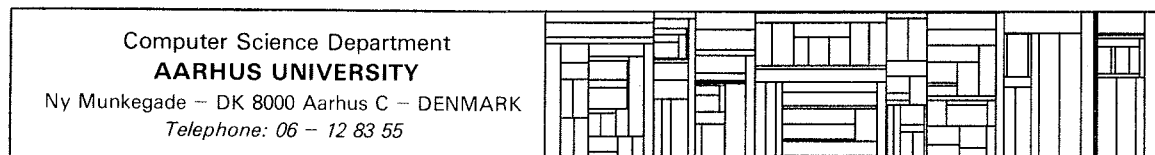


# SUBCLASSES OF ATTRIBUTE GRAMMARS

by  
Hanne Riis

DAIMI PB-114  
March 1980



## ABSTRACT

---

This thesis is a contribution to the development of a formal theory for attribute grammars, their languages and their translations.

There are given precise definitions of an attribute grammar, the language recognized by the attribute grammar and the translation specified by the attribute grammar. The various definitions are compared with some alternative ones. Based on properties of the translation specified by an attribute grammar two new subclasses of attribute grammars are introduced: the determinate and the unambiguous attribute grammars.

Furthermore the concept of an evaluator is considered. Based on properties of of an evaluator for an attribute grammar some new subclasses of attribute grammars are introduced: the  $k$ -visit attribute grammars and the  $k$  left-to-right pass attribute grammars ( $k$  is an integer). It turns out that the  $k$ -visit as well as the  $k$  left-to-right pass attribute grammars define proper hierarchies of translations when some conditions are satisfied. It is also shown that there are translations specified by 1-visit attribute grammars that cannot be specified by any  $k$  left-to-right pass attribute grammar (when some conditions are satisfied). On the other hand it turns out that any well-defined attribute grammar is  $k$ -visit for some  $k$ .

## CONTENTS

ABSTRACT	i
CONTENTS	ii
1. INTRODUCTION	1
1.1 Existing results	2
1.2 Overview	4
1.3 Notation	5
2. ATTRIBUTE GRAMMARS, THEIR LANGUAGES AND TRANSLATIONS	7
2.1 Definition of attribute grammars	8
2.2 Language definition by evaluated semantic trees	12
2.3 Language definition by least fixpoints	16
2.4 Determinate and unambiguous AGs	21
2.5 The formal power of AGs	23
3. WELL-DEFINED ATTRIBUTE GRAMMARS	27
3.1 Dependency graphs	28
3.2 The well-definedness property	32
3.3 Transformations on well-defined AGs	34
3.4 Useless attributes	38
4. K-VISIT ATTRIBUTE GRAMMARS	47
4.1 Definition of an evaluator	48
4.2 The k-visit property	56
4.3 An evaluator for a k-visit AG	60
5. K LEFT-TO-RIGHT PASS ATTRIBUTE GRAMMARS	73
5.1 The k left-to-right pass property	74
5.2 Testing the k left-to-right pass property	77
5.3 An evaluator for a k left-to-right pass AG	84
6. CONCLUSION	94
6.1 Review	94
6.2 Composition and decomposition of AGs	98
6.3 Final comments	99
REFERENCES	101

## 1. INTRODUCTION

Informally, an attribute grammar can be considered as an extension of a context free grammar. To each symbol we associate a fixed number of attributes and to the syntactic rules we associate semantic rules. These rules define some of the attributes of the symbols occurring in a (syntactic) rule in terms of others.

To each node in a derivation tree defined by the context free grammar we can associate attributes similar to those of the corresponding symbol. The semantic rules associated with the productions are used to give values to the attributes for each occurrence of the production in the derivation tree.

There exist several compiler writing systems based on attribute grammars. Some of them are compared in [JMR78]. Several aspects of the systems are discussed in that paper, two of the most important ones are

- how can we determine an order for evaluating the attributes associated with a derivation tree
- what kind of information can be stored in the attributes and what kind of operations are allowed in the semantic rules.

A question that naturally arises is: how important are the various restrictions on the attribute grammars for the translations that can be specified.

It will be desirable to have a theory for attribute grammars which makes it possible to compare various classes of attribute grammars. The classes of attribute grammars may be defined on the basis of how an evaluation order is determined for the attributes associated with a derivation tree, or on the basis of how the domains for the attributes are and how the operations in the semantic rules are.

This thesis is a contribution to such a theory but still much research remains.

The reader is assumed to be familiar with attribute grammars at least to a level corresponding to that of [Knu68], [Knu71] and [Boc76]. Some knowledge of lattice theory is recommended especially in chapter 2 (see e.g. [Man74] or [Sto77]).

ACKNOWLEDGEMENT: I wish to thank Sven Skyum for his advice during my work with this thesis. Furthermore I wish to thank Brian Mayoh and Erik Meineche Schmidt for useful comments. Personal thanks are due to Flemming Nielson.

## 1.1 EXISTING RESULTS

As far as I know the topics outlined above have only been treated by [LRS74] and [EnF79] and here only for very restricted classes of attribute grammars.

In [LRS74] so-called attributed translation grammars are considered. These grammars may be considered as a restricted form of attribute grammars. Each symbol will have associated a synthesized attribute denoting the translation corresponding to the translation of a derivation tree with the symbol as root. If therefore  $p: X ::= X_1 X_2 \dots X_n$  is a production in the attribute grammar then the translation  $b$  of a tree with root  $X$  is obtained by concatenation of the translations  $b_j$  of the subtrees with roots  $X_j$  for  $1 \leq j \leq n$ . More precisely we may have  $b = c_0 \sim b_1 \sim c_1 \sim b_2 \sim \dots \sim c_{n-1} \sim b_n \sim c_n$  where  $c_j$  is a constant value for  $0 \leq j \leq n$  and ' $\sim$ ' denotes string concatenation.

The results presented in [LRS74] hold for these restricted forms of attribute grammars.

An L-attribute grammar is defined as an attribute grammar where all the attributes associated with a derivation tree can be evaluated by a single left-to-right pass over the tree. Furthermore an S-attribute grammar is defined as an attribute grammar where all the attributes of every symbol are synthesized.

Two of the results from [LRS74] may then be stated as:

There is a translation that can be specified by an L-attribute grammar but which cannot be specified by any S-attribute grammar using the same

domains for the attributes and the same operations in the semantic rules.

There is a translation that can be specified by an attribute grammar but which cannot be specified by any L-attribute grammar using the same domains for the attributes and the same operations in the semantic rules.

In [EnF79] another subclass of attribute grammars are considered, the one-visit attribute grammars. This class of attribute grammars may be characterized by that each node is 'visited' at most once when evaluating the attributes associated with a derivation tree. Thus the L-attribute grammars are a subclass of the one-visit attribute grammars. We have the following result

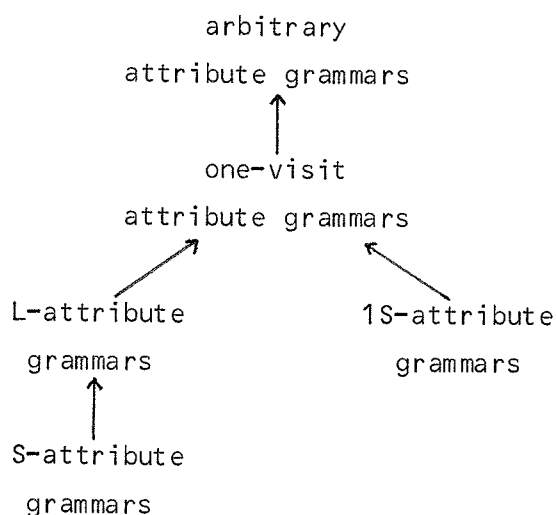
There is a translation that can be specified by a one-visit attribute grammar but which cannot be specified by any L-attribute grammar using the same domains for the attributes and the same operations in the semantic rules.

Also a class of attribute grammars called 1S-attribute grammars is considered by [EnF79]. An attribute grammar in this class is characterized by that each symbol has at most one synthesized attribute but it may have any number of inherited attributes. It turns out that a 1S-attribute grammar is one-visit but

There is a translation that can be specified by a one-visit attribute grammar but which cannot be specified by any 1S-attribute grammar using the same domains for the attributes and the same operations in the semantic rules.

The 1S-attribute grammars and the L-attribute grammars turn out to be incomparable with respect to the translations that can be specified.

Let us summarize the results about translations specified by attribute grammars referred above. An arrow in the following figure represents an inclusion and unconnected classes are incomparable.



