ISSN 0105-8517

ON DEFINING SEMANTICS
BY MEANS OF EXTENDED ATTRIBUTE GRAMMARS

by

Ole Lehrmann Madsen

DAIMI PB-109
January 1980

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

Abstract.

Attribute grammars have traditionally been used to define the context-sensitive syntax of programming languages and translations of these into intermediate forms or code for (hypothetical) machines. The purpose of this paper is to demonstrate different ways of using attribute grammars to define different kinds of semantics. The possibilities for defining predicate transformers, denotational semantics, and operational semantics are treated. The approach to opperational semantics consists of giving a set of attribute grammar rules that specifies the possible transformations upon a given program.

A major motivation for this work is a desire to construct a general translator writing system where different kinds of (practical or experimental) translators/compilers may be generated based on different kinds of semantic specifications.

A generative version of attribute grammars called extended attribute grammars are used. A proposal for making it possible to define the domains of an attribute grammar within the formalism is given.

Finally an efficient evaluator that works for all attribute grammars (including some circular ones) are described. This evaluator constructs during a left-to-right scan of a linear representation of the parse tree (a right-parse) a directed (acyclic) graph that represents the values of the attributes at the root of the parse tree. The parse tree itself need not to be constructed. During a (recursive) scan of this graph the attribute values may be evaluated.

(Revised version of DAIMI IR-14, September 1979)

Contents.

1. Introduction.

2. Basic terminology.

3. Defining verification generators.

4. Denotational semantics and attribute grammars.

4.1 A redefinition of attribute grammars

4.2 Relation to denotational semantics.

4.2.1 A simple reformulation.

4.2.2 A reformulation using a circular attribute grammar.

4.2.3 Other reformulations.

4.3 Conclusion.

5. Defining operational semantics by attribute grammars.

5.1 Specifying abstract data types.

5.2 Specifying semantics.

5.3 Nondeterminism and concurrency.

5.4 Conclusion

6. Pure and multi-level attribute grammars.

7. A general fast attribute grammar evaluator.

7.1 The DAG-evaluator.

7.2 Examples.

7.3 Evaluation.

8. Conclusion.

9. References.

1. Introduction.

Knuth has introduced attribute grammars (AGs) as a tool to define the semantics of context-free languages. The use of AGs in connection with programming language definitions has mostly been to define the context-sensitive syntax of the language and to define a translation into code for a hypothetical machine. The semantics of a program is then defined by the interpreter for this machine ([Knuth 68], [Wilner 72], [Marcotty et al 76]). This is a rather compiler oriented approach to semantics but it has among others resulted in a number of translator writing systems based upon AGs.

Defining semantics this way is useful for an implementer of a language but is less useful for a user or designer of a language. It is often assumed that this is the only way that AGs can be used to define semantics and for this reason AGs are not really considered as an acceptable way of defining semantics. AGs are viewed as a compiler writing tool. One of the reasons for this may be that AGs were not introduced as a complete formalism in the sense that it is not part of the formalism how to define the domains of the attributes.

We would like to point out that we find AGs to be a very useable tool for defining the context-sensitive syntax of a programming language. See e.g. the definition of Pascal in [Watt 78].

In the original paper by Knuth it was stated that any semantics for a language which can be defined as a function of the set of parse trees can be defined by an AG. The purpose of this paper is to demonstrate different ways and techniques for using AGs to define different kinds of semantics.

The motivation for this work comes from an interest in practical translator writing systems (TWSs). Having a TWS intended for implementing (parts of) practical compilers it would be desirable if the same TWS could also be used to make an experimental implementation based on a formal semantics of the language. This will ease experiments with definitions of new languages.

If several different kinds of complementary formal semantics can be used in the same TWS then one may start with a rather human oriented semantics as the basis for an initial (and inefficient) implementation. One may then successively develop more implementation oriented semantics that give more efficient implementations, and if still too inefficient it may be used as an 'implementation guide' for a hand written implementation.

In this paper we shall investigate the possibilities of defining predicate transformers, denotational semantics, and operational semantics. The approach to operational semantics is to specify an AG that defines the possible transformations upon a representation of the program. The operational approach is used (1) to specify abstract data types, (2) to specify semantics of programming languages, and (3) as a model for defining nondeterministic and concurrent computations.

As mentioned we are interested in exploring the possible uses of an AG based TWS, so we are concerned with what is possible to define by means of AGs, i.e. expressibility . We are however just as well interested in the ways things are done with respect to intuition, readability, efficiency, etc. This is often a matter of personal opinion. In connection with this there is no single way of using AGs. The attributes may be used in different ways just as it may be natural to use a translation grammar instead of having the translation as a synthesised attribute.

The power (or expressibility) of AGs is dependent upon the actual domains available. We propose that the domains shall be defined by other AGs. In this way AGs become multi-level instead of two-level. At the bottom we define pure AGs which have a 'built in' set of domain types. We have chosen tree languages as this basic domain. Pure AGs are then quite similar to vW-grammars ([van Wijngaarden et al. 75]), and extended affix grammars [Watt 74a], where the basic domains are context-free (string) languages.

We use a version of AGs called extended attribute grammars (EAGs) ([Watt & Madsen 77]). EAGs are generative in the same sense as affix-grammars ([Koster 70]) and vW-grammars, whilst retaining the essential advantages of AGs. In our opinion EAGs are better suited for analysis and lead to more readable and natural descriptions.

The notions of pure AG and multi-level AG are further refinements of EAGs.

In [Watt & Madsen 77] it is also mentioned that the idea of EAGs can be carried over to translation grammars. We shall also make use of these extended attributed translation grammars (EATGs). EATGs may be a useful tool to define programming languages where the (context-sensitive) syntax and one or more semantics are defined by an integrated formalism. We imagine that the input grammar defines the syntax and a particular semantics is defined by a particular output grammar.

It is still an open problem how to make a general and efficient implementation of AGs without enforcing strange requirements upon the dependencies allowed between attributes. Such requirements are often introduced in order to have a well defined (and efficient) order of evaluation of the attributes. We give an evaluator for AGs where the order of evaluation of attributes is no problem. The evaluator is general as is accepts all AGs and it is 'very fast'. However it still needs space proportional to the size of the parse tree.

We hope to achieve the following:

- demonstrate that AGs are a very powerful meta language for defining different kinds of semantics,
- demonstrate that a TWS based upon AGs can be used for many purposes,
- contribute to a better understanding of AGs and to the theory of AGs and show how AGs may be turned into a complete formalism,
- contribute to a unification of different formal semantics. We do not claim that we add anything new to these methods,
- present a general and fast evaluator for AGs.

The rest of the paper is organised as follows:

Section 2 reviews the basic terminology being used. The sections 3-7 fall into three parts: Part I (section 3-5) is about semantic definitions. Verification generators are treated in section 3, denotational semantics in section 4 and operational semantics in section 5. Part II (section 6) is a concluding section about the AG formalisms. Part III (section 7) describes a general evaluator for AGs called the DAG-evaluator. The paper is concluded in section 8.

Acknowledgement. I am grateful to Brian Mayoh and Mogens Nielsen for their advice during the preparation of this paper. The following people have also made useful contributions by participating in discussions or by giving comments: Poul Jespersen, Kurt Jensen, Niel Jones, Michael Madsen, Robin Milner, Peter Mosses, Hanne Riis, Erik Meineche Schmidt, Sven Skyum, and David Watt.

2. Basic Terminology.

We use a generative version of AGs called extended attribute grammars (EAGs) ([Watt & Madsen 77]) with the modification that we allow the start symbol to have synthesised attributes. The definition of EAG is repeated below. For a more expository exposition, the reader is referred to [Watt & Madsen 77] or [Madsen 79].

Defintion 2.1.

An extended attribute grammar is a 5-tuple

$$G = (D, V, Z, B, R)$$

whose elements are defined in the following paragraphs.

$D = (D_1,D_2,\ldots,f_1,f_2,\ldots)$ is an algebraic structure with domains D_1, D_2, ..., and (partial) functions f_1,f_2, ... 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 synthesised.

Z, a member of V_N, is the start-symbol of G.

The start-symbol Z and the terminal symbols have only synthesised 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 make use of infix operators where convenient.

Let $v \in V$, 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

$$<v\updownarrow a_1 \ldots \updownarrow a_p>$$

is an *attributed symbol* corresponding to v. In particular, it is an attributed nonterminal (terminal) if v is a nonterminal (terminal). Each $\updownarrow$ stands for either $\downarrow$ of $\uparrow$, prefixing an inherited or synthesised attribute-position as the case may be.

AV_N (AV_T) stands for the set of attributed nonterminals (terminals) , $AV = AV_N \cup AV_T$, and AZ is the set of attributed nonterminals corresponding to the start-symbol Z.

If $e_1, \ldots, e_p$ are attribute expressions whose ranges are included in $D_1, \ldots, D_p$, respectively, then

$$<v\updownarrow e_1 \ldots \updownarrow e_p>$$

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 a nonterminal.

A production rule form defines a set of *production rules* in the following way:

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 $A, A_1, \ldots, A_m$ are attributed symbols.

The relation => is defined as follows:

Let $\underline{a}, \underline{g} \in AV^*$, $A \in AV_N$, and let $A ::= \underline{b}$ be a production rule,

then $\underline{a}\ A\ \underline{g} \Rightarrow \underline{a}\ \underline{b}\ \underline{g}$.

$\Rightarrow^*$ and $\Rightarrow^+$ are define in the usual way.

The language generated by G, L(G) a subset of AV_T^*, is defined as

$$L(G) = \{\, w \mid S \Rightarrow^* w \text{ and } S \in AZ \,\}$$

Let $D_1, \ldots, D_p$ be the attribute domains of Z. The translation defined by G, T(G) a subset of $AV_T^* X(D_1 X \ldots X D_p)$, is defined as

$$T(G) = \{\, (w,m) \mid m=(a_1,\ldots,a_p),\ \langle Z{\uparrow}a_1 \ldots {\uparrow}a_p\rangle \Rightarrow^* w \}$$

If $(w,m) \in T(G)$ then m is a *meaning* of w.

The relation => defines in the usual way an *attributed parse tree*. An attributed parse tree defines in a unique way a corresponding parse tree from its underlying CFG.

One may distinguish between the following three kinds of ambiguity/unambiguity of an EAG:

(1) G is *semantically ambiguous* if there are meanings m1,m2, $m1 \neq m2$ and $(w,m1) \in T(G)$ and $(w,m2) \in T(G)$.

(2) G is *structurally ambiguous* if there is a w $\in$ L(G) and w is the frontier of two or more distinct attributed parse trees.

(3) G is *syntactically ambiguous* if the underlying CFG of G is ambiguous.

Observe that the distinction between inherited and synthesised attribute makes no difference to the language and translation defined by the EAG. The distinction is traditional, may improve the readability and is important when considering implementations of EAGs. This is also the case for the following definitions.

Inherited attribute-positions on the left-side and synthesised attribute-positions on the right-side of a rule are called *defining positions* . Synthesised attribute-positions on the left-side and inherited attribute-positions on the right-side are called *applied positions*.

An EAG is *well-formed* iff

(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 is injective.

//

We shall also use the EAG meta syntax for ordinary Knuth-like AGs, which we define in the following way

[2.2] A Knuth-like AG (or just an AG) is an EAG that satisfies: (1) it is well-formed, (2) only (attribute-) variables appear in defining positions, and (3) the same variable appears in only one defining position.

This definition of an AG differs form the one in [Knuth 68] in the following ways:

[2.3a] In Knuth's definition terminals cannot have synthesised attributes.

[2.3b] In Knuth's definition the semantic functions are apparently required to be total whereas [2.2] allows them to be partial.

[2.3c] AGs defined by [2.2] are always in normal form ([Bochmann 76]). By requiring normal form we avoid a number of tedious (and unimportant) complications in the following sections and we exclude only some obscure AGs.

[2.3d] In Knuth's definition, a string is assigned a meaning in the following well known way: (1) A parse tree for the string is constructed. (2) A node in the parse tree and an applied attribute-position of that node is selected. If the attribute-positions referred to (through attribute variables) in the expression of the selected attribute have defined values then the selected attribute-position is assigned the value of the expression. (3) Step (2) is repeated until either all attribute-positions have been assigned a value or no more attribute-position can be assigned a value by this process. (4) The value of a distinguished attribute-position of the root in the parse tree constitutes the meaning of the string corresponding to that parse tree.

[2.3e] Knuth defines an AG to be *well-defined* if all attributes can always be defined, in any conceivable parse tree using the strategy in [2.3d]. He then shows that an AG is well-defined if and only if it is non-circular.

