG is deterministic, the attributes can in fact be evaluated while the sentence is being parsed, whether or not a syntax tree is constructed. A variety of one-pass parsing methods for L-attributed AGs have been proposed or implemented; these methods include top-down [3, 11, 12], bounded-context [4], precedence [13], and LR [24].

Unfortunately, many programming languages cannot be defined naturally by L-attributed AGs. (For example, in most programming languages it is permissible for a reference to an identifier or label textually to precede its declaration or definition.) Fortunately, Bochmann has shown that for a large class of AGs, all the attributes can be evaluated in a fixed number of left-to-right passes over a syntax tree, the number of passes being easily computed from the grammar [2]. The first of these passes may coincide with the parsing pass. Typical programming languages require 1, 2 or 3 passes.

A still more general solution is described in [15]. Lorho's system DELTA delays determining the order of evaluation of attributes until after parsing each individual program (whereas in Bochmann's system the order of evaluation is determined by the constructor). DELTA accepts any AG which contains no circularities.

5.2. Extension to extended attribute grammars

The suitability of AGs for parser construction having been established, the simplest way to establish the same property for EAGs is to show how, and in

what circumstances, an EAG may 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 an 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 compile-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 X Integer X Type -> Type

has a partial inverse function

$array^{-1}$: Type -> Integer X Integer X Type

$array^{-1}(\underline{T}) = $ if (exist $\underline{L},\underline{U},\underline{T}'$)($\underline{T}$=array($\underline{L},\underline{U},\underline{T}'$))
then ($\underline{L},\underline{U},\underline{T}'$)
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 and discriminated unions.

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 <- 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 $x=x_1=\ldots=x_n$'.

(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, ..., 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 parsing technique for AGs can be adapted to well-formed EAGs as well. For example, we can define an EAG to be L-attributed if and only if it is well-formed and the above transformations convert it into an AG which is L-attributed. The EAG in Appendix A is not L-attributed, but if we apply Bochmann's analysis we may deduce that all its attributes can be evaluated in two passes.

5.3. Rule splitting

[2] and [24] mention the possibility of making the attributes actually influence the behaviour of the parser, rather than just being evaluated during or after context-free parsing or signalling a context-sensitive error if a constraint is not satisfied. A typical example of this possibility is rule-group (10) from Appendix A:

<actual parameter ↓ ENV ↓ value(TYPE)> ::=

(10a) <expression ↓ ENV ↑ TYPE>

<actual parameter ↓ ENV ↓ var(TYPE)> ::=

(10b) <variable ↓ ENV ↑ TYPE>

Here the underlying CFG is actually ambiguous, but the EAG is not, since no production rule generated from (10a) can ever have the same left-side as one generated from (10b). An inspection of the second inherited attribute of <actual parameter>, if known at parse-time, would allow one of the alternative right-sides to be rejected immediately.

In general, it is undecideable whether an ambiguity (or an LL or LR conflict) is resolved by rule-splitting. In particular cases involving attributes whose domains are discriminated unions, however, it is feasible automatically to detect and exploit rule-splitting.

To illustrate how rule splitting might work, here is a fragment of a recursive-descent parser, constructed from rule-group (10), in which inherited attributes are transcribed as value-parameters and inherited attributes are transcribed as result-parameters:

```
procedure actualparameter (ENV   : Environment;
                           PTYPE : Type        );
   var TYPE : Type;
   begin
   if PTYPE is of the form value(T) then
      begin
      expression (ENV, TYPE);
      if TYPE ≠ value^-1(PTYPE) then context-sensitive error
      end
   else
      begin
      variable (ENV, TYPE);
      if TYPE ≠ var^-1(PTYPE) then context-sensitive error
      end
   end;
```

```
procedure expression (    ENV  : Environment;
                      var TYPE : Type        );
   ..........
   ..........

procedure variable (    ENV  : Environment;
                    var TYPE : Type        );
   ..........
   ..........
```

The implementation of rule splitting in an LR parser is discussed in [24].

In the context of the complete grammar of Appendix A, unfortunately, this rule splitting cannot be exploited since Bochmann's analysis shows that the second attribute of <actual parameter> will be one of those evaluated during the second pass, i.e. it will not be available during the parsing pass. This is a genuine implementation problem, however, not a weakness of the formalism itself. Indeed, the possibility of detecting the problem automatically may be regarded as a triumph for the formalism.

The technique of rule splitting is rather useful in ad-hoc compilers where the source language is most naturally described by an ambiguous CFG (as in our example). As we have seen, EAGs allow an effective formalization of rule splitting. Proper techniques for exploiting this will usefully enhance the power of automated compiling techniques; however, further research is needed to develop such techniques.

6. The Aarhus compiler writing system

An experimental compiler writing system, NEATS, has been implemented at Aarhus by Poul Jespersen, Michael Madsen and Hanne Riis [9]. 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 [5]. 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 [9] and [18] 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 more practical 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 such as L-attributed AGs.