The inclusions hold for any fixed choice of domains for the attributes and operations in the semantic rules. Furthermore there is a choice for which the inclusions will be proper.

## 1.2 OVERVIEW

In chapter 2 is given a formal definition of an attribute grammar. There are given several definitions of the language recognized by and the translation specified by the attribute grammar. This is done in order to compare the many different and more or less formal definitions in the literature. Furthermore two new subclasses of attribute grammars are defined, the determinate and the unambiguous attribute grammars.

A subclass of attribute grammars which often is considered is the well-defined attribute grammars. This class is introduced in chapter 3. The language and translation definitions given in chapter 2 are applied to the well-defined attribute grammars and it turns out that most of the definitions coincide. We end up with two approaches to translation definition, a translational and a traditional. Furthermore we introduce a subclass of attribute grammars called the reduced attribute grammars and we show that any well-defined attribute grammar can be transformed into a reduced attribute grammar which specifies the same translation and uses the same domains for the attributes and the same operations in the semantic rules.

In chapter 4 and 5 we consider subclasses of attribute grammars which are extensions of the one-visit attribute grammars of [EnF79] and the L-attribute grammars of [Boc76]. We also introduce the concept of an evaluator. Both in chapter 4 and 5 we give algorithms which may be used to construct evaluators for the respective subclasses.

In chapter 4 the k-visit attribute grammars are defined. It is shown that any well-defined attribute grammar is k-visit for some k and that the k-visit attribute grammars define a proper hierarchy also with respect to translations when some conditions are satisfied.

In chapter 5 we consider a subclass of attribute grammars where the attributes can be evaluated by a fixed number of left-to-right passes over the derivation tree. We show that there are translations specified by one-visit attribute grammars which cannot be specified by any attribute grammar in this class when some conditions are satisfied.

At last in chapter 6 I give my conclusions and some proposals for further research.

### 1.3 NOTATION

In this section I will present some of the notation that will be used.

Let  $Q_1$  and  $Q_2$  be two graphs over the same set of nodes. We define  $Q = Q_1 \sqcup Q_2$  as the graph over the same set of nodes and with an arc from  $b$  to  $c$  if and only if there is an arc from  $b$  to  $c$  in either  $Q_1$  or in  $Q_2$ .

Let  $Q$  be a graph over a set of nodes  $A = A_0 \cup A_1 \cup \dots \cup A_n$  where  $A_j \cap A_i = \emptyset$  for  $j \neq i$ . Let  $Q_j$  be a graph over a set of nodes containing those in  $A_j$  for  $1 \leq j \leq n$ . Then the graph

$$Q' = Q \sqcup Q_1 \sqcup Q_2 \dots \sqcup Q_n$$

will be a graph over the set  $A$  of nodes. There will be an arc from  $b$  to  $c$  in  $Q'$  if and only if there is an arc from  $b$  to  $c$  in either  $Q$  or in  $Q_j$  for some  $j$ ,  $1 \leq j \leq n$ .

If  $Q_0$  is a graph over the set  $A_0$  of nodes then  $Q \sqcup Q_0$  is a graph over the set of nodes  $A$  and with an arc from  $b$  to  $c$  if and only if there is an arc from  $b$  to  $c$  in either  $Q$  or in  $Q_0$ .



Let  $Q$  be a graph. An arc from  $b$  to  $c$  will be denoted by  $b \rightarrow c$ . A path from  $b$  to  $c$  will be denoted  $b \rightarrow^* c$  and if the length is greater than zero then it may be denoted by  $b \rightarrow^+ c$ .

A tree  $t$  with root labelled  $X$  and subtrees  $t_1, t_2, \dots, t_n$  will be denoted  $t = X[t_1 t_2 \dots t_n]$ .

The concatenation of the leaves of a tree is called the yield of the tree.

The height of a tree consisting of a single node is zero. The height of a tree  $t = X[t_1 t_2 \dots t_n]$  is one plus the maximal height of the subtrees.

The empty string will be denoted by  $\lambda$ .

## 2. ATTRIBUTE GRAMMARS, THEIR LANGUAGES AND TRANSLATIONS

There does not seem to exist a generally accepted definition of an attribute grammar. Therefore the purpose of this chapter is to give a formal definition of attribute grammars and to clarify some questions concerning the languages recognized by and the translations specified by attribute grammars.

In section 2.1 is given a definition of an attribute grammar which emphasizes the importance of both the domains for the attributes and the functions used in the semantic rules.

There seems to be some disagreement about how to define the language and the translation specified by an attribute grammar. Two central points seem to be:

1. Do we care about the values of all the attributes associated with a derivation tree or are we only interested in the values of the attributes of the root of the derivation tree.
2. How shall we find the values of the attributes.

Here point 1 leads to a distinction between a 'traditional' and a 'translational' approach. Point 2 results in two language definitions, one in section 2.2 and an other in section 2.3. The first of the definitions requires the existence of a so-called evaluated semantic tree and the second requires a minimal fixpoint for a specific function.

The translation defined by an attribute grammar needs not be unambiguous. Therefore in section 2.4 we introduce the concepts of determinate and unambiguous attribute grammars. It turns out that it is undecidable whether an attribute grammar is determinate or unambiguous.

At last in section 2.5 we investigate the formal power of both the unrestricted, the unambiguous and the determinate attribute grammars. Furthermore we consider some decidability results.

## 2.1 DEFINITION OF ATTRIBUTE GRAMMARS

In many definitions of attribute grammars the domains for the attributes and the functions used in the semantic rules are just something that exist. But in fact the power of the formalism is strongly dependent on the choice of domains for the attributes and the operations on them.

In order to emphasize the importance of the domains and the operations I first introduce a so-called semantic domain and later I will define an attribute grammar over a semantic domain. These definitions are inspired by [EnF79].

### DEFINITION 1:

A semantic domain is a pair  $(\underline{D}, \underline{F})$  where

- $\underline{D}$  is a set of complete lattices
- $\underline{F}$  is a collection of total, continuous and recursive functions of functionality  $f: D_1 \times D_2 \times \dots \times D_m \rightarrow D$  with  $m \geq 0$  and where  $D, D_1, \dots, D_m$  are complete lattices from  $\underline{D}$ .

///

The bottom element of a complete lattice  $D$  will be denoted  $\perp$  and the top element  $\top$ . If an attribute has the value  $\perp$  it means that its value has not been determined. The value  $\top$  is an error value.

### EXAMPLE 1:

The set INTEGER of integers can be extended to a flat lattice INTEGER' by addition of a top and a bottom as described by e.g. [Sto77]. The set BOOLEAN = {true, false} can be extended in a similar manner to the lattice BOOLEAN'.

Consider the function

cond: BOOLEAN X INTEGER X INTEGER  $\rightarrow$  INTEGER

defined by

cond(true, a, b) = a

cond(false, a, b) = b

for  $a, b \in$  INTEGER.

The function can be extended to a function cond' operating on the complete lattices:

cond': BOOLEAN' X INTEGER' X INTEGER'  $\rightarrow$  INTEGER'

by letting

$$\begin{aligned} \text{cond}'(\underline{?}, a, b) &= \underline{?} & \text{cond}'(\underline{!}, a, b) &= \underline{!} \\ \text{cond}'(\text{true}, a, \underline{?}) &= a & \text{cond}'(\text{true}, a, \underline{!}) &= a \\ \text{cond}'(\text{false}, \underline{?}, b) &= b & \text{cond}'(\text{false}, \underline{!}, b) &= b \end{aligned}$$

where  $a, b \in \text{INTEGER}'$  and otherwise  $\text{cond}'(c, a, b) = \text{cond}(c, a, b)$  where  $c \in \text{BOOLEAN}'$ .

In a similar manner we can extend the identity function and the 'addition-by-one' function to operate on the complete lattices. If  $\underline{F}$  consists of these tree functions and  $\underline{D} = \{\text{INTEGER}', \text{BOOLEAN}'\}$  then  $(\underline{D}, \underline{F})$  will be a semantic domain.

///

DEFINITION 2:

An attribute grammar (AG)  $G$  over a semantic domain  $(\underline{D}, \underline{F})$  is a 4-tuple:

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

where

$V$ : a finite set of symbols.  $V$  is separated into two disjoint sets:  $V_n$ , the set of nonterminal symbols, and  $V_t$ , the set of terminal symbols.