[2.3f] Knuth's model is intended to define *the semantics of context-free languages* in the sense that all parse trees may be assigned a meaning in all well-defined AGs. Other AG formalisms, such as EAGs (and [2.2]) may be viewed as a language generating device in the sense that not all parse trees of the underlying CFG may get values assigned to its attributes. Knuth suggests to let an attribute-position in the root of the parse tree decide whether the parse tree (string) is 'malformed' or not.
The use of partial functions in [2.2] implies that not all parse trees may be assigned attribute values, even if the AG is non-circular. Furthermore a parse tree may be assigned attribute values even if the AG is circular. Thus circularity is not an inherent problem in EAGs and AGs as defined in [2.2]. We return to that later in the paper.

One may reformulate Knuth's definition (2.3d) in order to obtain a definition that is equivalent to [2.2]:

[2.4] A string is assigned a meaning in the following way: (1) A parse tree for the string is constructed. (2) A set of equations corresponding to the parse tree is constructed. Each attribute-position is an unknown; each attribute expression determines an equation in the sense that if a is an attribute expression occupied by the expression e, then a=e is an equation; variables in the expressions are also unknowns and may have to be renamed properly. (3) The parse tree may be assigned attribute values if and only if the equations have a solution. (4) The attributes of the root in a solution constitute the meaning of the string corresponding to the parse tree.

In some definitions of AGs ([Marcotty et al 76]) an AG rule has an associated *constraint* which is a predicate over attribute values. This constraint must be true in order that the attributes of the rule can be assigned values. We interpret constraints in the following way:

[2.5] Each symbol is given a synthesised attribute with a domain consisting of one value. The constraint is converted to a partial function that assigns the value to this attribute of the left side symbol if the constraint is true.

Consequently if some constraint is false there is an attribute that cannot be assigned a value and this is now captured by the definition of AGs in [2.2] or [2.4].

In [Watt & Madsen 77] it is shown how to convert an EAG into an AG by using constraints.

We also make use of *extended attributed translation grammars* (EATGs) ([Watt & Madsen 77]). An EATG consists of a translation grammar (like syntax directed translation schemes in [Aho & Ullman 72]) equipped with attributes in the same way as an EAG is a CFG equipped with attributes. An EATG is naturally divided into an *input-grammar* and an *output-grammar*. The terminals of the input-grammar (output-grammar) are called *input-symbols* (*output-symbols*). As with EAGs input-symbols may have synthesised attributes whereas output-symbols may have only inherited attributes. Each rule in the input-grammar has an associated rule in the output-grammar. The output rule may refer to attribute variables in the input rule but not vice versa. Pairs of input production rules and output production rules are obtained by applying the systematic substitution rule to both the input rule and the corresponding output rule taken together.

When requiring the restrictions in [2.2] to EATGs one obtains *attributed translation grammars* similar to those in [Lewis et al. 74].

An EATG defines a translation from strings of attributed input-symbols to strings of attributed output-symbols.

There is a choice between defining a translation by using an EATG or by using synthesised attributes of an EAG as in definition 2.2. The actual choice depends upon the kind of semantics (or translation) to be defined. It is often a matter of modularity and by using EATGs one may separate the definition of the (context-sensitive) syntax from the definition of the semantics.

As attribute domain constructors we make use of *discriminated unions*, *cartesian products*, *sequences*, and *partial mappings* which (among others) may be found in [Watt & Madsen 77] and [Madsen 79].

3. Defining Verification Generators.

Here we treat the possibilities of expressing predicate transformer semantics by means of EAGs. It is well known that any predicate transformer semantics may be reformulated as a denotational semantics. In section 4 it is shown how any denotational semantics may be defined by an AG. The technique of that section may then be used to define any predicate transformer semantics.

A predicate transformer semantics may be used as a basis for a system which generates verification conditions. In this section we sketch an example on how such a verification generator may be defined by means of an EATG. The example is based upon a forward predicate transformer for partial correctness in the style of [Gerhart 76].

If P is a predicate which is supposed to be true before the execution of a statement S, then the value of the _forward predicate transformer_ FPT(P,S) is a predicate which is true after the execution of S.

Consider the statements :

if B **then** S1 **else** S2 and

while B **assert** A **do** S.

Assert A defines an invariant which must be supplied by the programmer. We may define the following FPTs:

FPT(P, _if_ B _then_ S1 _else_ S2) =

FPT(P AND B, S1) OR FPT(P AND NON B, S2)

FPT(P, _while_ B _assert_ A _do_ S) = A AND NON B

verify: P => A , FPT(B AND A, S) => A.

The FPT for the while-statement is only true if the so-called _verification condition_ following verify can be proved to be true.

A verification generator for a language can be defined by an EATG, where the input grammar defines the (context-sensitive) syntax. The output grammar generates a sequence of verification conditions, and the symbols have predicates as attributes. If <stmt> is the nonterminal generating statements, then in the output grammar <stmt> may typically have two attributes:

<stmt↓P_{before} ↑P_{after} >

where P_{before} is the predicate which is true before the execution of the statement generated by <stmt> and P_{after} a predicate which is true after.

An output rule for if-then-else and while-assert-do might look like (in order to ease the reading, the input symbols are included in the rule):

```
<stmt↓P↑Q OR R> ::=
     if <exp↑B> then <stmt↓P AND B ↑Q>
     else <stmt↓P AND NON B ↑R>

<stmt↓P ↑A AND NON B> ::=
     while <exp↑B> assert <predicate↑A>
     <stmt↓A AND B ↑Q> <verify ↓P => A, Q => A>
```

<exp> has a synthesised attribute B which is the predicate corresponding to the expression generated by <exp> and similarly the synthesised attribute of <predicate> is the invariant supplied by the programmer.

<verify> is an output symbol.

If <prg> is the start symbol of the grammar then we may have a rule

```
<prg↑P> ::= <stmt↓ true ↑P>,
```

where P then will be a predicate which is true after the execution of the program. Instead of initialising the inherited attribute of <stmt> with true one might as well do as follows:

```
<prg↑P> ::= assert <predicate↑A> <stmt↓A ↑P>,
```

where A then is an input assertion.

The idea of generating a verifier from a grammar appears in [Mayoh 76] and is used in the JQNS-system [Nielsen 75]. Most verifiers are designed for a specific language. By means of a TWS based on AGs such verifiers can be automatically constructed from an AG description.

An attribute domain for predicates must be available. In JQNS predicates are basically text strings with an associated set of operations. This is a simple solution. It might be desirable to have a more structural definition of predicates, especially if the verifier is combined with a theorem prover. The domains mentioned in chap. 2 should be sufficient for this.

By using an EATG to define the syntax and a verification generator for a language it should be possible to tie the two definitions together. This is e.g. not the case with the original definition of Pascal where the context-free syntax is defined by BNF, an axiomatic definition appears in [Hoare & Wirth 73], but the context-sensitive part of the syntax is only informally and very imprecisely defined in [Wirth 71].

The output grammar of the EATG defining semantics could rely upon the input grammar defining the syntax. Type checking would normally appear in the input grammar. In case of Pascal the input grammar could check certain other assumptions made by the semantics. E.g. that aliasing does not appear. The input grammar will normally have attributes corresponding to a symbol table to collect declared identifiers. If the semantics needs a renaming of all identifiers then this may be done using the 'symbol table' of the input grammar.

4. Denotational Semantics and Attribute Grammars.

In [Mayoh 78b] it is shown that any AG can be reformulated into an equivalent Denotational Semantics (DS) ([Tennent 76]). An algebraic formulation of AGs is given in [Chirica 76]. In this section we shall discuss the possibilities of reformulating a Denotational Semantics within AGs.

In [Knuth 68] it was proposed that one let a 'meaning' of a string generated by an AG be the synthesised attributes of the start symbol in a parse tree for the string. Furthermore it was shown that this meaning could be any function of the parse tree. According to Knuth an AG defines a function from the set of parse trees into some domain. So AGs are in fact a meta language for defining a kind of mathematical semantics. Reformulation is then a question of using a different meta language.

Another result in [Knuth 68] is that any AG has an equivalent one using only synthesised attributes. The reformulation of AGs as defined by Chirica and Mayoh can be used in order to transform any AG into an equivalent one using only synthesised attributes. This transformation is more natural and constructive compared to the one of Knuth.

Definition 4.1. Attribute grammar notation.

Let A be a symbol of an AG. Define

$$INH(A) = ID_1 \text{ X } ID_2 \text{ X } \ldots \text{ X } ID_k \text{, and}$$

$$SYN(A) = SD_1 \text{ X } SD_2 \text{ X } \ldots \text{ X } SD_n \text{,}$$

where ID_1, ID_2, ..., ID_k, SD_1, SD_2, ... SD_n are the domains of the inherited and synthesised attributes of A.

Let $A_0 \text{ -> } A_1 \; A_2 \; \ldots \; A_m$ be the p'th production in an AG. Define

$$DEF(p) = INH(A_0) \text{ X } SYN(A_1) \text{ X } SYN(A_2) \text{ X } \ldots \text{ } SYN(A_m) \text{, and}$$

$$APP(p) = SYN(A_0) \times INH(A_1) \times INH(A_2) \times \ldots INH(A_m).$$

In general the attributes of rule p are defined by a function:

$$F_p : DEF(p) \times APP(p) \to APP(p)$$

However we assume (as mentioned in chap. 2) that our AGs are in normal form, i.e. F_p is defined by

$$F_p : DEF(p) \to APP(p).$$

There is a canonical correspondence between sets of functions

$$f_0 : DEF(p) \to SYN(A_0)$$

$$f_i : DEF(p) \to INH(A_i),\ i=1,2\ldots,m,$$

and F_p.

If $D=(I_0,S_1,\ldots,S_m)$, where I_0 is a value of the inherited attributes of A_0 and S_i is a value of the synthesised attributes of A_i, $i=1,2,\ldots m$, then the attributes of an instance of rule p are defined as follows:

$$S_0 = f_0(D),$$

$$I_i = f_i(D),\ i=1,2,\ldots,m, \text{ and then}$$

$$F_p(D) = \langle f_0(D), f_1(D), \ldots, f_m(D)\rangle$$

If f_i $(i=0,1\ldots,m)$ defines k attributes then f_i is defined by k functions f_{i1}, $\ldots,f_{ik}$ each defining an attribute. (In practice the functions $f_{i1},\ldots f_{ik}$ will not depend upon the whole of DEF(p).)

Definition 4.2. Reformulated AG with only synthesised attributes.

Let G be an AG. G_s is an AG with only synthesised attributes and defined by the following transformations.

Each symbol A will have one synthesised attribute with domain [INH(A) -> SYN(A)].

Each production has one function that defines the synthesised attribute of the leftside in terms of the synthesised attributes of the right side.

For rule p we get

[4.2*] $S'_0 = \lambda I.f_0(I,S'_1(I'_1(I)),\ldots,S'_m(I'_m(I))),$

where

$I'_i = \lambda I.f_i(I,S'_1(I'_1(I)),\ldots S'_m(I'_m(I))),\ i=1,2,\ldots,m,$

and S'_i is the value of the synthesised attribute of A_i, $i=0,1,2,\ldots,m$, in rule p of the reformulated AG, G_s.

//

As mentioned in def. 4.1 each f_i ($i \in [0..n]$) defines a number of attributes, thus each of the above equations defines a number of equations corresponding to the attributes defined by each f_i. Thus if rule p has k defining positions then the above m+1 equations define k equations.

Below we formulate in what sense G is equivalent to G_s. This is similar to the formulations in [Chirica 76] and [Mayoh 78b].

For each parse tree, [4.2*] defines a set of equations. Each instance of a production p defines a set of equations using [4.2*]. These equations have exactly one solution if the AG is non-circular.

Theorem 4.3.

Let G be a non-circular AG with all semantic functions being total and let G_s be the corresponding AG defined by 4.2. Let t be a parse tree of the underlying CFG, and let $A_0 \rightarrow A_1 \; A_2 \; \dots \; A_m$ be production p and let an instance of p appear in t. A_0 is then a node in t with sons $A_1, A_2, \dots, A_m$.

Let I_i, S_i be the unique values of the inherited and synthesised attributes of A_i, $i=0,1,2,\dots,m$. The equations associated with p have exactly one solution, and $S'_0(I_0) = S_0$ and $I'_i(I_0) = I_i$, $i=1,2,\dots,m$.

proof:

We use structural induction on t.

Bottom : Assume that $A_1, A_2,\dots,A_m$ are terminals, i.e. leaves in t. The only unknown in the equations is S'_0 which is well defined as it only depends on I_0 and $S_1,\dots S_m$, and clearly $S_0 = S'_0(I_0)$.

Induction step: Assume that S'_i ($i\in[1..m]$) are defined and that $S'_i(I_i) = S_i$.