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 rule-splitting;

(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.

The present experiments are very promising, and we acknowledge the work of Jespersen, Madsen and Riis in producing this system.

7. 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 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:

- the explicit attribute structure and the distinction between inherited and synthesized attributes;
- the visibility of the underlying context-free syntax;
- generative definition of languages (like context-free and van Wijngaarden grammars), rather than algorithmic definition (like AGs);
- the implicit and concise specification of context-sensitivities by means of attribute expressions in applied and defining positions;
- 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 [17] and [26].

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 [22] or extended affix grammars [23], but this leads to some artificiality; compare, for example, rule (19) in Appendix A with the corresponding syntax in [22]. 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 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, e.g. [6].

Appendix A. A complete example of an extended attribute grammar

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 [28] containing the following features:-

- boolean, integer, and array data types;
- variable declarations;
- procedure declarations, with value- and variable-parameters;
- assignments, procedure calls, compound-, if- and while-statements;
- expressions involving integer and relational operators;
- 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 [7].

Cartesian products

If T_1, ..., T_n are domains and g_1, ..., g_n are distinct names, then

P = (g_1:T_1; .....; g_n:T_n)

is a Cartesian product with field selectors g_1, ..., g_n.

For every a_1 in T_1, ..., and every a_n in T_n, (a_1,...,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,...,n, $p.g_i$ is in T_i, and denotes the ith field of p.

Discriminated unions

If T_1, ..., T_n are domains (or Cartesian products of domains) and g_1, ..., g_n are distinct names, then

$$U = (g_1(T_1) \mid ..... \mid g_n(T_n))$$

is a discriminated union with selectors g_1, ..., g_n. If any T_i is void, then we abbreviate $g_i(T_i)$ to g_i.

For every $i=1,...,n$, 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, ..., d_n are distinct elements of D and r_1, ..., r_n are in R, then $\{d_1 \rightarrow r_1, ..., d_n \rightarrow r_n\}$ is in M, and denotes a map defined at points d_1, ..., d_n and nowhere else.

For each m_1 and m_2 in M, $m_1 U m_2$ is the disjoint union of m_1 and m_2: $m_1 U m_2$ is undefined if, for any d in D, both $m_1[d]$ and $m_2[d]$ are defined; otherwise

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

For each $\underline{m}_1$ and $\underline{m}_2$ in $\underline{M}$, $\underline{m}_1 \backslash \underline{m}_2$ is the map $\underline{m}_1$ overridden by $\underline{m}_2$; i.e.

$(\underline{m}_1 \backslash \underline{m}_2)[\underline{d}]$ = if $\underline{m}_2[\underline{d}]$ is undefined
then $\underline{m}_1[\underline{d}]$
else $\underline{m}_2[\underline{d}]$

Sequences

If $\underline{D}$ is a domain, then

$\underline{S} = \underline{D}^*$

is the domain of sequences of elements of $\underline{D}$.

[] denotes the empty sequence.

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

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 = the domain of integers

Name = the domain of character strings denoting names

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 terminals are written enclosed in quotes ("...").

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

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

procedure call	↓ Environment	4
procedure declaration	↓ Level ↓ Environment ↑ Environment	24
procedure declarations	↓ Level ↓ Environment ↑ Environment	23
program		1
relational operator	↑ Operator	15
serial	↓ Environment	6
simple expression	↓ Environment ↑ Type	12
statement	↓ Environment	2
term	↓ Environment ↑ Type	13
type	↑ Type	27
variable	↓ Environment ↑ Type	18
variable declaration	↓ Level ↑ Environment	22
variable declarations	↓ Level ↑ Environment	21
while statement	↓ Environment	8
where comparable	↓ Type ↓ Type	17

The distinguished nonterminal 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 nonterminals 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 parameter ↓ 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 *)

<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>

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

<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 nonterminal <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\(FORMALSUVARSUPROCS) ↑ PROCS>
<compound statement ↓ ENV
↓ NONLOCALS\(FORMALSUVARSUPROCS)>

(* The first attribute of <block> is its depth of nesting. <block> also has two inherited "environment" attributes, representing nonlocal identifiers and local formal-parameter identifiers 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 (FORMALSUVARSUPROCS), which then overrides the nonlocal identifiers to form the "environment" inside the block (NONLOCALS\(FORMALSUVARSUPROCS)). The use of the disjoint-union operator U 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 ↑ DECLSUDECL>  ::=
(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 ↑ DECLSUDECL>  ::=
(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 ↑ DECLSUDECL> ::=
(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:

- expressions are in postfix form;
- each identifier is made unique by attaching to it the depth of nesting of the block where it was declared;
- 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 nonterminal symbols, showing the type and domain of each attribute-position used in the output grammar. Nonterminals which have no such attribute-positions 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 nonterminals, and in which attributes are merely copied.