To each symbol  $X$  in  $V$  is associated a fixed set  $A(X)$  denoted the attributes of  $X$ . Each attribute is associated with a lattice from  $\underline{D}$ , and the value of the attribute is a member of the lattice.

An attribute can be either inherited or synthesized. The set of inherited attributes for a symbol  $X$  in  $V$  is denoted  $I(X)$  and the set of synthesized  $S(X)$ . (We shall require that  $I(X) \cap S(X) = \emptyset$  and  $I(X) \cup S(X) = A(X)$ .)

For each synthesized attribute of a terminal symbol there is given an external semantic rule which defines its value.

$B$ : a set of attribute variables used in  $R$ . Any attribute variable denotes an element from a (fixed) lattice in  $\underline{D}$ .

$R$ : a set of productions. Any production consists of a rule from a reduced context free grammar (CFG)

$$G_u = (V_n, V_t, R_u, Z)$$

together with some semantic rules.  $G_u$  is called the underlying grammar of the AG.

If in  $R_u$  we have a rule  $p: X ::= X_1 X_2 \dots X_n$  then the values of

the inherited attributes of  $X$  and the synthesized attributes of  $X_j$  for  $1 \leq j \leq n$  (called the defining attributes in  $p$ ) will be denoted by distinct attribute variables associated with the corresponding lattices for the attributes. The values of the synthesized attributes of  $X$  and the inherited attributes of  $X_j$  for  $1 \leq j \leq n$  (called the applied attributes in  $p$ ) will be denoted by semantic rules.

A semantic rule is a function from  $\underline{F}$  whose parameters are either attribute variables denoting defining attributes in  $p$  or they are constant values from the appropriate lattices.

$Z$ : the start symbol.  $Z$  is in  $V$  and  $I(Z) = \emptyset$ .

///

We use a BNF-like notation for the productions. Behind each symbol we write its attributes - an up-wards arrow ( $\uparrow$ ) prefixes a synthesized attribute, a down-wards arrow ( $\downarrow$ ) prefixes an inherited attribute. We will assume that there is a fixed ordering of the attributes of each symbol.

EXAMPLE 2:

We will now define an AG  $G = (V, B, R, Z)$  over the semantic domain  $(\underline{D}, \underline{F})$  from example 1.

$V: V_n = \{X, Y, Z\}, V_t = \{A, A_1, A_2, A_3, A_4\}$   
 $I(X) = \{x_1, x_2\} \quad S(X) = \{x_3\}$   
 $I(Y) = \{y_1\} \quad S(Y) = \{y_2\}$   
 $I(Z) = \emptyset \quad S(Z) = \{z_1\}$   
 $I(A) = \emptyset \quad S(A) = \{a\}$   
 $I(A_j) = \emptyset \quad S(A_j) = \emptyset \quad \text{for } 1 \leq j \leq 4$

where  $x_1, a$ : BOOLEAN' and  $x_2, x_3, y_1, y_2, z_1$ : INTEGER'.

$a$  is defined by an external semantic rule to be either true or false.

$B: a$ : BOOLEAN' and  $b$ : INTEGER'.

$R: p_1: \langle Z \uparrow 1 \rangle ::= A_1 \langle A \uparrow a \rangle$   
 $p_2: \langle Z \uparrow b \rangle ::= \langle X \downarrow a \downarrow \text{cond}'(a, 0, b) \uparrow b \rangle \langle A \uparrow a \rangle$   
 $p_3: \langle X \downarrow a \downarrow b \uparrow \text{cond}'(a, b, 1) \rangle ::= A_2$   
 $p_4: \langle Z \uparrow b \rangle ::= A_3 \langle Y \downarrow b \uparrow b \rangle$   
 $p_5: \langle Y \downarrow b \uparrow b \rangle ::= \langle A \uparrow a \rangle$   
 $p_6: \langle Z \uparrow 0 \rangle ::= A_4 \langle Y \downarrow (b+1) \uparrow b \rangle$

Z: is the start symbol

///

We have already mentioned the most important difference between the definition given above and other definitions of AGs, namely the introduction of a semantic domain as a part of an AG. There are however some minor differences.

In most definitions of an AG there are restrictions on the type of attributes (inherited and/or synthesized) allowed to be associated with a terminal symbol X. There are at least the following possibilities:

- i) no restrictions on I(X) and S(X)
- ii)  $I(X) = \emptyset$  and no restrictions on S(X)
- iii)  $S(X) = \emptyset$  and no restrictions on I(X)
- iv)  $I(X) = S(X) = \emptyset$

These alternatives are discussed by [Räi77]. I have chosen to allow inherited attributes for terminal symbols as they do not impose further problems. Also synthesized attributes are allowed, but they are defined by external semantic rules (i.e. by the lexical analysis of a compiler). They correspond to the intrinsic attributes defined by [Sch76]. The notion 'external semantic rule' is due to [Tie79]. We will assume that the external semantic rules define the values for the synthesized attributes independent of the values for the inherited attributes of the symbol.

In some but not all definitions of AGs the start symbol Z is allowed to have inherited attributes. There are advantages by allowing inherited attributes (defined by external semantic rules). For instance it becomes conceptually cleaner to consider subgrammars of AGs.

But by allowing inherited attributes for Z it becomes more difficult to handle the AGs in a theoretical environment as we will do - and therefore I refrain from allowing them.

In some definitions of AGs so-called global and local attributes are introduced ([Räi77], [GRW77], and [Poz79]). Global attributes are special attributes associated with the start symbol of an AG. Their value may be changed and accessed by semantic rules of any production. But they seem to make the understanding of an AG more complicated.

A local attribute is an attribute whose value only is used in one production. It may be an advantage to have local attributes associated with not the symbols (as in [Poz79]) but with the productions (i.e. acting as local variables). They may make it easier to write and to understand AG specifications, since for example common subexpressions may be extracted or complex tests may be split into smaller pieces.

But in order not to make the definition of an AG more complicated than necessary I refrain from introducing global and local attributes.

At last I will comment on the parameters of the semantic rules. In my definition an applied attribute will always depend on defining attributes exclusively. In Knuth's original definition this restriction is not imposed. I introduce only AGs in normal form ([Boc76]). By doing so some AGs are avoided, namely those, where an attribute is defined recursively by itself in one production. Incidentally, the BNF-like notation used in the examples is only possible for AGs in normal form.

## 2.2 LANGUAGE DEFINITION BY EVALUATED SEMANTIC TREES

In order to define the language recognized by an AG  $G$  we review the definition of a derivation tree for a CFG  $G_u = (V_n, V_t, R_u, Z)$ . Let  $DOM(X)$  be the set of derivation trees with root labelled  $X$  where  $X$  is a symbol in  $V$ .  $DOM(X)$  is defined recursively as:

- i) if  $X \in V_t$  then  $DOM(X) = \{X\}$
- ii) if  $X \in V_n$  and  $p: X ::= X_1 X_2 \dots X_n$  ( $n > 0$ ) is in  $R$  then  $X[t_1 t_2 \dots t_n]$  is in  $DOM(X)$  where  $t_j \in DOM(X_j)$  for  $1 \leq j \leq n$ .
- iii) if  $X \in V_n$  and  $p: X ::= \lambda$  is in  $R_u$  then  $X[\lambda]$  is in  $DOM(X)$ .

Often we will not distinguish between a node and its label.

To each node in a derivation tree we associate attributes corresponding to those of the equivalent symbol.

Let  $X$  be a node in a derivation tree  $t$ . If  $p: X ::= X_1 X_2 \dots X_n$  is the production used to expand  $X$  in  $t$  then the semantic rules for the synthesized attributes of  $X$  given in  $p$  are used to give values to the corresponding attributes in  $t$ . If  $X$  occurs on the right hand side of a production  $q: Y ::= Y_1 Y_2 \dots Y_m$  and  $X$  is introduced (in  $t$ ) by an application of

q then the semantic rules for the inherited attributes of X given in q are used to give values to the corresponding attributes in t.

We will now give some definitions making that of giving values to the attributes associated with a derivation tree more formal. It is achieved in two steps. First the nodes in a derivation tree are extended to hold the values for the attributes. These trees are called semantic trees. In the second step we put restrictions on the values of the attributes in the semantic tree: the relations between the attributes defined by the semantic rules of the AG must hold. This subset of the semantic trees is called the evaluated semantic trees.