As G is non-circular and t is fixed there is a partial ordering of the attributes in p, such that x<y means that the value of y depends on the value of x. This partial ordering can be used to solve the equations by subsitution.

Let B_i ($i\in[1..k]$) be the attributes of the defining positions of p, and let $B_i<B_{i+1}$ ($i\in[1..k-1]$) where the partial dependency ordering is extended to some total ordering). Each B_i has an associated equation defining a function B_i': $[INH(A_0)\rightarrow DOM(B_i)]$. The equation defining B_i' is independent of $B_1',\dots, B_{i-1}'$; if not then G is circular. Consequently the equations may be

solved by substitution in the order B_1', B_2',..., B_k'.

//

4.1 A redefinition of Attribute Grammars.

In his original definition of AGs Knuth has excluded circular AGs in order to assure that all attributes in all possible derivation trees can be assigned unique values. This may be too strong a requirement. Consider the following examples:

- consider def. 4.2. If we transform a given AG into one with only two attributes for each symbol A, an inherited with domain INH(A) and a synthesised with domain SYN(A), then the new AG is circular if some inherited attribute of A depends on some synthesised attribute of A. However if the original AG is well defined then the new one should not give problems with assigning values to attributes.

- using conditional expressions . Consider the rules:

```
<A↑y+z> ::= <B↓cond(c,y,z)↑c↑y↑z>
     where cond(c,y,z)=If c then y else z

<B↓x↑true↑7↑x+2> ::= 'a'

<B↓x↑false↑x+1↑8> ::= 'b'
```

  The dependency graph for the derivation <A> => <B> => a has cycles in it but there is no problem in assigning unique values to the attributes.

The equations [4.2*] give a basis for discussing circularity. A given parse tree defines a set of equations defined by the productions in the tree and the corresponding equations [4.2*]. These equations may have either

1. Exactly one solution,

2. more than one solution, or

3. no solution.

Case 1 captures all non-circular AGs and some circular ones where unique values may be assigned. Case 2 and 3 capture circular AGs where unique values cannot be assigned.

In the style of denotational semantics we shall now assume that an AG is extended in the following way. The domains are extended to Scott-type domains with a least element and a partial ordering. The equations associated with a parse tree and defined by [4.2*] will then always have a unique least solution.

In this connection it is thus natural to define the meaning of a parse tree based on the minimal solution to the corresponding equations. *In the rest of section 4 we shall thus use this definition of AGs*.

This in fact gives a difference when compared to EAGs. If there is no solution to the equations then the EAG cannot generate the corresponding input string. If there are more than one solution then the EAG may generate the string in ways corresponding to each solution.

As mentioned in section 2 the definition of EAGs in [2.1] and the corresponding AG definition in [2.4] give also an interpretation to circular AGs. When we extend the domains to Scott-type domains we could also base 'the least solution' approach to AGs on the minimal solution to the equations defined in [2.4].

If we when using Scott-domains use the EAG approach then we must define an ordering upon attributed parse trees in order to get a unique attribute assignment to all parse trees.

With these constructions of 'the least solution' approach we loose the ability to recognise certain kinds of semantic ambiguity. This may be the case if the EAG can generate distinct attributed parse trees with the same corresponding (context-free) parse tree. But this is perhaps reasonable since any AG-evaluator will probably compute the least solution.

4.2 Relation to Denotational Semantics.

By using the reformulated AG of definition 4.2 it is straightforward how to make an equivalent Denotational Semantics. However this reformulation is in general very complicated and unreadable as one must express a general solution to the equations independent of the parse trees. In [Mayoh 78b], [4.2*] is viewed as a set of equations defining a function from the set of parse trees into the attribute domains of the start symbol of the AG. Such a solution using the fixpoint operator may be found in [Mayoh 78b]. He shows that if the AG is non-circular then the solution may be expressed without using fixpoints. Next Mayoh defines a hierarchy of AGs which in turns simplifies the solution and improves the readability. The hierarchy classifies AGs by the evaluation order of attributes and is also interesting when considering implementations of AGs.

Here we shall consider the opposite direction, namely converting a denotational semantics into an equivalent AG. This is done by passing denotations as attributes. It should be clear that the two formalisms are equally powerful so the purpose of this is to show how AGs may be used to express a denotational semantics. We note that in general mechanical transformations from one meta language into another may not work very well in the sense that the readability of the transformed definition will usually be bad. It is like transforming PASCAL programs mechanically into FORTRAN programs.

4.2.1 A simple reformulation.

A straightforward transformation of a denotational semantics(DS) is as follows:

A DS is defined by means of functions from syntactic domains (parse trees) into some semantic domains. Let Com be a syntactic domain for the nonterminal <com> (then Com is the set of parse trees derivable from <com>). Consider the semantic function

cc: [Com -> A], where A is some semantic domain, and the production

$<com> ::= w_0 \ <A_1> w_1 \ <A_2> \ldots <A_n> w_n$, and the semantic equation

$cc[w_0<A_1>w_1<A_2>\ldots<A_n>w_n] = f(aa_1[A_1],aa_2[A_2],\ldots,aa_n[A_n])$

where f is some function and aa_i (i=1,2,...,n) are semantic functions.

In the corresponding AG, <com> will have a synthesised attribute with domain A and we will have the following rule:

$$<com\uparrow f(C_1,C_2,...,C_n)> ::= w_0 <A_1\uparrow C_1> w_1 <A_2\uparrow C_2> ... <A_n\uparrow C_n> w_n .$$

If there are more functions defined on the domain Com, then <com> has a synthesised attribute for each of these.

4.2.2. A reformulation using a circular AG.

If we turn to more specific DSs, then the meaning of a construct like <com> is often defined relatively to an environment (Env) and a (command) continuation (CC), i.e.

A=[Env -> CC -> B]

In the AG terminology Env and CC intuitively correspond to inherited attributes and B to a synthesised attribute. This will in fact correspond more to the domain [Env X CC -> B] which however is isomorphic to [Env -> CC -> B].

This is illustrated by the following example:

Example 4.4. Denotational Semantics.

Syntactic Domains.

PROG=Programs
COM=Commands
EXP=Expressions
VAR=Variables
CONST=Constants

Semantic Domains.

V=Values=Integers
S=States=[VAR -> V]
CC=Command Continuations = [S -> S]
EC=Expression Continuations = [V -> EM]

EM=Expression Meaning = [CC + EC]

Note that for technical reasons we use an expression continuation which is (slightly) different from what is normal been used. We return to that later. An EC will require a sequence of values in order to produce a CC. Consider the following auxillary domains: EC_0=CC, and for each n>0 EC_n=[V -> EC_{n-1}]. For a given k ∈ EC there exists an n>0 such that k ∈ EC_n.

We do not use an environment in this example as this will not change the principle of the reformulation. In fact environments are straightforward to pass as inherited attributes whereas continuations cause more problems.

Auxillary Functions.

COND : [CC X CC -> [V -> CC]]

COND(c_1,c_2)(v)= if v>0 then c_1 else c_2

CONTENT : [VAR X EC -> EC]

If k ∈ EC_n, n>0 then CONTENT(A,k) ∈ EC_{n-1} and

CONTENT(A,k)v_2...v_n s = k(s(A))v_2..v_n

UPDATE: [VAR X CC -> [V -> CC]]

UPDATE(n,c)(v)s = c(s[v/n])

Syntax

```
<program> ::= <com>

<com> ::= <com> ; <com> | <var> := <exp>
    | if <exp> then <com> else <com>
    | while <exp> do <com>

<exp> ::= <exp> + <exp> | <var> | <const>
```

Semantic Functions.

pp: [PROG -> CC]

cc: [COM -> [CC -> CC]]

ee: [EXP -> [EC -> EM]]

vv: [CONST -> V]

For each exp in a program there will exist an n>0 such that ee[exp] ∈ $[EC_n \rightarrow EC_{n-1}]$. This is easily seen by checking the semantic equations below.

Consider the command $a{:=}e_1+e_2+e_3+e_4$, and assume a left associative parse tree. Then for each i ∈ [1..4], $ee[e_i] \in [EC_i \rightarrow EC_{i-1}]$.

Semantic equations

$$pp[com] = cc[com]c_0 \quad \text{where } c_0 \text{ is the initial continuation}$$

$$cc[com_1;com_2]c = cc[com_1]\{cc[com_2]c\}$$

$$cc[var{:=}exp]c = ee[exp]\{UPDATE(var,c)\}$$

$$cc[\text{if } exp \text{ then } com_1 \text{ else } com_2]c = ee[exp]\{COND(cc[com_1]c,cc[com_2]c)\}$$

$$cc[\text{while } exp \text{ do } com]c = \underline{fix}\ \lambda c'.ee[exp]\{COND(cc[com]c',c)\}$$

$$ee[exp_1+exp_2]k = ee[exp_1]\{ee[exp_2]\{\lambda v_2.\lambda v_1.k(v_1+v_2)\}\}$$

$$ee[var]k = CONTENT(var,k)$$

$$ee[const]k = k(vv[const])$$

//

Example 4.5. An equivalent attribute grammar.

In the AG the idea is to let each symbol have two attributes, an inherited defining its continuation and a synthesised defining its meaning in that contiuation. We will e.g. have the following rule:

$$<com_0 \downarrow C \uparrow C2> ::= <com_1 \downarrow C1 \uparrow C2> \ ; <com_2 \downarrow C \uparrow C1>$$

The meaning of $<com_0>$ in a given continuation C, is C2 which is the meaning of

<com$_1$> in the contiuation C1 which again is the meaning of <com$_2$> in the contiuation C.

We use the same semantic domains as in the DS.

<u>Symbols and their attribute domains</u>

<prog↑CC>
<com↓CC↑CC>
<exp↓EC↑EM>
<var↑VAR>
<const↑V>

<u>AG rules</u>

<prog↑C> ::= <com↓C0↑C>

<com↓C↑C2> ::= <com↓C1↑C2> ; <com↓C↑C1>

<com↓C↑C1> ::= <var↑N> := <exp↓UPDATE(N,C)↑C1>

<com↓C↑C3> ::= if <exp↓COND(C1,C2)↑C3> then <com↓C↑C1> else <com↓C↑C2>

<com↓C↑C2> ::= while <exp↓COND(C1,C)↑C2> do <com↓C2↑C1>

<exp↓K↑K2> ::= <exp↓K1↑K2> + <exp↓λV2.λV1.K(V1+V2)↑K1>

<exp↓K↑CONTENT(N,K)> ::= <var↑N>

<exp↓K↑K(C)> ::= <const↑C>

//

If the above AG is transformed into an equivalent one using the method of section 4.2, then we obtain synthesised function attributes which are similar to the functions defined by the semantic equations. AS the while-imperative is the most difficult one (involves circularity) we shall make the transformation.

The corresponding rule is:

<com↑C'> ::= while <exp↑E'> do <com↑C'1>

where C'= λC. C2,

$C2 = E'(K),$

$K = COND(C1,C),$ and

$C1 = C'1(C2).$

This implies that

$C2 = E'(COND(C'1(C2),C))$

and thus

$C' = \lambda C.\ \underline{fix}\ \lambda C2\ .E'(COND(C'1(C2),C))$

Now comparing this with the while-equation and letting C'=cc[while exp do com], E'=ee[exp], and C'1=cc[com] we see that our reformulated AG defines the same function.

About expression continuations.

As mentioned we use another definition of the meaning of expressions than the usual one. A more standard one is

EC = [V -> CC]

ee: [EXP -> [EC -> CC]]

Consider the rule:

$ee[exp_1+exp_2]k = ee[exp_1]\{\lambda V_1 .ee[exp_2]\{\lambda V_2 .k(V_1+V_2)\}\}$

In our corresponding AG <exp> should have the domains <exp↓EC↑CC>, and the AG rule should be

<exp↓K↑K2> ::= <exp↓λV1.K1↑K2> + <exp↓λV2.K(V1+V2)↑K1>

Now this does not work as λV2.K(V1+V2) is passed as an EC attribute with V1 as a free variable. This works in the DS-rule as V1 is bound past the meaning of exp_2.

The meaning of a construct (command or expression) is interpreted relative to a function (its continuation) which specifies what is to be done after executing the construct. When applying the meaning of a construct to its continuation one gets the meaning of the whole program if executing it beginning with the construct.

If we consider expressions then the meaning of an expression (relative to a continuation) depends on the context of the expression. Consider the expression exp in the following constructs:

(1) If exp then com_1 else com_2

(2) exp' + exp

In the above DS the meaning of an expression (relative to an EC) is a CC. In case (1) the EC of exp can be determined by the program text alone (a static continuation). In case (2) the EC of exp depends upon the execution of the program (a dynamic continuation) since it includes the value of exp'.