(* PROGRAMS *)

<program> ::=

(1) program <block> endprogram

(* STATEMENTS *)

<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>

(* 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). *)

(* 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))>

References

1. Aho, A.V., Ullman, J.D. The Theory of Parsing, Translation and Compiling. Vol. 1: Parsing, 1972. Vol. 2: Compiling, 1973. Englewood Cliffs (N.J.): Prentice Hall

2. Bochmann, G.V.: Semantic evaluation from left to right. Comm. ACM 19, 55-62 (1976)

3. Bochmann, G.V., Ward, P.: Compiler writing systems for attribute grammars. Département d'Informatique, Université de Montréal, Publication #199, July 1975

4. Crowe, D.: Constructing parsers for affix grammars. Comm. ACM 15, 728-734 (1972)

5. Eriksen, S.H., Kristensen, B.B., Madsen, O.L.: The BOBS-system. Aarhus University, Report DAIMI PB-71 (revised version), 1979

6. Ganzinger, H., Ripken, K., Wilhelm, R.: Aotomatic generation of optimizing multipass compilers. In: Proc. IFIP 77 Congress, pp. 535-540. Amsterdam: North-Holland 1977

7. Hoare, C.A.R.: Notes on data structuring. In: Structured Programming (O.-J. Dahl, E.W. Dijkstra, C.A.R. Hoare), pp. 83-174. London-New York: Academic Press 1972

8. Hoare, C.A.R., Wirth, N.: An axiomatic definition of the programming language Pascal. Acta Informatica 2, 335-355 (1973)

9. Jespersen, P., Madsen, M., Riis, H.: NEATS, New Extended Attribute Translation System. Aarhus University, 1979

10. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory 2, 127-145 (1968)

11. Koster, C.H.A.: Affix grammars. In: ALGOL 68 Implementation (J.E. Peck, ed.), pp. 95-109. Amsterdam: North-Holland 1971

12. Koster, C.H.A. A compiler compiler. Mathematisch Centrum, Amsterdam, Report MR127 (November 1971). Also: Using the CDL compiler compiler. In: Compiler Construction, an Advanced Course (F.L. Bauer, J. Eickel, eds.), pp. 366-426. Lecture Notes in Computer Science, Vol. 21. Berlin-Heidelberg-New York: Springer 1974

13. Lecarme, O., Bochmann, G.V.: A (truly) usable and portable compiler writing system. In: Proc. IFIP 74 Congress, pp. 218-221. Amsterdam: North-Holland 1974

14. Lewis, P.M., Rosenkrantz, D.J., Stearns, R.E.: Attributed translations. J. Computer and System Sciences 9, 279-307 (1974)

15. Lorho, B.: Semantic attributes processing in the system DELTA. In: Methods of algorithmic language implementation (C.H.A. Koster, ed.), pp. 21-40. Lecture Notes in Computer Science, Vol. 47. Berlin-Heidelberg-New York: Springer 1977

16. Madsen, O.L.: On the use of attribute grammars in a practical translator writing system. Aarhus University, Master thesis, July 1975

17. Madsen, O.L., Kristensen, B.B., Staunstrup, J.: Use of design criteria for intermediate languages. Aarhus University, Report DAIMI PB-59, August 1976

18. Madsen, O.L.: On defining semantics by means of extended attribute grammars. Aarhus University, Report DAIMI IR-14, September 1979

19. Marcotty, M., Ledgard, H.F., Bochmann, G.V.: A sampler of formal definitions. Computing Surveys 8, 191-276 (1976)

20. Mosses, P.: SIS, Semantics Implementation System. Aarhus University,

Report DAIMI MD-30, 1979

21. Tennent, R.D.: The denotational semantics of programming languages. Comm. ACM 19, 437-453 (1976)

22. Van Wijngaarden, A., Mailloux, B., Peck, J.E.L., Koster, C.H.A., Sintzoff, M., Lindsey, C.H., Meertens, L.G.L.T., Fisker, R.G.: Revised Report on the Algorithmic Language ALGOL 68. Acta Informatica 5, 1-236 (1975); Berlin-Heidelberg-New York: Springer 1976

23. Watt, D.A.: Analysis-oriented two-level grammars. University of Glasgow, Ph.D. thesis, January 1974

24. Watt, D.A.: LR parsing of affix grammars. Computing Science Department, University of Glasgow, Report 7, August 1974

25. Watt, D.A.: The parsing problem for affix grammars. Acta Informatica 8, 1-20 (1977)

26. Watt, D.A.: An extended attribute grammar for Pascal. SIGPLAN Notices 14, 2, 60-74 (1979)

27. Wilner, W.T.: Declarative semantic definition. Stanford University, Report STAN-CS-233-71, 1971

28. Wirth, N.: The programming language Pascal. Acta Informatica 1, 35-63 (1971)