**DEFINITION 3:**

A semantic tree for a string w is a tree t satisfying:

- i) each node in t has a label of the form  $\langle X, v \rangle$  where X is in V,  $A(X) = \{a_1, a_2, \dots, a_m\}$ , and  $v = (v_1, v_2, \dots, v_m)$  satisfies that  $v_j$  is in the lattice for  $a_j$  for  $1 \leq j \leq m$
- ii) if h is the homomorphism defined by
  - $h(\langle X, v \rangle) = X$  for X in Vt
  - $h(\langle X, v \rangle [t_1 t_2 \dots t_n]) = X[h(t_1) h(t_2) \dots h(t_n)]$

then h(t) is a derivation tree for w according to the grammar G<sub>u</sub>.

///

**DEFINITION 4:**

An evaluated semantic tree for a string w is a semantic tree t for w satisfying:

- Let  $\langle X, v \rangle$  be a node in t with n subtrees with roots labelled by resp.  $\langle X_1, v_1 \rangle, \langle X_2, v_2 \rangle, \dots, \langle X_n, v_n \rangle$ . Then  $p: X ::= X_1 X_2 \dots X_n$  is a production in R<sub>u</sub> and if a semantic rule associated with p defines an attribute b and uses the attributes  $b_1, b_2, \dots, b_m$  as parameters then the actual values for b,  $b_1, b_2, \dots, b_m$  defined by v,  $v_1, v_2, \dots, v_n$  must satisfy the relation defined by the semantic rule.
- If X is a terminal symbol and  $\langle X, v \rangle$  is a node in t then all the components of v corresponding to synthesized attributes of X have values determined by the external semantic rules.

///

Originally Knuth defined the meaning of a string in the language recognized by the underlying grammar as the values of the synthesized attributes associated with the root of a derivation tree for the string



([Knu68]). Later on the meaning has some times been defined as the whole derivation tree decorated with values for the attributes.

These two approaches give rise to two different views on languages and translations defined by AGs. The two approaches have been named resp. the translational and the traditional approach by [EnF79]. In the translational approach the meaning of a string is the values of the attributes at the root of an evaluated semantic tree; in the traditional approach it is the whole evaluated semantic tree.

DEFINITION 5:

Let  $G$  be an AG over a semantic domain  $\mathcal{D} = (\underline{D}, \underline{F})$ . The meaning of a string  $w$  is defined in two ways:

$$\text{meaning-}e(w, G, \mathcal{D}) = \{v \mid \langle Z, v \rangle \text{ is the root in an evaluated semantic tree for } w\}$$

$$\text{meaning}'-e(w, G, \mathcal{D}) = \{t \mid t \text{ is an evaluated semantic tree for } w\}$$

///

If  $G$  and  $\mathcal{D}$  are obviously known from the context then we will often write  $\text{meaning-}e(w)$  in stead of  $\text{meaning-}e(w, G, \mathcal{D})$ . The same convention is adopted for  $\text{meaning}'-e(w, G, \mathcal{D})$ . The suffix 'e' refers to that the meaning is defined on the basis of an evaluated semantic tree.

The two approaches result in two definitions of the language recognized by an AG.

DEFINITION 6:

The language  $\underline{L-}e(G, \mathcal{D})$  recognized by an AG  $G$  over a semantic domain  $\mathcal{D} = (\underline{D}, \underline{F})$  is defined as

$$\underline{L-}e(G, \mathcal{D}) = \{w \in \underline{L}(Gu) \mid \text{there is a } v \in \text{meaning-}e(w), v = (v_1, v_2, \dots, v_m) \text{ such that } v_j \neq ? \text{ and } v_j \neq \perp \text{ for } 1 \leq j \leq m\}$$

The language  $\underline{L}'-e(G, \mathcal{D})$  recognized by an AG  $G$  over a semantic domain  $\mathcal{D} = (\underline{D}, \underline{F})$  is defined as

$$\underline{L}'-e(G, \mathcal{D}) = \{w \in \underline{L}(Gu) \mid \text{there is a } t \in \text{meaning}'-e(w) \text{ such that if } \langle X, (v_1, v_2, \dots, v_m) \rangle \text{ is a node in } t \text{ then } v_j \neq ? \text{ and } v_j \neq \perp \text{ for } 1 \leq j \leq m\}$$

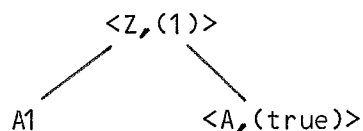
///

EXAMPLE 3:

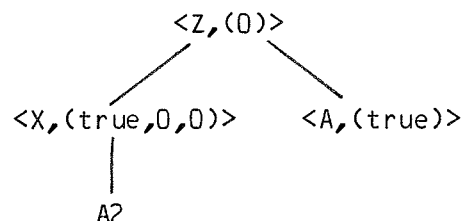
Consider the AG from example 2. Then we may have the evaluated seman-

tic trees (the values of the synthesized attributes of the terminal symbols are given in brackets):

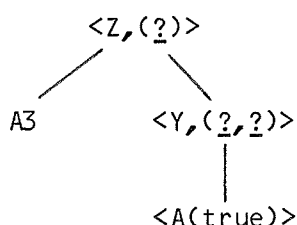
for A1 A(true):



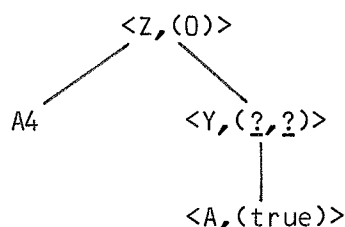
for A2 A(true):



for A3 A(true) (among others):



for A4 A(true) (among others):



It is not difficult to show that

$$\text{meaning-e}(A1 \ A(true)) = \{1\}$$

$$\text{meaning-e}(A2 \ A(true)) = \{0\}$$

$$\text{meaning-e}(A3 \ A(true)) = \text{INTEGER}^1$$

$$\text{meaning-e}(A4 \ A(true)) = \{0\}$$

We have

$$\underline{L}\text{-e}(G, \mathbb{D}) = \{A1 \ A(true), \ A2 \ A(true), \ A3 \ A(true), \ A4 \ A(true), \\ A1 \ A(false), \ A2 \ A(false), \ A3 \ A(false), \ A4 \ A(false)\}$$

$$\underline{L}'\text{-e}(G, \mathbb{D}) = \{A1 \ A(true), \ A2 \ A(true), \ A3 \ A(true), \\ A1 \ A(false), \ A2 \ A(false), \ A3 \ A(false)\}$$

Thus we see that the two languages may be different.

///

If we in the definition of the language recognized by an AG care about the actual values of the meaning of the accepted strings then we obtain the following alternative definition:

$$\underline{L}\text{-e}(P, G, \mathbb{D}) = \{w \in \underline{L}(G_u) \mid P(\text{meaning-e}(w))\}$$

where P is a (total, recursive and monotonic) predicate operating on the complete lattices of  $\underline{D}$  and giving a value in the complete lattice

{true,false} with the ordering 'true > false'. This definition may be useful if context sensitive tests are given in the attributes. Note that not very much is said about the complexity of the predicate P and that we therefore have the same 'problems' as in connection with affix-grammars ([Wat74], [Mad75]) where a great deal of the analysis of a string can be made by a predicate.

The translational and the traditional approach result in two different definitions of the translation specified by an AG.

DEFINITION 7:

The translation  $\underline{I}\text{-e}(G, \mathcal{D})$  specified by an AG G over a semantic domain  $\mathcal{D} = (\underline{D}, \underline{E})$  is defined as

$$\underline{I}\text{-e}(G, \mathcal{D}) = \{(w, v) \mid w \in \underline{L}\text{-e}(G, \mathcal{D}) \text{ and } v \in \text{meaning}\text{-e}(w), \text{ no components of } v \text{ is equal to } ? \text{ or } !\}$$

The translation  $\underline{I}'\text{-e}(G, \mathcal{D})$  specified by an AG G over a semantic domain  $\mathcal{D} = (\underline{D}, \underline{E})$  is defined as

$$\underline{I}'\text{-e}(G, \mathcal{D}) = \{(w, t) \mid w \in \underline{L}'\text{-e}(G, \mathcal{D}) \text{ and } t \in \text{meaning}'\text{-e}(w), \text{ no components of nodes in } t \text{ is equal to } ? \text{ or } !\}$$

///

From our point of view the translational approach is the most natural one but it is often theoretical easier to handle the traditional approach.

The definitions of languages and translations defined in this section is only of theoretical interest, the reason being the lack of an algorithm that can be used to find the evaluated semantic trees. Therefore in the next section we present another set of definitions.