In an AG one can only express as attributes values which are a static property of the program text unless one turns to include rules for executing the program. One may then discuss whether or not it is reasonable to require a continuation to be static or not. Perhaps it is.

Towards a general rule for using inherited attributes.

There is of course no general rules for transforming a DS into and AG with both inherited and synthesised attributes. It seems likely that a semantic function in a DS can be made more readable by decomposing it into a number of attributes when each attribute is a static property of the program text. One may often benefit by converting a semantic function f with domain [A->B] into an inherited attribute with domain A and a synthesised attribute with domain B.

This may be reasonable if A is naturally expressed as a (static) property determined by the context of the constructs defined by f. The following is a rule for this to be natural and possible:

[4.6] Consider a semantic rule

c[..] = e

where e involves one or more applications of f and possibly c=f.

(1) all occurences of f in e must be applied to an expression of type A.

(2) If f is applied to the expression a, then all free variables in a must be convertable to attributes.

The semantic functions cc and ee of example 4.4 satisfy this whereas ee with the 'standard' definition of EC in the previous section does not.

An extension of the AG model which makes it possible to express dynamic properties as attributes is discussed in [Gantzinger 79a]. In section 5 we shall (among others) demonstrate a technique for doing this within the existing model of AGs.

4.2.3 Other reformulations.

Instead of passing functions around as attributes one might pass lambda-expressions. This will look like the reformulation of section 4.2.1 but be quite different. The meaning of a program will then be a lambda-expression instead of a function. This would then correspond to an AG defining a code generation. In a practical TWS based on AGs this might be a reasonable way of implementing a denotational semantics.

Yet another approach would be to define a denotational semantics by means of syntax directed translation schemes as used in [Aho & Ullman 72]. In their generalised translation schemes they allow nonterminals to have translation elements other than just strings, e.g. integers, booleans. If one allows translation elements to be functions (or lambda-expressions) then a denotational semantics may be defined in a notation which is quite close to the usual notation of denotational semantics. Such a definition will however just be another notation for the one of section 4.2.1 (or the above mentioned).

4.3 Conclusion.

It has been shown that AGs are a suitable tool for defining a TWS in which compilers may be generated based upon a denotational semantics, like e.g. SIS ([Mosses 79]). We also think that the AG notation in many situations gives a more natural and readable definition than the corresponding DS. This is due to the fact that one in the AG may have simpler domains and thus simpler expressions.

A further modularization (and simplification) can be obtained by using a model based on EATGs. Here it is possible to separate the context-sensitive syntax from the semantics.

If one does not like the AG notation then with the right TWS it should be no problem to define ones own notation and just use AGs as an implementation.

In addition to a higher degree of modularity in the semantic definitions one may also benefit when the semantic definition has to be converted into a more implementation oriented semantics. In the AG it is possible to isolate the static propertiets of the definition; the context-sensitive syntax may (as mentioned) be isolated and the semantic functions may (as mentioned) be split up into attributes describing static properties, like environments and contiuations. The possibilities of transforming a DS into an implementation oriented AG have been studied by [Bjorner 78] and [Gantzinger 79b].

5.Defining operational semantics by Attribute Grammars.

In this section a technique to specify operational semantics by means of EAGs will be presented. The approach is to specify a set of EAG rules which defines the possible transformations upon an abstract representaion of the program. Consequently it is not a traditional operational semantics where the program is transformed into code for a hypothetical machine which then executes the code. The examples presented in this section are inspired by recent work in the area of specifying abstract data types and make use of techniques which have been used with vW-grammars ([Marcotty et al.76]). Section 5.1 is related to the specification of abstract data types, section 5.2 is about specifying semantics of programming languages, and finally section 5.3 shows how AGs may be viewed as a model for defining nondeterministic and concurrent computations.

5.1 Specifying Abstract Data Types.

An abstract data type is considered to consist of an (abstract) set of values and an (abstract) set of operations. The operations may be combined into expressions denoting abstract values.

An abstract data type is specified by an EAG in the following way: The EAG generates the set of all expressions yielding values of the data type. The synthesised attributes of the start symbol is then the value of the generated expression. The values of the data type are defined by the domains of the EAG and the set of expressions and their values are defined by the production rules of the EAG.

We illustrate the approach by specifying the famous stack:

Example 5.1. Stack specification.

Domain

```
   S: SEQ=(empty | cat(SEQ , ELM))
   E: ELM
```

The values of a stack is a sequence of elements (not specified here).

Rules

<stack↑empty> ::= newstack

<stack↑cat(S,E)> ::= push (<stack↑S> , <element↑E>)

<stack↑S> ::= pop (<stack↑cat(S,E)>)

<element↑E> ::= top (<stack↑cat(S,E)>)

<boolean↑true> ::= empty (<stack↑empty>)

<boolean↑false> ::= empty (<stack↑cat(S,E)>)

// The above EAG generates all valid stack expressions and the synthesised attribute of <stack> is the value of the stack expression. An example of a stack expression is

push(pop(push(push(newstack ,e1),e2)),e3)

with the value

cat(cat(empty,e1),e3).

Note that this EAG also contains rules for generating expressions of type element and boolean.

One nice property of the EAG is that errors are treated implicitly in the sense that only valid stack expressions can be generated. E.g. it is impossible to generate an expression like: pop(newstack).

The type ELM could be integers in which case we could define the domain

E: ELM=INTEGER=(zero | suc(INTEGER) | pred(INTEGER))

The stack example makes only use of synthesised attributes. Below we define the data type partial mapping from a set D into a set R ({D->R}). Let d:D, r:R, f,g:{D->R}, then {} is the empty map, {d->r} is the map defined in one point, fUg is the union of f and g (only defined if the domains of f and g are disjoint), f\g is the overriding of f by g (the values of g are used 'before' those of f).

Example 5.2. Specification of partial mapping.

Domain

```
f,g,h: M=(empty | add(M,D,R))
d: D=...
r: R=...
```

Rules

```
<map↑empty> ::= {}

<map↑add(empty,d,r)> ::= { <dom↑d> -> <range↑r> }

<map↑h> ::=
    <map↑f> U <map↑g> <disjoint↓f↓g> <union↓f↓g↑h>
|   <map↑f> \ <map↑g> <union↓f↓g↑h>

<range↑r> ::= apply ( <map↑f> , <dom↑d>) <apply↓f↓d↑r>

<disjoint↓empty↓f> ::= EMPTY

<disjoint↓add(f,d,r)↓g> ::= <undef↓d↓g> <disjoint↓f↓g>

<undef↓d↓empty> ::= EMPTY

<undef↓d↓add(f,d,r)> ::= <not-equal↓d↓d1> <undef↓d↓f>

<union↓f↓empty↑f> ::= EMPTY

<union↓f↓add(g,d,r)↑add(h,d,r)> ::= <union↓f↓g↑h>

<apply↓add(f,d,r)↓d↑r> ::= EMPTY

<apply↓add(f,d1,r1)↓d↑r> ::= <not-equal↓d↓d1> <apply↓f↓d↑r>
```

The nonterminals <disjoint> ,<undef>, <union> and <apply> are used in a special way. They can only generate the empty string and they do if some relations hold between their actual attributes. This technique is also used in connection with vW-grammars where such nonterminals are called primitive predicate symbols. Nonterminals may be classified according to their attribute structure:

- nonterminals with only synthesised attributes correspond to domains (or sets),

- nonterminals with only inherited attributes correspond to predicates , and
- nonterminals with both synthesised and inherited attributes correspond to relations (or functions) between its inherited and synthesised attributes.

Comparison with other methods.

In e.g. [Gougen et al. 75] abstract data types are specified in form of an algebraic specification (and so is also semantics of programming languages). In these approaches a data type is (the isomorphism class of) a many-sorted algebra

$$D = (D_1, D_2, \ldots, D_n, f_1, f_2, \ldots, f_m),$$

where $D_1, D_2, \ldots, D_n$ are an indexed family of sets (carriers) and $f_1, f_2, \ldots, f_m$ are an indexed family of operations between the carriers. Some of the operations denote constants as they have an empty domain. In this way the set of well formed expressions built by means of the functions defines (or denotes) the values of the data type. A set of equations (or sometimes axioms) is used to identify expressions which should denote the same value. The values in the domains are then determined by the equivalence classes defined by these equations.

The stack may be defined in the following way:

```
Stack=(Istack,Integer,Boolean,push,pop,top,newstack,empty)

   newstack: -> Istack
   push: Istack X Integer -> Istack
   pop: Istack -> Istack
   top: Istack -> Integer
   empty: Istack -> Boolean
```

Equations

```
    pop(push(S,I))=S
    top(push(S,I))=I
    empty(newstack)=true
    empty(push(S,I))=false
```

These equations correspond to ones intuition about a stack. They do not state that e.g. pop and top are illegal on an empty stack. In order to handle such errors, a special error value is introduced, and any expression containing an error value is the error value. Additional error equations are introduced. For the stack they would be

pop(newstack) = error
top(newstack) = error

See e.g. [Goguen 77].

For a given algebraic definition there exists a canonical term algebra and a homomorphism from the free algebra into the canonical term algebra.

The canonical term algebra has the property that all different expressions belong to different equivalence classes, i.e. denote different values. This canonical term algebra is in our opinion essential in order to understand the data type being defined.

An approach which is more similar to the EAG approach is exploited in [Mayoh 78a]. Mayoh defines a data type to be a (total) function, f, from a set of expressions ,E, to a (partially ordered) set of values V (with a least element in the partial order). This approach is similar to the EAG approach in the sense that the abstract values (V) are specified explicitly. In the EAG approach the abstract values are specified as the domains of the EAG. In Mayoh's approach it is (like in EAGs) explicitly specified (by f) how to reduce an expression to a value.

These approaches differ from the algebraic methods in several ways: The algebraic methods do not specify the abstract values explicitly but use equations to identify expressions that denote the same value. The equivalence classes or the canonical term algebra may then be viewed as the abstract set of values. The function f of Mayoh's method may then be compared to the homomorphism between the free algebra and its canonical term algebra.

One problem with the Mayoh/EAG methods is that the abstract values also have to be specified by using some meta language. It may be difficult and awkward to define those abstract values such that different abstract value expressions in fact are different values of the data type being specified. Often it is more natural to reduce expressions of the data type into a unique value (-expression).

5.2 Specifying Semantics of Programs.

The techniques used for defining abstract data types may easily be used to define an operational semantics of a programming language. This is done by using an EAG to define all possible executions of a given program. In [Marcotty et al. 76] a small programming language is defined this way but using vW-grammars.

We define the semantics of the language presented in the example of 4.2.2.

Example 5.3

Domain

```
    T: TREE=(seq(TREE,TREE) | assign(NAME,EXP)
             | cond(EXP,TREE,TREE) | rep(EXP,TREE));
    E: EXP=(plus(EXP,EXP) | v(NAME) | c(INTEGER));
    N: NAME;
    I: INTEGER;
    S: STATE={NAME->INTEGER}
```

Rules

```
   <program↑S> ::= <stmt↑T> <execute↓{}↓T↑S>

   <stmt↑seq(T1,T2)> ::= <stmt↑T1> ; <stmt↑T2>

   <stmt↑assign(N,E)> ::= <var↑N> := <exp↑E>

   <stmt↑cond(E,T1,T2)> ::= if <exp↑E> then <stmt↑T1> else <stmt↑T2>

   <stmt↑rep(E,T)> ::= while <exp↑E> do <stmt↑T>

   <var↑N> ::= <name↑N>

   <exp↑plus(E1,E2)> ::= <exp↑E1> + <exp↑E2>

   <exp↑v(N)> ::= <var↑N>

   <exp↑c(I)> ::= <const↑I>

   <execute↓S↓seq(T1,T2)↑S2> ::= <execute↓S↓T1↑S1> <execute↓S1↓T2↑S2>

   <execute↓S↓assign(N,E)↑S\{N->I}> ::= <eval↓S↓E↑I>
```

```
<execute↓S↓cond(E,T1,T2)↑S1> ::=
     <eval↓S↓E↑pos(I)> <execute↓S↓T1↑S1>
  |  <eval↓S↓E↑zero> <execute↓S↓T2↑S1>

<execute↓S↓rep(E,T)↑S> ::= <eval↓S↓E↑zero>

<execute↓S↓rep(E,T)↑S2> ::=
     <eval↓S↓E↑pos(I)> <execute↓S↓T↑S1> <execute↓S1↓rep(E,T)↑S2>

<eval↓S↓plus(E1,E2)↑I1+I2> ::=
     <eval↓S↓E1↑I1> <eval↓S↓E2↑I2>

<eval↓S↓v(N)↑S[N]> ::= EMPTY

<eval↓S↓c(I)↑I> ::= EMPTY
```

The definition of the language consists of some rules that define the syntax and collect the given program in a tree structure. The remaining rules define an execution of the given program starting with an empty state and returning a final state. the result of the program is this final state which is the meaning of the program (a synthesised attribute of the start symbol).