### 2.3 LANGUAGE DEFINITION BY LEAST FIXPOINTS

Consider an AG G and a tree  $t \in \text{DOM}(Z)$ . We can consider the semantic rules defining the attributes associated with the nodes of X as a set of equations that must be satisfied by the values of the attributes. A solution to these equations is an evaluated semantic tree. In stead of being interested in all possible evaluated semantic trees for t we may be interested in the least one. In this section we present definitions based on these special evaluated semantic trees (inspired by [Jon79]).

We will define a function F which transforms one semantic tree into

another. The least fixpoint of that function will be the evaluated semantic tree we are searching for.

DEFINITION 8:

Let  $t$  be a semantic tree for  $w$ . Then  $F(t)$  is the semantic tree for  $w$  defined by

- $h(t) = h(F(t))$  where  $h$  is the homomorphism given in definition 3
- if  $\langle X, v \rangle$  is a node in  $t$  with sons  $\langle X_1, v_1 \rangle, \langle X_2, v_2 \rangle, \dots, \langle X_n, v_n \rangle$  then the corresponding nodes  $\langle X, u \rangle, \langle X_1, u_1 \rangle, \langle X_2, u_2 \rangle, \dots, \langle X_n, u_n \rangle$  in  $F(t)$  will satisfy that if  $b$  (an applied attribute) is defined by a semantic rule  $f(b_1, b_2, \dots, b_m)$  in  $p$  then the value of  $b$  in  $u$  (resp.  $u_k$  for some  $k$ ) is determined on the basis of the values of  $b_1, b_2, \dots, b_m$  in  $v, v_1, v_2, \dots, v_n$ .

///

The existence of a minimal fixpoint for  $F$  follows from the continuity of the functions in the semantic domain and thereby of  $F$ . In order to find the wanted fixpoint we introduce the concept of an initial semantic tree:

DEFINITION 9:

An initial semantic tree for a string  $w$  is a semantic tree  $t$  for  $w$  satisfying:

- if  $\langle X, v \rangle$  is a node in  $t$  and  $X \in V_n$  then all components of  $v$  are equal to  $\underline{?}$
- if  $\langle X, v \rangle$  is a node in  $t$  and  $X \in V_t$  then all components of  $v$  corresponding to inherited attributes are equal to  $\underline{?}$  and all components corresponding to synthesized attributes have values determined by the external semantic rules.

///

We can note that if the external semantic rules define the attributes of the terminal symbols uniquely then an initial semantic tree for a string can be uniquely determined for each derivation tree for the string.

In the previous section we defined the meaning of a string in two different ways corresponding to the two approaches translational and traditional. We also had two definitions of the language recognized by an AG. We have similar definitions here.

DEFINITION 10:

Let  $G$  be an AG over a semantic domain  $\mathcal{D} = (\underline{D}, \underline{F})$ . The meaning of a string  $w$  is defined in two ways:

$$\text{meaning}(w, G, \mathcal{D}) = \{v \mid \langle Z, v \rangle \text{ is the root of } F^*(t) \text{ where } t \text{ is an initial semantic tree for } w\}$$

$$\text{meaning}'(w, G, \mathcal{D}) = \{t \mid t = F^*(t') \text{ where } t' \text{ is an initial semantic tree for } w\}$$

///

The continuity of  $F$  ensures that  $F^*(t')$  is the least fixpoint of  $F$  stronger than  $t'$  where  $t'$  is an initial semantic tree.

DEFINITION 11:

The language recognized by an AG  $G$  over a semantic domain  $\mathcal{D} = (\underline{D}, \underline{F})$  is defined by

$$\underline{L}(G, \mathcal{D}) = \{w \in \underline{L}(Gu) \mid \text{there is a } v \in \text{meaning}(w), v = (v_1, v_2, \dots, v_m) \text{ such that } v_j \neq ? \text{ and } v_j \neq ! \text{ for } 1 \leq j \leq m\}$$

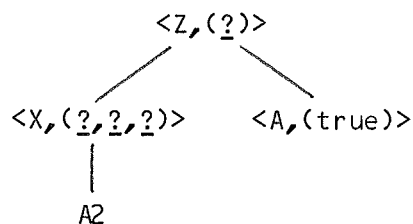
The language  $\underline{L}'(G, \mathcal{D})$  recognized by an AG  $G$  over a semantic domain  $\mathcal{D} = (\underline{D}, \underline{F})$  is defined by

$$\underline{L}'(G, \mathcal{D}) = \{w \in \underline{L}(Gu) \mid \text{there is a } t \in \text{meaning}'(w), \text{ such that if } \langle X, (v_1, v_2, \dots, v_m) \rangle \text{ is a node in } t \text{ then } v_j \neq ? \text{ and } v_j \neq ! \text{ for } 1 \leq j \leq m\}$$

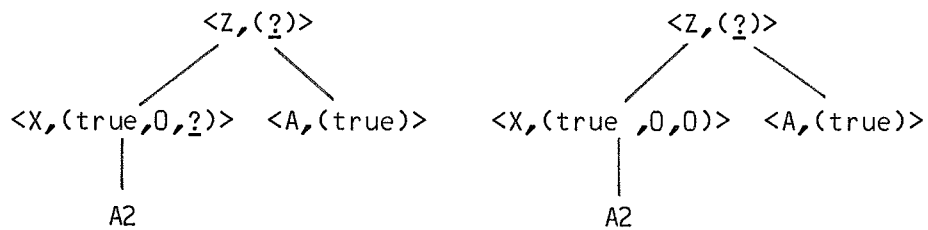
///

EXAMPLE 4:

Let  $G$  be the AG from example 2. The semantic trees given in example 3 are the least fixpoints of the various  $F$ -functions. Consider for instance the string  $A2$   $A(\text{true})$ . The initial semantic tree for the string is:



After application of  $F$  one resp. two times we have these semantic trees:



By applying F once more we get the evaluated semantic tree of example 3.

We have

$$\text{meaning}(A1 \ A(\text{true})) = \{1\}$$

$$\text{meaning}(A2 \ A(\text{true})) = \{0\}$$

$$\text{meaning}(A3 \ A(\text{true})) = \{?\}$$

$$\text{meaning}(A4 \ A(\text{true})) = \{0\}$$

and

$$\underline{L}(G, \mathcal{D}) = \{A1 \ A(\text{true}), \ A2 \ A(\text{true}), \ A4 \ A(\text{true}), \\ A1 \ A(\text{false}), \ A2 \ A(\text{false}), \ A4 \ A(\text{false})\}$$

$$\underline{L}'(G, \mathcal{D}) = \{A1 \ A(\text{true}), \ A2 \ A(\text{true}), \ A1 \ A(\text{false}), \ A2 \ A(\text{false})\}$$

///

#### EXAMPLE 5:

The language recognized by an AG depends very much on the semantic domain. We could have extended the function cond in example 1 in another way:

Let  $\mathcal{D}' = (\underline{D}, \underline{F}')$  be the semantic domain from example 1 where  $\underline{F}'$  is as  $\underline{F}$  except that cond is extended in another way:

$$\text{cond}': \text{BOOLEAN}' \times \text{INTEGER}' \times \text{INTEGER}' \rightarrow \text{INTEGER}'$$

where

$$\text{cond}'(\underline{?}, a, b) = \underline{?}$$

$$\text{cond}'(\text{true}, a, \underline{?}) = \underline{?}$$

$$\text{cond}'(\text{false}, \underline{?}, b) = \underline{?}$$

independent of the values of a and b, and

$$\text{cond}'(\underline{!}, a, b) = \underline{!}$$

$$\text{cond}'(\text{true}, a, \underline{!}) = \underline{!}$$

$$\text{cond}'(\text{false}, \underline{!}, b) = \underline{!}$$

where a and b are arbitrary values different from  $\underline{?}$ .

If the AG G from example 2 is over the semantic domain  $\mathcal{D}'$  then we have

$$\text{meaning}(A1 \ A(\text{true})) = \{1\}$$

meaning(A2 A(true)) = {?}

meaning(A3 A(true)) = {?}

meaning(A4 A(true)) = {0}

and

$\underline{L}(G, \mathcal{D}) = \{A1 A(true), A4 A(true), A1 A(false), A4 A(false)\}$

$\underline{L}'(G, \mathcal{D}) = \{A1 A(true), A1 A(false)\}$

The difference between the AG in example 2 and here corresponds to that between call-by-name and call-by-value evaluation of the semantic rules.

///

The translations specified by an AG can be defined in a manner very similar to that in definition 7.

In the rest of this section we will compare the two languages  $\underline{L}\text{-e}(G, \mathcal{D})$  and  $\underline{L}(G, \mathcal{D})$ . Similar results can be obtained for the two languages  $\underline{L}'\text{-e}(G, \mathcal{D})$  and  $\underline{L}'(G, \mathcal{D})$ .

THEOREM 1:

For any AG  $G$  over a semantic domain  $\mathcal{D} = (\underline{D}, \underline{F})$  we have  $\underline{L}(G, \mathcal{D}) \subseteq \underline{L}\text{-e}(G, \mathcal{D})$

PROOF: The theorem obviously follows since the least fixpoint of  $F$  is an evaluated semantic tree.

///

THEOREM 2:

There exists an AG  $G$  over a semantic domain  $\mathcal{D} = (\underline{D}, \underline{F})$  such that  $\underline{L}(G, \mathcal{D}) \neq \underline{L}\text{-e}(G, \mathcal{D})$ .

PROOF: Consider the AG  $G$  from example 1 and 2, and assume that  $A1$ ,  $A2$ ,  $A3$ , and  $A4$  are different. Then  $\underline{L}\text{-e}(G, \mathcal{D})$  and  $\underline{L}(G, \mathcal{D})$  are different.

///

The definition of meaning in section 2.2 may be considered as that of finding all possible fixpoints of the function  $F$ . If one only is interested in the minimal fixpoint of  $F$  then it is possible to construct algorithms that find it ([Man74]). The problem to find algorithms for the general AGs will not be investigated further here. Later on the topic will be discussed for a restricted class of AGs.

## 2.4 DETERMINATE AND UNAMBIGUOUS ATTRIBUTE GRAMMARS

Let us for a moment reconsider the definition of  $\underline{L}(G, \mathcal{D})$  (definition 11). A string  $w$  is in  $\underline{L}(G, \mathcal{D})$  if there exists an element  $v$  in  $\text{meaning}(w)$  and none of the components of  $v$  are equal to  $\perp$  and  $\_$ . But  $\text{meaning}(w)$  may contain several elements with that property. A natural requirement to an AG will be that the meaning of a string in the language is uniquely determined. This leads to the definition of two subclasses of AGs, the determinate and the unambiguous AGs.

In the following we will only consider the translational approach to language definition. Similar results can be obtained for the traditional approach.

Let us first introduce and discuss the determinate AGs.

### DEFINITION 12:

An AG  $G$  over a semantic domain  $\mathcal{D} = (\underline{D}, \underline{E})$  is determinate if for any string  $w \in \underline{L}(G, \mathcal{D})$  and any  $v_1, v_2 \in \text{meaning}(w)$  we have  $v_1 = v_2$ .

An AG  $G$  over a semantic domain  $\mathcal{D}$  is e-determinate if for any string  $w \in \underline{L}_e(G, \mathcal{D})$  and any  $v_1, v_2 \in \text{meaning}_e(w)$  we have  $v_1 = v_2$ .

///

Corresponding to these properties of AGs we define two classes of AGs: D-AG and ED-AG:

### DEFINITION 13:

D-AG =  $\{G \mid G \text{ is a determinate AG}\}$

ED-AG =  $\{G \mid G \text{ is an e-determinate AG}\}$

///

### THEOREM 3:

ED-AG  $\subseteq$  D-AG and ED-AG  $\neq$  D-AG

PROOF: Let  $G \in$  ED-AG and assume that  $G$  is not in D-AG. Then there is a string  $w \in \underline{L}(G, \mathcal{D})$  with two elements  $v_1, v_2 \in \text{meaning}(w)$  and such that  $v_1 \neq v_2$ . But  $\text{meaning}(w) \subseteq \text{meaning}_e(w)$  and  $G$  will not be in ED-AG.

If  $A_1, A_2, A_3,$  and  $A_4$  are different then the AG in example 3 is in D-AG but not in ED-AG:  $A_3 A(\text{true})$  is not in  $\underline{L}(G, \mathcal{D})$  and we do not care about its meaning; but  $A_3 A(\text{true})$  is in  $\underline{L}_e(G, \mathcal{D})$  and now we are interested in its



meaning.

///

THEOREM 4:

It is undecidable whether an AG  $G$  is in D-AG (or ED-AG).

PROOF: Consider a CFG  $G_u = (V_n, V_t, R_u, Z)$ . Construct an AG  $G$  with the underlying grammar  $G_u$  such that each symbol has a single synthesized attribute. The lattice for the attribute of the symbol  $X$  is the set  $DOM(X)$  extended to a flat lattice. The tree concatenation operation is extended to the complete lattices by letting the operation be strict in its parameters. If in  $R_u$   $p: X ::= X_1 X_2 \dots X_n$  then we have in  $G$

$$p': \langle X \uparrow [t_1 \ t_2 \ \dots \ t_n] \rangle ::= \langle X_1 \uparrow t_1 \rangle \langle X_2 \uparrow t_2 \rangle \dots \langle X_n \uparrow t_n \rangle$$

If  $X \in V_t$  then the external semantic rules defining the attribute of  $X$  gives it the value  $X$ . Then  $G$  is in D-AG if and only if  $G_u$  is an unambiguous CFG. Since it is undecidable whether a CFG is unambiguous ([AhU72]) it will also be undecidable whether an AG is in D-AG. In a similar way we have that it is undecidable whether an AG is in ED-AG.

///

One may be interested in AGs with the property that for any string  $w$  in the language  $\text{meaning}(w)$  will only contain one single element with components different from  $\perp$  and  $\top$ . This leads to the introduction of the second subclass of AGs, the unambiguous AGs.

DEFINITION 14:

An AG  $G$  over a semantic domain  $\mathcal{D} = (\underline{D}, \underline{F})$  is unambiguous if for any string  $w \in \underline{L}(G, \mathcal{D})$  and any  $v_1, v_2 \in \text{meaning}(w)$  where in both  $v_1$  and  $v_2$  all components are different from  $\perp$  and  $\top$  we have  $v_1 = v_2$ .

An AG  $G$  over a semantic domain  $\mathcal{D} = (\underline{D}, \underline{F})$  is e-unambiguous if for any string  $w \in \underline{L}\text{-e}(G, \mathcal{D})$  and any  $v_1, v_2 \in \text{meaning}\text{-e}(w)$  where in both  $v_1$  and  $v_2$  all components are different from  $\perp$  and  $\top$  we have  $v_1 = v_2$ .

///

DEFINITION 15:

U-AG = { $G$  |  $G$  is an unambiguous AG}

EU-AG = { $G$  |  $G$  is an e-unambiguous AG}

///

The following theorems are easy to show:

THEOREM 5:

$EU-AG \subseteq U-AG$  and  $EU-AG \neq U-AG$

///

THEOREM 6:

Any determinate AG is an unambiguous AG but not vice versa.

///

THEOREM 7:

It is undecidable whether an AG is in U-AG (or EU-AG).

///

## 2.5 THE FORMAL POWER OF ATTRIBUTE GRAMMARS

By associating attributes with the symbols of a CFG and semantic rules with the productions as defined in section 2.1 we get a very powerful tool. In fact it is possible to recognize any recursive enumerable set by an AG if one allow some simple operations in the semantic domains. We will now show how this can be done by simulating a Turing Machine (for a definition see e.g. [AHU74]). A similar construction is given by Watt in [Wat74].

Let  $M = (Q, T, I, \delta, \emptyset, q_0, q_f)$  be a deterministic Turing Machine. We have

$Q$ : is a set of states

$T$ : is a set of tape symbols

$I$ : is a set of input symbols,  $I \subseteq T$

$\delta$ : is the transition function  $\delta: Q \times T \rightarrow Q \times T \times \{L, R, S\}$

$\emptyset$ : is the blank symbol,  $\emptyset \in T - I$

$q_0$ : is the initial state

$q_f$ : is the final state

We define an AG  $G = (V, B, R, Z)$  over a semantic domain  $\mathcal{D} = (\underline{D}, \underline{E})$ :

$\underline{D}$ : contains flat lattices constructed by extensions of the sets  $Q$ ,  $T$ ,  $T^*$ , and  $T \times Q$ .