The integers are defined in the following way:

```
Integer=(neg(N) | zero | pos(N))
N=(one | suc(N))
```

5.3 Nondeterminism and Concurrency.

We shall now investigate the possibilities for using EAGs as a model for defining nondeterministic and concurrent computations.

Let us for a moment consider CFGs as a model for defining computations consisting of derivations, and let a meaning of a (terminal-) string be a derivation of it. CFGs are by nature nondeterministic because of the alternative operator |.

For a given derivation one may define a partial order between applications of productions in the derivation: Let r and s be applications of productions, then r<s if s cannot be done before r is done. Juxta position in a CFG may then be

viewed as a concurrency operator as all nonterminals of a sentential form may be rewritten independently of each other.

If one turns to context-sensitive grammars then juxta position is no longer necessarily a concurrency operator as some dependence may exist between symbols in a sentential form.

We shall not go further with this view upon Chomsky-grammars but instead turn to EAGs. We shall only consider non-circular EAGs in this connection.

Nondeterminism of an EAG is still described by the alternative operator in the sense that a given string may have several parse trees and thus several meanings. Consider a parse tree t. The dependency graph defines a partial ordering between attributes in t in the sense that a<b if the value of b depends upon the value of a. This partial ordering may be interpreted to define the amount of *concurrency* that can be applied during the computation of the attribute values by some machine.

A dependency and independency relation may be defined between symbols on the right side of a production:

Let A and B be symbols on the rightside of some production. A *depends* on B if an attribute in a defining position of B is used in an applied position of A. A and B is *independent* iff neither A is dependent on B nor vice versa.

Consider example 5.3 and the rules defining <execute↓S↓seq(T1,T2)↑S2> and <eval↓S↓plus(E1,E2)↑I1+I2>. In the rule defining seq(T1,T2) the two instances of <execute> on the right side are dependent whereas the two instances of <eval> on the right side of the rule defining plus(E1,E2) are independent. In the first case sequentiallity is imposed. In the second case no evaluation order is imposed and it may in fact go on concurrently.

Below we give four examples of how EAGs may be used to define nondeterminism and concurrency in programming languages.

Example 5.4 *Nondeterminism*.

Assume that we add the following statement to the grammar of example 5.3:

<stmt> ::= *one* *of* <stmt> *or* <stmt> *end*

We may then extend the definition of TREE by oneof(TREE,TREE), and add the rules

```
<stmt↑oneof(T1,T2)>::= oneof <stmt↑T1> or <stmt↑T2> end

<execute↓S↓oneof(T1,T2)↑S1> ::=
      <execute↓S↓T1↑S1>
  |   <execute↓S↓T2↑S1>
```

Now either of the two alternatives may be selected in a given execution.

Example 5.5. Interleaving.

Consider the following statement:

```
   allof <- S1 -> ; <- S2 -> ; ... ; <- SN -> ;
   and <- S1' -> ; <- S2' -> ; ... ; <- SM' -> ;
   end
```

The two sequences of statements may be executed in 'parallel'. The statements enclosed by <- , -> are considered to be indivisible actions. Parallel execution will in this case then mean interleaving of the two sequences of indivisible actions. This kind of 'parallel' construct is in fact more nondeterministic than parallel.

We extend the definition of TREE by allof(ITREE,ITREE), and add the domain

```
   IT: ITREE={TREE}*
```

and add the rules

```
<stmt↑allof(IT1,IT2)> ::=

     allof <indiv-stmt↑IT1> and <indiv-stmt↑IT2> end

<indiv-stmt↑[]> ::= EMPTY

<indiv-stmt↑T.IT> ::= <- <stmt↑T> -> ; <indiv-stmt↑IT>

<execute↓S↓allof(T.IT,IT1)↑S2> ::=
     <execute↓S↓T↑S1> <execute↓S1↓allof(IT,IT1)↑S2>

<execute↓S↓allof(IT1,T.IT)↑S2> ::=
     <execute↓S↓T↑S1> <execute↓S1↓allof(IT1,IT)↑S2>
```

```
<execute↓S↓allof([],[])↑S> ::= EMPTY
```

Example 5.6. Concurrency.

We now turn to specify the semantics of concurrent computations. By concurrent we mean that two computations can be carried out independently of each other. Consider the following construct:

```
cobegin S1 and S2 end
```

We shall require that S1 and S2 do not refer to the same variables in order to ensure that S1 and S2 can in fact be executed concurrently.

In order to specify the semantics of cobegin we make the following extensions of example 5.3:

- <stmt> is extended with an extra synthesised attribute that is used to collect the set of names used in the statement,
- TREE is extended with co(TREE,TREE),
- the following rules are added

```
<stmt↑co(T1,T2)↑R1 U R2> ::= cobegin <stmt↑T1↑R1> and <stmt↑T2↑R2> end

<execute↓S↓co(T1,T2)↑S1\S2>::= <execute↓S↓T1↑S1> <execute↓S↓T2↑S2>
```

Now why does this define a concurrent execution of T1 and T2. Consider the last rule: The two instances of execute on the right side are independent according to our previuos definition, i.e. this defines two independent computations. Note that the expression S1\S2 is not symmetric. One might instead pass only the restriction of S corresponding to the names being used by T1 to the first <execute> and similarly for the second. The result will then be S\(S1 U S2) instead of S1\S2.

Example 5.7. Guarded commands.

We conclude the examples of this section by giving a semantics of the well known guarded if-statement. For simplicity we only allow two alternatives.

We extend example 5.3 in the following way:

- TREE is added if(EXP,TREE,EXP,TREE),
- we add the rules

```
<stmt↑if(E1,T1,E2,T2)> ::=
   if <exp↑E1> -> <stmt↑T1> [] <exp↑E2> -> <stmt↑T2> fi

<execute↓S↓if(E1,T1,E2,T2)↑S1> ::=
     <eval↓S↓E1↑V1> <eval↓S↓E2↑V2> <select↓S↓V1↓T1↓V2↓T2↑S2>

<select↓S↓pos(I)↓T1↓V↓T2↑S1> ::= <execute↓S↓T1↑S1>

<select↓S↓V↓T1↓pos(I)↓T2↑S1> ::= <execute↓S↓T1↑S1>
```

The evaluation of the guards may go on concurrently; the selection has to wait for all guards to be evaluated. If both guards are true (positive values) then there is a nondeterministic selection between the two alternatives. If both guards are false then <select> cannot generate the empty string and the computation is thus undefined.

Note that if both guards are true and one of the alternatives fails because of an error then the whole command is still well defined as the other alternative may be selected. We may repair this defect by always enforcing all alternatives to be 'executed' and then only use one of the results. This seems to be necessary when using an operational definition. The problem is to handle non-terminating programs.

It is straightforward to include a definition of the guarded do-statement. The main extensions are that <stmt> must have an alternative corresponding to all guards being false, and <execute> must be recursive in the same way as it defines the while-statement in ex. 5.3.

5.4 Conclusion.

It has been shown how to use EAGs to define operational semantics for various programming language constructs. In our opinion these definitions are straightforward to make and easy to read. Again this is a matter of personal opinion. The useability of such definitions have to be tested on real programming languages.

Another interesting question is the possibility for generating a compiler automatically from such a description. We shall not discuss methods for doing this but just indicate some of the problems and give some ad hoc rules that may be a basis for further investigations.

Let nonterminals that only generate the empty string be called *primitive predicates*. (like <execute> and <eval> in example 5.3)

None of the standard AG-evaluators (including the one of section 7) are directly useable for this puspose. The reason is that there are an infinite number of parse trees (due to the heavy use of primitive predicates) to be considered.

It is possible to strip the underlying CFG for primitive predicates and then construct a parse tree in the usual way (this may still cause difficulties if the underlying CFG is ambiguous, but the EAG is structurally unambiguous). Afterwards one may then fill in the primitive predicates by simulating all possible derivations. In order to make this process efficient one must probably restrict the degree of nondeterminism in order to recognise deterministic parts of the EAG.

The generation process may be simplified if all rules defining primitive predicates are required to be left-attributed. This is in fact the case in all our examples.

In some cases it is possible to factorise the EAG. Consider the rules in ex. 5.3 defining the execution of cond(E,T1,T2). These rules may be factorised into one rule with the following right-side: <eval↓S↓E↑V> <selection↓S↓T1↓T2↓V↑S1> where <selection> selects between T1 or T2 depending upon the value of V. This definition makes it easier to avoid two evaluations of <eval>.

6. Pure and Multi-level Extended Attribute Grammars.

Here we shall give certain proposals for future work with the AG/EAG formalisms.

When using an EAG to define a language one needs to specify the domains and operations upon these domains. A disadvantage of EAGs (AGs in general) is that the formalism itself does not contain a method for specifying these domains. This disadvantage is not present in vW-grammars, (pure) affix-grammars and extended affix grammars as the domains here are specified as context free (string) languages.

In section 5.1 we have seen that EAGs may be used to define abstract data types. We shall thus propose to use EAGs to define the domains of a particular EAG. This has the effect that EAGs now become a complete formalism. In order to accomplish this we must define a bottom in the hierarchy of EAGs, i.e. a domain constructor which is part of the formalism. Because of the similarities between extended affix grammars and EAGs it seems natural to let extended affix grammars be the bottom. However as argued in [Watt & Madsen 77] the discriminated union seems much more adequate as the basic domain constructor. A discriminated union may be viewed as an abstract context free grammar and is useful to define abstract properties of a language instead of its concrete syntax. A sentential form of a CFG does not contain information about the structure of the form (how it is derived) whereas this is explicitly available in a discriminated union. Altogether we find the latter to be more useful as it leads to more compact and readable descriptions.

A set of discriminated unions can be viewed as a regular tree grammar ([Engelfriet 74]). The theory of tree languages is well founded and gives a good notation. In our formal model we shall thus use tree languages as the basic domain.

In the following we shall need some concepts from the theory of tree automata and tree grammars. We use the terminology of [Engelfriet 74], especially the following concepts: (1) ranked alphabet W, (2) T_W, the set of trees over W, (3) regular tree grammar, and (4) tree rewriting system.

Definition 6.1

A Pure (Extended) Attribute Grammar (PEAG) is a 5-tuple:

$G = (D,V,Z,B,R)$.

The elements of G are similar to the ones of an EAG except that $D = (N,W,P,S)$ is a regular tree grammar and the domains of attribute positions and variables are defined as nonterminals of D.

//

Below we shall make relations between PEAGs and the theory of tree automata and tree languages. We show that the rules of a PEAG can be formulated as a tree rewriting system.

We shall construct two kinds of tree rewriting systems correpsonding to the rules of a given PEAG.

First we construct a top down tree rewriting system $TD=(W \cup \{[,]\} ,TR)$ where W is a ranked alphabet and TR is a set of tree rewriting rules (T-rules).

W is constructed as follows:

(1) $W_0 = V_T$ (the terminals of G);

(2) $A \in W_k$ if $A \in V_N$ (the nonterminals of G) and there is a rule in R of the form: $A ::= A_1 A_2 \ldots A_k$, $k \geq 0$;

(3) $A' \in W_k$ if $A \in V=(V_N \cup V_T)$ and A has k attribute positions.

TR is constructed as follows:

Let $B_0 ::= B_1 B_2 \ldots B_p$ be a rule in R, where each B_i ($i \in [0..p]$) has the form

$<A_i \updownarrow e_{i1} \updownarrow e_{i2} \updownarrow \ldots \updownarrow e_{in}>$

where $in \geq 0$ is the number of attribute positions for A_i. A_0 will be in W_p and for each A_i ($i \in [0..p]$), A'_i is in W_{in}.

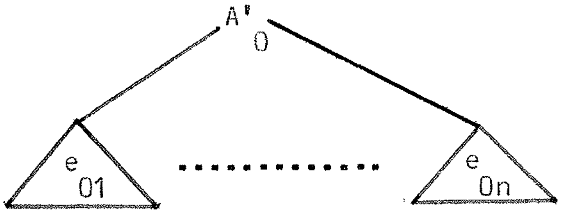
TR will contain the following rule:

$$X_0 \texttt{ -> } A_0 [X_1 \; X_2 \; \ldots \; X_p],$$

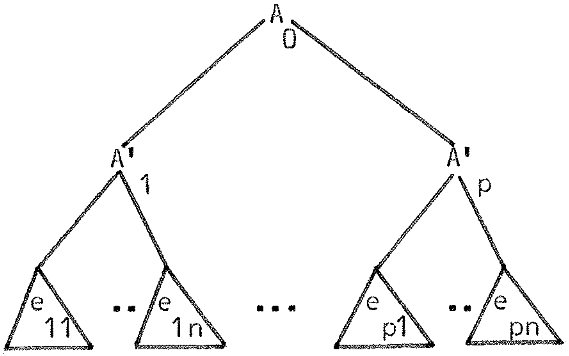
where each X_i ($i \in [0..p]$) has the form

$$A'_i [e_{i1} \; e_{i2} \; \ldots \; e_{in}].$$

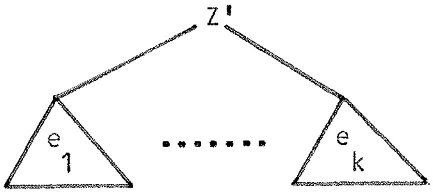
A subtree of the form:



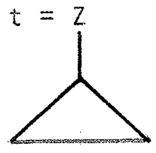
will be transformed into:



If the start symbol Z has k attributes (all synthesised) then TD will transform trees of the form



into trees of the form



where t is parse tree of the underlying CFG of G , provided that

$$<Z\uparrow e_1\uparrow\ldots\uparrow e_k> \; =>*_G \; \mathrm{yield}(t).$$

We have the following theorem:

Theorem 6.2.

Let G be a PEAG with start symbol Z and corresponding top down rewriting system TD, then

w∈ L(G) if and only if

there exist a parse tree t of G with w=yield(t), and

there exists a production rule

$$<Z\uparrow e_1\uparrow e_2\uparrow\ldots\uparrow e_k> \; ::= Z_1\ Z_2 \ldots Z_p$$

such that

$$Z'[e_1\ e_2 \ldots e_p] \; =>*_{TD} \; t$$

//

Next we shall construct a *bottom-up tree rewriting system* that transforms parse trees into their meanings. For a given PEAG, G we construct BV=(W U {[,]}, BR). W is the same as for TD and BR consists of the rules from TD where the left and right sides are exchanged. Thus if

$$X_0 \; -> A_0[X\ X_2 \ldots X_p]$$

is in TR, then

$$A_0[X_1\ X_2 \ldots X_p] -> X_0$$

is in BR. We then have the following theorem which is the converse of theorem 6.2.

Theorem 6.3.

Let G be PEAG with corresponding bottom-up tree rewriting system BV, then

$w \in L(G)$ iff

there exists a derivation tree t of G with w=yield(t) and

there exists a production rule

$<A\uparrow e_1 \uparrow e_2 \uparrow \ldots \uparrow e_k> ::= Z_1\ Z_2\ \ldots\ Z_p$

derivable from R, such that

$t \Rightarrow^*_{BU} Z'[e_1\ e_2\ \ldots\ e_k]$

//

The above formulation of PEAGs within the framework of tree grammars gives a possibility for using the theory of tree grammars and tree automata on PEAGs. An ordinary EAG may be viewed as a PEAG if the attribute expressions are treated as trees. It still remains to be seen whether or not this is of any help for the theory of EAGs. Some attempts in this direction have been taken in [Engelfriet & File 79] where the translations realisable by a subclass of AGs are analysed. In particular AGs with trees or strings as domains are analysed, and the AGs are compared to tree transducers which are a subclass of tree rewriting systems.

Furthermore the practical use of PEAGs and multi-level EAGs has to be investigated.

Presently we find no need to make a formal definition of multi-level EAGs as this is straightforward to do.

7. A General Fast Attribute Grammar Evaluator.

When implementing an evaluator for AGs one is among others concerned with the following problems:

(a) Decide whether or not to test the AG for circularity,

(b) deciding the order of evaluation of the attributes,

(c) the storing of the syntax tree and the attribute values during the evaluation,

(d) the possibility of evaluating some of the attributes during the (context-free) parsing phase,

(e) the possibility of letting the attributes influence the (context-free) parsing (so-called affix-(or attribute) directed parsing [Watt 74b]).

In order to overcome (or ease) some of these problems a number of subclasses of AGs have been defined. These include L-attributed grammars ([Lewis et al. 74]), multipass AGs ([Bochmann 76]), and many more ([Watt 77],[Mayoh 78b]). Affix grammars ([Koster 70]) may also be considered as such a subclass.

A class of general evaluators build the syntax tree and then evaluate the attributes in some order. The efficiency of these evaluators depends on the method used to determine the order of evaluation of the attributes. These evaluators may roughly be divided into two classes:

(1) Static evaluation order: Those where the order of evaluation can be determined from the AG; i.e. the attributes of a symbol X are evaluated in an order which is independent of the subtree below X ([Kennedy & Warren 76], [Saarinen 78]). These AGs are called benign in [Mayoh 78b].

(2) Dynamic evaluation order: Those where the order of evaluation is determined by the parse tree built during the parsing phase ([Lorho 77], [Cohen & Harry 79], [Kennedy & Ramanathan 79]).

In case (1) the order of evaluation may be defined once and for all by some analysis done by the TWS. In case (2) the analysis for determining the order of evaluation must be done for each parse tree. It is obvious that evaluators in class (1) accept a smaller class of AGs than those in class (2).

Less general methods avoid building the parse tree but use a linear representation in form of e.g. a right-parse or left-parse ([Aho & Ullman 72]). Attributes are then assigned values during one or more scans (forward or backwards) of the linear tree. These methods include L-attributed AGs, multipass AGs, The Alternating Semantic Evaluator ([Jazayeri & Walter 75]).

It is well known that the underlying CFG plays an important role here. If the underlying CFG is LL(k) then the construction of a left-parse may coincide with a first left-to-right scan of the linear syntax tree. If the underlying CFG is known to be LR(k) then one can in general only evaluate synthesised attributes during the parsing phase. *Affix-free* LR-grammars ([Watt 77]) are a *large* and *useful* subclass of AGs with LR(k) underlying CFGs. They include all LL(k) grammars and have the same advantages as LL(k)-grammars and so have SD-grammars ([Lewis et al. 74]).

The evaluator presented below (from now on called the *DAG-evaluator*) is based on dynamic evaluation order. The parse tree has to be represented in the form of a right-parse. A DAG is constructed during one left-to-right scan of the right-parse. This DAG represents the dependency-graph of the parse tree and the necessary information for evaluating all attributes. The value of the attributes may be evaluated by a recursive scan of the DAG.

The advantages of the DAG-evaluator are: (1) it works for all AGs, (2) the syntax tree is represented as a right-parse, (3) the DAG is constructed through a single scan of the right-parse, (4) if the underlying CFG is LR(k) then the DAG may be constructed during the parsing phase.

Furthermore the method will also work for circular AGs. The DAG will then contain cycles. The recursive scan may detect these cycles. As mentioned in section 4 it may be meaningful to continue the evaluation. In many cases the DAG itself may be considered as the result of the evaluation. E.g. the AG in ex. 4.5 defines a translation from parse trees to functions (command continuations). In this case the DAG itself may be a suitable representation of the function and such a DAG may have cycles in it.

In many cases it may be desirable to let the TWS check the AG for circularity and not defer it to the evaluator.

The DAG-evaluator is based upon the fact that any AG has an equivalent one using only synthesised attributes. It may in fact be viewed as an implementation based upon the transformation of def. 4.2 The DAGs being constructed by the evaluator may thus be viewed as a representation of the function valued attributes.

The DAG-evaluator is used in NEATS which is a TWS based on EATGs ([Jespersen et al. 78]).

7.1 The DAG-evaluator.

Below we describe the DAG evaluator. It is assumed that a right-parse of the input string has been constructed. During a left-to-right scan of the right-parse a DAG is constructed. In the following description this left-to-right scan is described as a simulation of the parsing phase (as mentioned it may in fact be performed during the parsing if the grammar is e.g. LR(k)).

The evaluator uses three stacks, the ordinary parse stack (PARSE), a stack for storing information about synthesised attributes (SYN), and one for storing information about inherited attributes (INH).

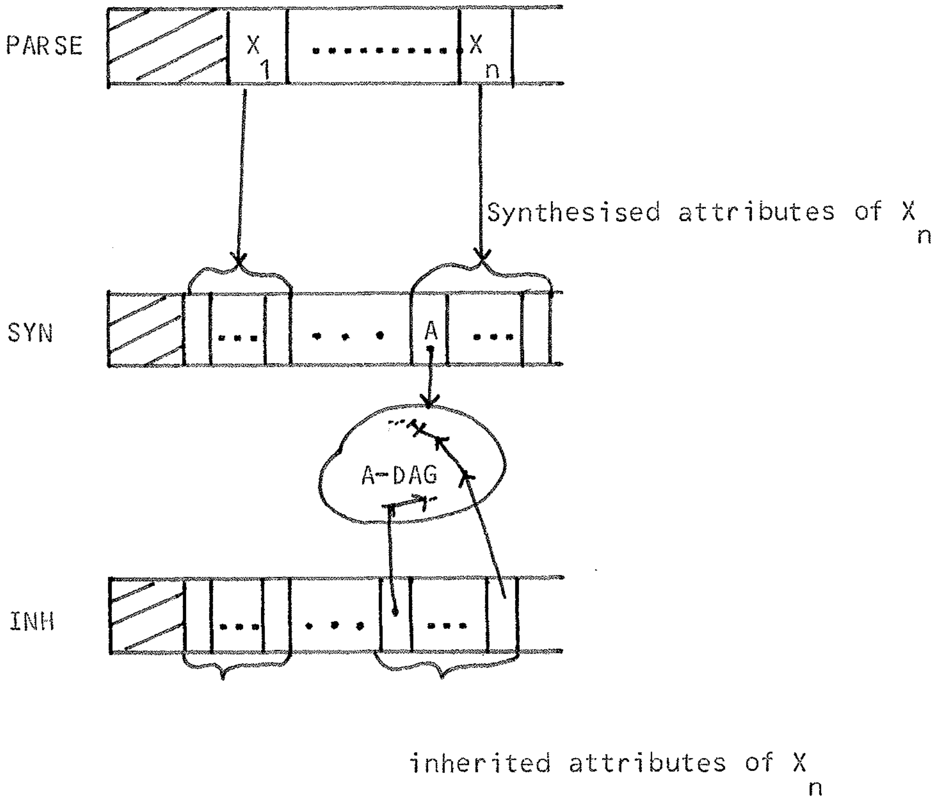
Assume that the parser reduces by a production

$p:\ X_0 \to X_1\ X_2\ \ldots\ X_n$

and that

- SYN contains a value for each right side synthesised attribute of p (called RS[p]). Each value is a pointer to an expression-DAG which when evaluated will yield the value of the corresponding synthesised attribute. Let SA be a synthesised attribute of X_i. Some of the leaves in the DAG for SA will correspond to inherited attributes of X_i, and a value has not yet been filled in for these leaves.
- INH contains a value for each right side inherited attribute of p (RI[p]). Let IA be an inherited attribute of X_i. The value on INH corresponding to IA is a pointer to a list of leaves in the DAGs corresponding to the synthesised attributes of X_i. A value for IA has to be inserted at each such leaf in order to define the synthesised attributes.

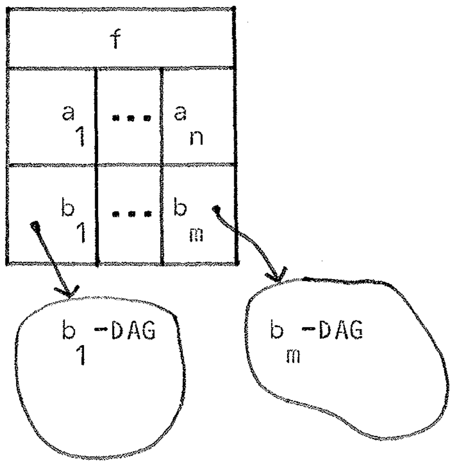
The following picture illustrates the situation:


PARSE
X_1
X_n
Synthesised attributes of X_n
SYN
A
A-DAG
INH
inherited attributes of X_n

When the reduction is applied the following steps are performed:

(1) For each leftside inherited attribute (LI[p]) an empty list is constructed.

(2) For each rightside inherited attribute A (RI[p]) a DAG-node is constructed: The value of A is a function: $f(a_1,\dots,a_n,b_1,\dots,b_m)$ where each $a_i \in$ LI[p] and $b_i \in$ RS[p]. The DAG-node has the form:



The DAG-node has a pointer corresponding to each a_i and b_i. The pointer for b_i points to the DAG for b_i. The pointer field for a_i is chained in the list constructed for a_i in step (1).