$\underline{E}$ : contains the functions:

append:  $T^* \times T^* \rightarrow T^*$

but-first, but-last:  $T^* \rightarrow T^*$

$first, last: T^* \rightarrow T$   
 $cond: (T \times Q) \times (T \times Q) \rightarrow T \times Q$   
 $condz: (T \times Q) \rightarrow Q$   
 $cond-first, cond-last: (T \times Q) \times (T \times Q) \times T^* \times Q \rightarrow T^*$

The function 'append' concatenates two strings. Only if both the parameters are  $\perp$  the result is  $\perp$ .

The functions 'but-first' resp. 'but-last' remove the first resp. the last element from the string. If the string is the empty string then the result will be  $\perp$ .

The functions 'first' and 'last' give the first resp. the last element of a string. If the string is the empty string then the result will be  $\perp$ .

The function 'cond' takes as parameters two pairs. If the two first of these pairs are equal the result will be the third pair, otherwise it is  $\perp$ .

The function 'condz' takes a pair  $(A, q)$  as parameter and gives the value  $qf$  as result if  $q = qf$ , otherwise the result is  $\perp$ .

The functions give the value  $\perp$  as result if at least one of their parameters have the value  $\perp$ .

cond-first is defined by

$$cond-first(th1, th2, tr, q) = cond(th1, th2, (first(th), q))$$

cond-last is defined by

$$cond-last(th1, th2, tl, q) = cond(th1, th2, (last(tl), q))$$

The components of the AG are defined as:

$$V: V_n = \{X, Y, Z\}, V_t = I$$

$$S(X) = \{t\}, t \text{ is associated with } T^*$$

$$I(X) = \emptyset$$

$$S(Y) = \{tl, th, tr\}, tl \text{ and } tr \text{ are associated with } T^*$$

$$th \text{ is associated with } T \times Q$$

$$I(Y) = \emptyset$$

$$S(Z) = \{q\}, q \text{ is associated with } Q$$

$$I(Z) = \emptyset$$

B:  $tl$ : associated with  $T^*$ , will denote the string to the left of the head.

$th$ : associated with  $T \times Q$ , will denote the symbol under the head and

the state of the Turing Machine  
 $tr$ : associated with  $T^*$ , will denote the string to the right of the head.

R: For each symbol  $A \in I$  we have:

$\langle X \uparrow \text{append}(tr, A) \rangle ::= \langle X \uparrow tr \rangle A$

$\langle X \uparrow \lambda \rangle ::= \lambda$

A lot of chain productions  $Y ::= Y$  is used in order to simulate the Turing Machine:

$\langle Y \uparrow \lambda \uparrow (\text{first}(tr), q_0) \uparrow tr \rangle ::= \langle X \uparrow tr \rangle$

If  $\delta(q, B) = (q', B', S)$  then

$\langle Y \uparrow tl \uparrow \text{cond}(th, (B, q), (B', q')) \uparrow tr \rangle ::= \langle Y \uparrow tl \uparrow th \uparrow tr \rangle$

If  $\delta(q, B) = (q', B', L)$  then

$\langle Y \uparrow \text{but-last}(tl) \uparrow \text{cond-last}(th, (B, q), tl, q') \uparrow \text{append}(B', tr) \rangle ::= \langle Y \uparrow tl \uparrow th \uparrow tr \rangle$

If  $\delta(q, B) = (q', B', R)$  then

$\langle Y \uparrow \text{append}(tl, B') \uparrow \text{cond-first}(th, (B, q), tr, q) \uparrow \text{but-first}(tr) \rangle ::= \langle Y \uparrow tl \uparrow th \uparrow tr \rangle$

$\langle Z \uparrow \text{condz}(th) \rangle ::= \langle Y \uparrow tl \uparrow th \uparrow tr \rangle$

If  $\underline{L}(M)$  is the language defined by the Turing Machine then  $\underline{L}(M) = \underline{L}(G, \mathbb{D})$ . And we have the result:

THEOREM 8:

Any recursively enumerable set can be defined by an AG.

///

Inspection of the construction above shows that the AG is unambiguous, that is we have

THEOREM 9:

Any recursively enumerable set can be defined by an unambiguous AG.

///

An immediately consequence of this is

THEOREM 10:

The general membership problem is not solvable for neither arbitrary

AGs nor unambiguous AGs.

///

If we consider the determinate AGs we have these results:

THEOREM 11:

Any recursive set can be defined by the determinate AGs.

PROOF: We construct an AG  $G = (V, B, R, Z)$  over a semantic domain  $\mathcal{D} = (\underline{D}, \underline{F})$ :  $\underline{D}$  consists of the two sets  $Vt^*$  and  $\{\text{accept}\}$  extended to flat lattices.  $\underline{F}$  consists of two functions, 'append' (almost equal to the function with the same name mentioned earlier) and  $f: Vt^* \rightarrow \{\text{accept}\}$ , a recursive function.  $f$  is extended to operate on the complete lattices. The function  $f'$  is defined by  $f'(w) = f(w)$  if  $f$  is defined on  $w$ ,  $f'(!) = !$ ,  $f'(? ) = ?$  and  $f'(w) = !$  otherwise.

The underlying grammar of  $G$  is  $G_u = (\{X, Z\}, Vt, R_u, Z)$  where  $R_u$  contains the rules:

$Z ::= X$

$X ::= X A$  for every  $A \in Vt$

$X ::= A$  for every  $A \in Vt$

We have  $S(X) = \{b\}$ ,  $I(X) = \emptyset$ ,  $S(Z) = \{c\}$ .  $b$  is associated with the flat lattice constructed from  $Vt^*$  and  $c$  with that from  $\{\text{accept}\}$ . For every  $A \in Vt$  we have  $A(A) = \emptyset$ . In  $R$  we have the productions

$\langle Z \uparrow f(a) \rangle ::= \langle X \uparrow a \rangle$

$\langle X \uparrow \text{append}(a, A) \rangle ::= \langle X \uparrow a \rangle A$  for every  $A \in Vt$

$\langle X \uparrow A \rangle ::= A$  for every  $A \in Vt$

///

THEOREM 12:

The general membership problem is solvable for the determinate AGs.

PROOF: Let  $G$  be a determinate AG over a semantic domain  $\mathcal{D} = (\underline{D}, \underline{F})$ . In order to determine whether a string  $w$  is in  $\underline{L}(G, \mathcal{D})$  we construct a derivation tree for  $w$ , determine an initial semantic tree and the minimal evaluated semantic tree (it is possible according to the discussion at the end of section 2.3). Inspection of the evaluated semantic tree shows whether  $w$  is in the language or not.

///

### 3. WELL-DEFINED ATTRIBUTE GRAMMARS

---

In this chapter (and in the rest of this thesis) I put restrictions on the semantic domains of an AG. It is required that the operations are strict (in a slightly different sense than usually):

Let  $\mathcal{D} = (\underline{D}, \underline{F})$  be a semantic domain. Then every function  $f: D_1 \times D_2 \times \dots \times D_m \rightarrow D_0$  in  $\underline{F}$  must satisfy:

- $f(v_1, v_2, \dots, v_m) = \perp$  if and only if  $v_j = \perp$  for some  $j$
- $f(v_1, v_2, \dots, v_m) = \perp$  if  $v_j \neq \perp$  for all  $j$ , and if  $v_j = \perp$  for some  $j$ .

When using semantic domains with these properties it becomes convenient to represent the dependencies between the attributes as graphs. In section 3.1 I introduce so-called dependency graphs for both the productions and the symbols of the AG. These graphs are of great importance for this and the remaining chapters.

This leads to the introduction of a subclass of AGs, the well-defined AGs, in section 3.2. An AG in this class has the property that in any obtainable derivation tree for the underlying grammar it is possible to evaluate all the attributes to values different from  $\perp$ . It turns out that the class of well-defined AGs and the classes D-AG and U-AG from the previous chapter are incomparable.

In section 3.3 I introduce a subclass of the well-defined AGs and show how any well-defined AG can be transformed into an AG in this class. This class and transformation are mainly introduced in order to simplify some of the proofs in the following sections. And a slightly modification of the transformation shows that any AG can be transformed into a well-defined AG over the same semantic domain and defining the same translation.

In the translational approach there may be some attributes associated with a derivation tree whose values do not influence the meaning of the string. In section 3.4 I give transformations that remove these attributes. There are given two constructions, the first one removes all the attributes that never will influence the meaning. The second construction transforms the AG into an equivalent AG where every attribute of every symbol influen-

ces the meaning if the symbol occurs in the derivation tree.