The list stored in INH corresponding to A is scanned and each leaf in the list is replaced by a pointer to the newly constructed DAG-node for A.

(3) For each leftside synthesised attribute A in LS[p], a DAG is constructed in the same way as in step (2).

(4) All elements on INH and SYN corresponding to rightside attributes of p are popped off INH and SYN. The values corresponding to leftside attributes of p (the ones constructed in steps (1) and (3)) are pushed on INH and SYN.

Remark. If in steps (2) and (3) n=0 and m=1 and f is the identity function, then a new DAG-node is not constructed. Pointers to the DAG-node for A will instead point to the DAG for b_1.

Traversing the DAG.

When the scan of the right parse is finished the SYN-stack will contain a pointer (a root of the DAG) for each synthesised attribute of the start-symbol.

The values of these attributes may be evaluated by a recursive scan of the DAG starting at each root. Assume that the DAG contains no cycles. By using a so-called 'depth-first-search' of the DAG each node need only be visited once. After the visit of a node a value may be assigned to the node. The next visit to the node may then use this value.

Cycles in the DAG may easily be detected by the depth-first-search algorithm. It will then depend upon the actual AG whether or not a further evaluation is meaningful.

Some domains, like partial maps, may contain 'large' values. In such cases it may not be practical to store a value at each node without using a shared representation. The DAG itself is such a shared representation. By using a sub-DAG as the representation of a value, this sub-DAG may of course be traversed several times during the scan of the DAG.

For common used domains (partial maps) it may be possible to transform the DAG into a more suitable representation.

7.2 Examples.

Example 7.1

Consider the following EAG.

Domain Env: [NAME -> INTEGER]; V: INTEGER; N: NAME;

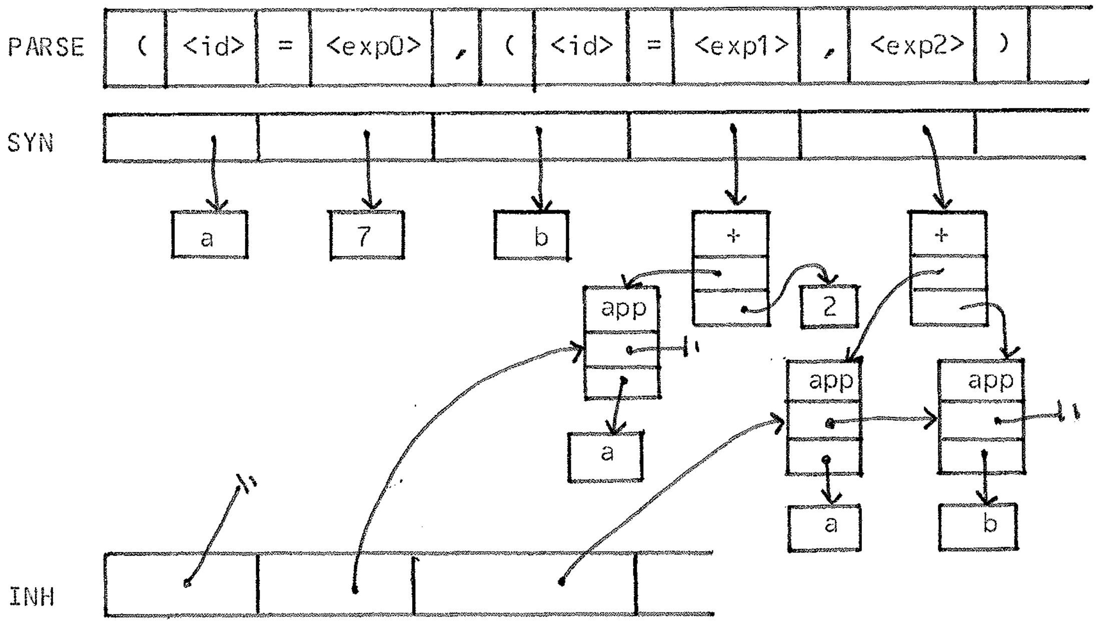
Rules

```
<evaluation↑V> ::= <exp↓{}↑V>

<exp↓Env↑V> ::= ( <id↑N> = <exp↓Env↑V1> , <exp↓Env\{N->V1}↑V> )

<exp↓Env↑V1+V2> ::= <exp↓Env↑V1> + <exp↓Env↑V2>

<exp↓Env↑V> ::= <term↓Env↑V>

<term↓Env↑Env(N)> ::= <id↑N>

<term↓Env↑V> ::= <const↑V>
```

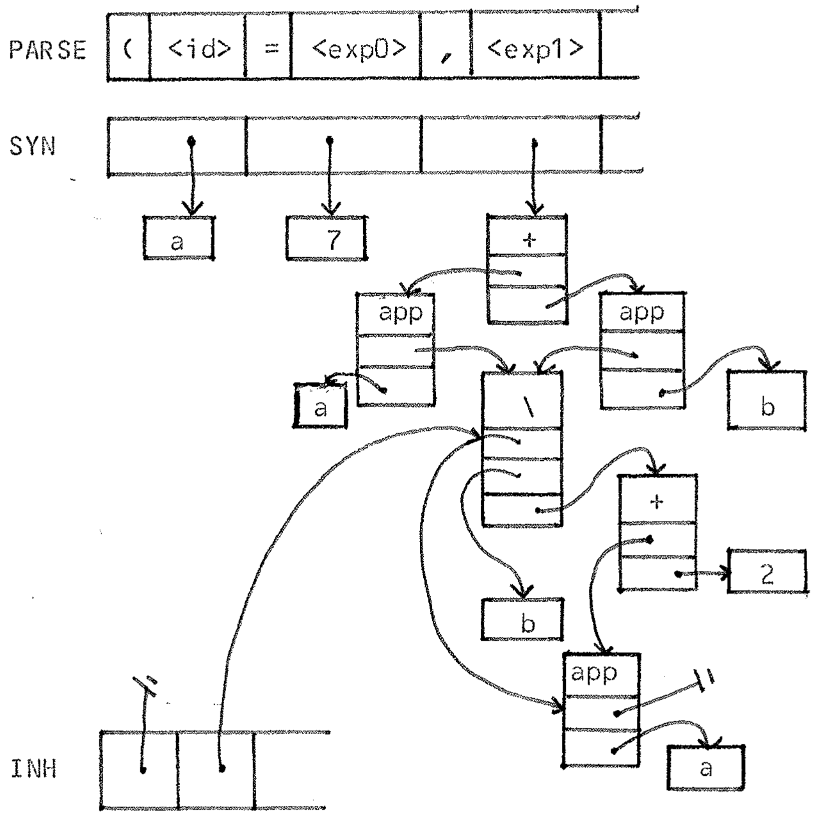
The construct (a=e1,e2) is an abbreviation of LET a=e1 IN e2. Consider a parse of the following string:

```
    (a=7,(b=a+2,a+b))
                    ↑
```

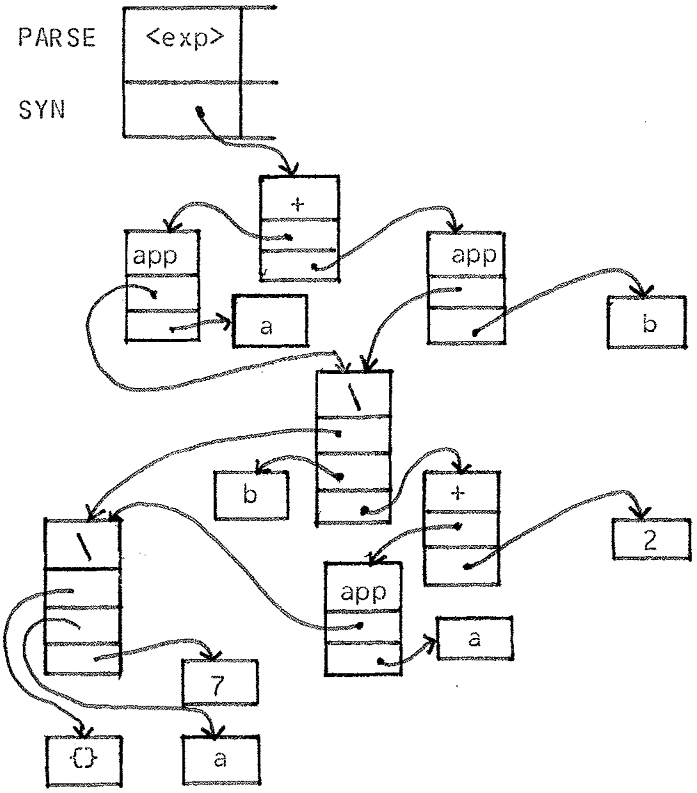
Consider the following snapshot of the stacks when the input is placed at ↑:



After the reduction <exp> ::= (<id>=<exp>,<exp>) we have



The next reduction yields



A reduction of the DAG will yield the value 16.

Example 7.2. Circular AG.

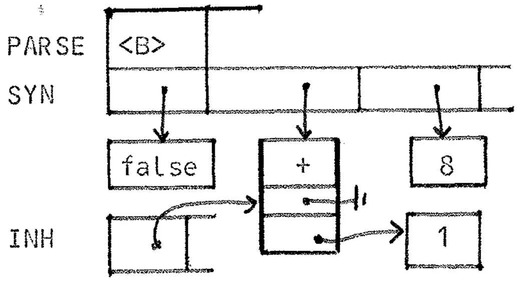
Consider the following EAG

<A↑y+z> ::= <B↓cond(c,y,z)↑c↑y↑z>

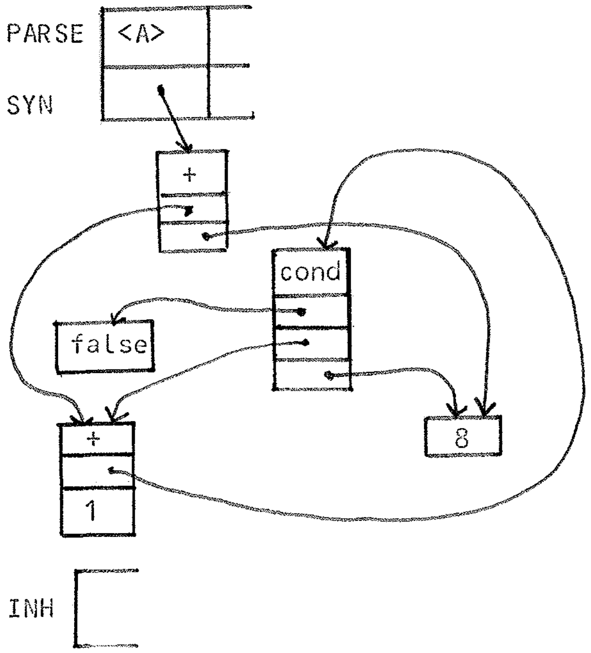
<B↓x↑true↑7↑x+2> ::= a

<B↓x↑false↑x+1↑8> ::= b

Assume that the input string b has been reduced to <B>. We then have the following snapshot:



Next when <B> is reduced to <A> we have:



The constructed DAG is circular, but an evaluation may reduce it to the value 17.

7.3 Evaluation.

It should be obvious that the DAG-evaluator works for any well formed AG in normal form. It is also reasonably fast as it does not need to detect an evaluation order for the attributes in the tree of the parsed string. Furthermore the evaluation of the DAG may be done during one scan of the DAG.

The main problem is that the DAG-evaluator may take up too much space to be practical. We shall compare the space requirements with an evaluator that builds the parse tree and then assigns values to attributes in some order. We assume that at each node there will be space to store the values of the associated attributes (or a pointer to values).

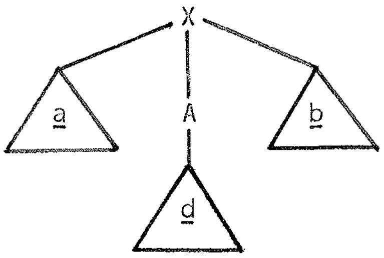
In the DAG-evaluator the situation is as follows:

(1) The parse tree is not stored,

(2) only a subset of the attributes in the parse tree have a corresponding node in the DAG.

C.f. (2): Consider a symbol appearing in the parse tree as part of rules

X -> a A b and A -> d



A synthesised attribute of A will have a DAG-node if the function in the rule A -> d defining the attribute is nontrivial (not the the identity function). Similarly for an inherited attribute of A. Thus attributes that are simple copies of other attributes will not occupy space in the DAG. Notice that the lists for collecting the use of inherited attributes take no additional space as they use the pointer fields that have not yet been filled in.

The DAG-node corresponding to a nontrivial function f may take up more space than the value of the attribute.

The size of the DAG is a linear function of the parse tree. It may be smaller or larger depending on the amount of copying and the format of the nontrivial functions.

It should be noticed that some attribute domains such as partial mappings are of a form where it is unreasonable to store the values at the nodes. These domains must be implemented as data structures with a shared representation. The DAG-representation of such domains may then turn out to be smaller.

In some situations it is possible to reduce the space requirements:

- If A is a synthesised attribute of X and the values of all the inherited attributes of X are known in the DAG for A then the value of A may be evaluated.

- In many situations during parsing, the values of some inherited attributes are in fact known. This is the case if the grammar is L-attributed and the underlying CFG is LL(k). Using a bottom-up parser this information is not directly available. In [Watt 77] an algorithm is given which is able to parse L-attributed AGs having an underlying CFG in a subset of the LR(k)-grammars. This subset includes LL(k) grammars and most practical

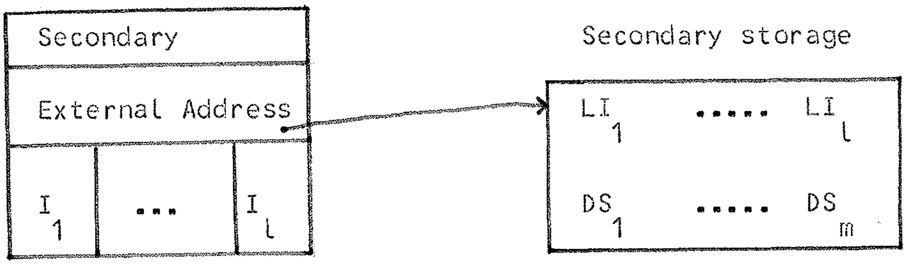
LR-grammars. Watt's algorithm makes use of the fact that for many practical L-attributed AGs most rules simply copy the inherited attributes of the left side. In the rules where the inherited attributes are given 'new' values the parser is often in an LL(k) situation (the LR-table has one item in its closure). Extra nonterminals are inserted in the rules before nonterminals where inherited attributes are given 'new' values. If the grammar is still LR, then the original AG is called affix-free. It is possible to detect automatically where such e-productions may be inserted without destroying LR-properties (introducing conflicts).

The algorithm of section 7.1 may be improved by using Watt's method and computing inherited attributes whereever possible.

Unfortunately this only improves the one pass case and does not help the multipass situation. In general one may use secondary storage to store the DAGs if not enough primary storage is available.

Below we sketch how to introduce secondary storage.

Suppose when A_0 -> A_1 ... A_n is applied we have to move some DAGs to secondary storage. Let I_1, ..., I_l, S_1, ..., S_m be the attributes of A_0. For each S_i we have a DAG DS_i and for each I_i we have a pointer, LI_i to a list of nodes in the DAGs. The DAGs DS_1, ..., DS_m may all be mixed up so we have to move them all to secondary storage. We introduce for each S_i a new node of the form



The block on secondary storage contains the pointers LI_1, ..., LI_i and the DAGs DS_1, ..., DS_m. The new node (marked secondary) contains the external address of the block, and a pointer field corresponding to each inherited attribute I_1, ..., I_l. These pointer fields will then be chained in the lists constructed in step (1) of 7.1.

In general it is not enough to move the DAGs corresponding to A_0, but one may have to move the DAGs of more (all) symbols on PARSE.

When entering a secondary node during the evaluation of the DAG, the corresponding secondary block must be entered into memory. The number of times one has to enter the block depends upon the number of times the corresponding subtree needs to be traversed in order to evaluate the attributes. If a subtree needs m traversals then in the worst case the block must be entered 2*m times; corresponding to m down and m up traversals of the subtree.

The strategy sketched above must be further developed and investigated in order to find out if it is practical. Perhaps it is possible to analyse a given AG in order to find an efficient strategy for storing DAGs on secondary storage.

8. Concluding remarks.

We think that it has been demonstrated that EAGs (and AGs and EATGs) may be used for many different purposes, and that a TWS based upon EATGs may serve many different purposes.

EAGs may also be used as a powerful tool for defining computations not necessarily connected with semantics of programming languages.

Various variants of attribute grammars have been discussed, EAGs, pure EAGs, multi-level EAGs, and AGs with Scott-type domains. Furthermore pure EAGs have been formulated within the theory of tree languages.

The three main approaches are the original AGs of Knuth, AGs with Scott-type domains and EAGs. We tend to prefer the generative approaches or the equational approaches for the Knuth AGs.

It has been demonstrated elsewhere ([Madsen et al.76], [Watt & Madsen 77], and [Watt 79]) that EAGs are suitable for defining the analysis phase (context-sensitive syntax and a translation into an intermediate form) for realistic programming languages.

The purpose of this paper has been to demonstrate that EAGs have a much wider range of applications. The main subject has been to show how three dominating methods for specifying semantics can be expressed within the framework of EAGs.

This makes it possible within the same TWS to experiment with several definitions and implementations. Consistency between different semantic definitions can be proved within the framework of the same formalism. This may turn out to be an advantage although it has not been treated here.

Given a TWS based on EAGs it should be possible to define
(1) The context-sensitive syntax of a programming language, and
(2) the following kinds of translators/compilers

- a verification generator,
- a compiler based on denotational semantics corresponding to the ones generated by SIS ([Mosses 79]),
- a compiler based on abstract interpretation,

- the analysis phase of a (production) compiler, and
- a compiler generating code to an abstract machine.

For some of the approaches to be satisfactory/efficient there are (as mentioned) certain problems to be considered.

First of all an efficient AG evaluator must be available. We think that the DAG-evaluator presented in section 7 is a useful contribution to this. It avoids most of the standard problems concerning the order of evaluation of attributes and it construct the DAG during one left-to-right scan of a right parse of the input string. Furthermore it works for all AGs and does not imply a strange subclass. This is important for the users of a TWS.

The possibilities of letting the attributes influence the parse tree/translation selected for a given string needs a satisfactory solution in order to make full use of EAGs to handle ambiguities in the underlying CFG.

9. References.

[Aho & Ullman 72] A.V. Aho, J.D. Ullman:
The theory of parsing, translation, and compiling. Vol 1: Parsing, 1972. Vol 2: Compiling, 1973. Englewood Cliffs (N.J.): Prentice Hall.

[Björner 78] D. Björner :
The systematic development of a compiling algorithm. In: Le point sur la compilation. (M. Amirchahy, D.Néel, ed.), IRIA, 1978.

[Bochmann 76] G.V. Bochmann:
Semantic evaluation from left to right. Comm. ACM 19, 55-62(1976).

[Chirica 76] L.M. Chirica:
Contributions to compiler correctness. University of California, Los Angeles, Ph. D. Thesis, October 1976.

[Cohen & Harry 79] R. Cohen, E. Harry:
Automatic generation of near-optimal linear-time translators for non-circular attribute grammars. Conference record of the Sixth ACM Symposium on Principles of Programming Languages, San Antonio, January 1979.

[Dahl et al.70] O.-J. Dahl, B. Myrhaug, K. Nygaard:
SIMULA 67, common base language, Norwegian Computing Center, Oslo, 1970.

[Engelfriet 74] J. Engelfriet:
Tree automata and tree grammars. DAIMI FN-10, 1974.

[Engelfriet & Filé 79] J. Engelfriet, G. Filé:
The formal power of one-visit attribute grammars. Twente University of Technology, The Netherlands, mem. no. 286, October 1979.

[Ganzinger 79a] H. Ganzinger:
An approach to the derivation of compiler descriptions from the mathematical semantic concept. GI-9. Jahrestagung, Informatik-Fachbereichte 19, Berlin-Heidelberg-New York, Springer 1979.

[Ganzinger 79b] H. Ganzinger:

Some principles for the development of compiler descriptions from denotational language definitions. Der Technischen Universitat Munchen, (summary), 1979.

[Gerhart 76] S.L. Gerhart:

Proof theory of partial correctness verification systems. SIAM J. Comput., Vol 5., No. 3., September 1976.

[Goguen 77] J.A. Goguen:

Abstract errors for abstract data types. Proceedings IFIP Working Conference on Formal Description of Programming Concepts, St. Andrews, New Brunswick,pp.21.1-21.32, August, 1977.

[Goguen et al. 75] J.A. Goguen, J.W. Thatcher, E.G Wagner, J.B. Wright:

Abstract data types as initial algebras and correctness of data representations. Proceedings, Conference on Computer Graphics, Pattern Recognition and Data Structure, May 1975, pp.89-93.

[Jazayeri & Walter 75] M. Jazayeri, K.G. Walter:

Alternating semantic evaluation. Proceedings of the ACM Annual Conference Minneapolis, Minn., October 1975.

[Jespersen et al. 78] P. Jespersen, M. Madsen, H. Riis:

New extended attribute system (NEATS). DAIMI, 1978.

[Kennedy & Ramanathan 1979] K. Kennedy, J. Ramanathan:

A deterministic Attribute Grammar evaluator based on dynamic sequencing. ACM Toplas, Vol 1, no. 1, July 1979 (142-160).

[Kennedy & Warren 76] K. Kennedy, S.K. Warren:

Automatic generation of efficient evaluators for attribute grammars. Conference record of the Third ACM Symposium on Principles of Programming Languages, January 1976, 32-49.

[Knuth 68] D.E. Knuth:

Semantics of context free languages. Math. Sys. Theory, Vol. 2, No. 2, 1968.

[Koster 79] C.H.A. Koster:
Affix grammars. In ALGOL 68 Implementation (J.F. Peck, ed.) pp. 95-109. Amsterdam: North-Holland 1971.

[Lewis et al. 74] P.M. Lewis, D.J. Rosenkrantz, R.E. Stearns:
Attributed translations. J. Computer and System Sciences 9, 279-307 (1974).

[Lorho 77] B. Lorho:
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.

[Madsen et al. 76] O.L Madsen, B.B. Kristensen, J.Staunstrup:
Use of design criteria for intermediate languages. DAIMI PB-59, August 1976.

[Madsen 79] O.L. Madsen:
An introduction to attribute grammars by examples. Daimi, April 1979.

[Marcotty et al. 76] M. Marcotty, H.F. Ledgard, G.V. Bochmann:
A sampler of formal definitions. ACM, Computing Surveyes, Vol. 8, no. 2, 1976.

[Mayoh 76] B. H. Mayoh:
Verification and compilation in MSL. DAIMI, June 1976, unpublished manuscript.

[Mayoh 78a] B.H. Mayoh:
Data types as functions. DAIMI PB-89, July 1978. (short version in: Math. Foundations of Computer Science 1978. Proceedings, 7th Symposium, Zakopane, Poland, 1978. Springer Lecture Notes in Computer Science, no. 64, Ed. by J. Winkowski)

[Mayoh 78b] B.H. Mayoh:
Attribute grammars and mathematical semantics. DAIMI PB-90, August 1978.

[Mosses 79] P. Mosses:
SIS - Semantics implementation system. Reference manual and user guide. DAIMI MD-30, August 1979.

[Nielsen 75] J.E.G. Nielsen:
JQNS systemet. Et system til automatisk generering af verifiere. DAIMI, 1975, Master thesis (in Danish).

[Saarinen 78] M. Saarinen:
On constructing efficient evaluators for attribute grammars. Automata, Languages and Programming, Fifth Collloquium, Udine, July 1978. Lecture Notes in Computer Science, Berlin-Heidelberg-New York, Springer 1978.

[Tennent 76] R.D. Tennent:
The denotational semantics of programming languages. Comm. ACM, Vol. 19, 8, 437-453(1976).

[Watt 74a] D.A. Watt:
Ananalysis oriented two-level grammars. University of Glasgow, Ph. D. Thesis, 1974.

[Watt 74b] D.A. Watt:
LR-parsing of affix-grammars. University of Glasgow, Report no. 7, 1974.

[Watt 77] D.A. Watt:
The parsing problem for affix-grammars. ACTA Informatica, 8, 1-20(1977).

[Watt 79] D.A. Watt:
An extended attribute grammar for Pascal. Sigplan Notices, Vol. 14, no. 2, 1979.

[Watt & Madsen 77] D.A. Watt, O.L. Madsen:
Extended attribute grammars. University of Glasgow, Report no. 10, July 1977.

[van Wijngaarden et al. 75] A. van Wijngaarden, B. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, R.G. Fisker: Revised report on the algoritmic language ALGOL 68. ACTA Informatica, 5, 1-236(1975).

[Wilner 72] W. Wilner:
Declarative semantic definition. STAN-CS-233-71, Ph. D. Thesis, Stanford, California, 1971.

[Wirth 71] N. Wirth:
The programming language PASCAL. ACTA Informatica 1, 35-63(1971).