### 3.1 DEPENDENCY GRAPHS

When using semantic domains with properties as described above it is possible statically to discuss whether one attribute depends on another. An important concept when doing so is the dependency graphs. A dependency graph is a directed graph where the nodes represent attributes and an arc from a node representing an attribute  $b$  to a node representing another attribute  $c$  means that the value of  $c$  may depend on the value of  $b$ . Often we will refer to a node in a dependency graph as an attribute.

We define dependency graphs for both symbols and productions in an AG. A dependency graph for a symbol is any graph with a node for each of the attributes of the symbol. A dependency graph for a production is any graph with a node for each of the attributes of each of the symbols occurring in the production.

The fundamental of the dependency graphs is a dependency graph  $D(p)$  for each of the productions  $p$  in the AG.

#### DEFINITION 1:

Let  $p: X ::= X_1 X_2 \dots X_n$  be a production in an AG  $G$ . The dependency graph  $D(p)$  for  $p$  is the graph determined by that there is an arc from  $b$  to  $c$  if and only if the attribute variable for  $b$  in  $p$  occurs in the semantic rule for  $c$  in  $p$ .

///

We can note that any arc in  $D(p)$  will begin at a defining attribute and end at an applied.

#### EXAMPLE 1:

Consider an AG  $G$  over a semantic domain  $\mathcal{D}$  which contains the identity function.  $G$  contains the productions:

$p_1: \langle Z \uparrow b \rangle ::= B \langle X \downarrow a \downarrow 0 \uparrow b \uparrow a \rangle$

$p_2: \langle Z \uparrow b \rangle ::= C \langle Y \downarrow b \uparrow b \rangle$

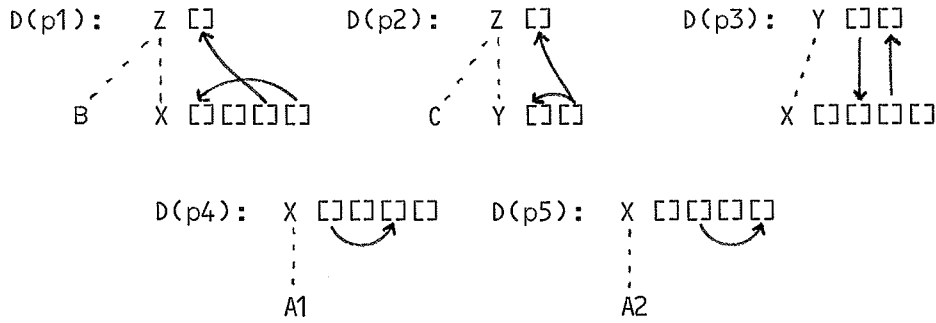
$p_3: \langle Y \downarrow a \uparrow b \rangle ::= \langle X \downarrow 1 \downarrow a \uparrow b \uparrow c \rangle$

$p_4: \langle X \downarrow a \downarrow b \uparrow a \uparrow 1 \rangle ::= A_1$

$p_5: \langle X \downarrow a \downarrow b \uparrow 0 \uparrow b \rangle ::= A_2$

Then we have (the ordering of the nodes is equal to that of the at-

tributes):



///

The dependency graphs  $D(p)$  can be put together in a way determined by a derivation tree and thereby define a dependency network for the tree telling how the attributes associated with the tree depend on each other.

DEFINITION 2:

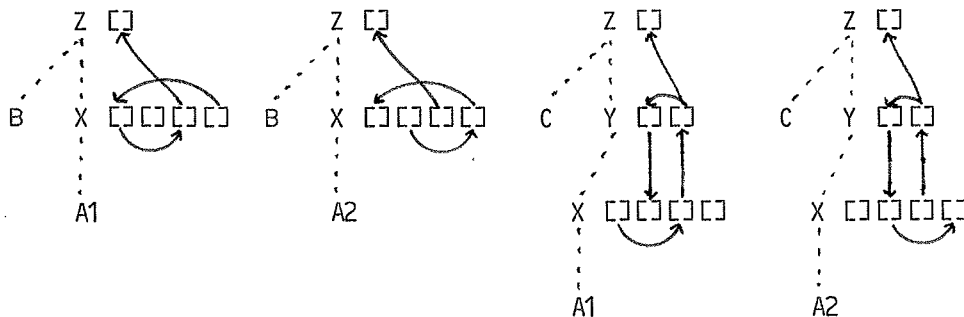
Let  $G$  be an AG and let  $t \in \text{DOM}(X)$ . The dependency network  $D(t)$  for  $t$  is:

- if  $t = X, X \in V_t$  then  $D(t)$  is the symbol dependency graph for  $X$  with no arcs
- if  $t = X[t_1 t_2 \dots t_n]$  and the production applied at  $X$  is  $p: X ::= X_1 X_2 \dots X_n$  then
 
$$D(t) = D(p) \cup D(t_1) \cup D(t_2) \dots \cup D(t_n)$$
- if  $t = X[\lambda]$  then  $D(t) = D(p)$  where  $p: X ::= \lambda$ .

///

EXAMPLE 2:

Consider the AG from example 1. We may have the following dependency networks:



///

The dependency network  $D(t)$  for  $t \in \text{DOM}(X)$  shows among other things how the attributes of  $X$  depend on each other in  $t$ . It is possible to con-



struct a set of dependency graphs for X showing all possible dependencies between the attributes of X that can occur in trees in  $DOM(X)$ .

DEFINITION 3:

Let G be an AG. For each symbol X in V we define a set of dependency graphs  $SYM(X)$  for X:

- if  $X \in V_t$  then  $SYM(X)$  consists of a single graph, the graph with no arcs
- if  $X \in V_n$  and  $p: X ::= X_1 X_2 \dots X_n$  ( $n > 0$ ) is a production in R then there will be a graph Q in  $SYM(X)$  for each choice of graphs  $Q_j \in SYM(X_j)$  for  $1 \leq j \leq n$ . The graph Q has an arc from b to c if and only if there is a path from b to c in the graph

$$D(p) \equiv Q_1 Q_2 \dots Q_n$$

- if  $X \in V_n$  and  $p: X ::= \lambda$  is a production in R then the graph  $Q = D(p)$  will be in  $SYM(X)$ .

///

EXAMPLE 3:

For the AG in example 1 we have:

$SYM(Z)$ :  $Q_1$ :  $\square$

$SYM(X)$ :  $Q_2$ :  $\square \xrightarrow{\quad} \square \square \square \square$        $Q_3$ :  $\square \square \xrightarrow{\quad} \square \square$

$SYM(Y)$ :  $Q_4$ :  $\square \square$

///

THEOREM 1:

Let  $t \in DOM(X)$  and define  $sym(t)$ , the symbol dependency graph for X corresponding to t by:

$$b \rightarrow c \text{ in } sym(t) \iff b \rightarrow^+ c \text{ in } D(t)$$

Then  $sym(t) \in SYM(X)$ .

If  $Q \in SYM(X)$  then there is a tree  $t \in DOM(X)$  such that  $sym(t) = Q$ .

PROOF: The theorem follows from the proof for the circularity test in [Knu71].

///

The dependency graphs and networks presented in these three definitions are originally introduced by Knuth ([Knu68], [Knu71]).

We can note that in a graph Q in  $SYM(X)$  any arc  $b \rightarrow c$  will begin at an inherited attribute for X (b) and end at a synthesized attribute (c). Thus

graphs in  $SYM(X)$  will only tell how the synthesized attributes of a symbol may depend on the inherited attributes. I will now define dependency graphs for the symbols that can tell how the inherited attributes of a symbol depends on the synthesized in a derivation tree. These graphs are called context dependency graphs.

DEFINITION 4:

Let  $G$  be an AG. For each symbol  $X$  in  $V$  we define the set of context dependency graphs  $CON(X)$  for  $X$ :

- if  $X = Z$  then  $CON(Z)$  consists of a single graph, the graph with no arcs
- if  $p: X ::= X_1 X_2 \dots X_n$  is a production in  $R$  then there will be a graph in  $CON(X_k)$  for each choice of graphs  $Q \in CON(X)$  and  $Q_j' \in SYM(X_j)$ ,  $j \neq k$ . Let  $Q_k'$  be the graph with no arcs. Then the graph  $Q_k$  has an arc from  $b$  to  $c$  if there is a path from  $b$  to  $c$  in the graph  $Q \cup Q_1' \cup Q_2' \dots Q_n'$

///

EXAMPLE 4:

For the AG in example 1 we have:

$CON(Z): \quad Q_1': \square$

$CON(X): \quad Q_2': \begin{array}{c} \curvearrowright \\ \square \square \square \square \end{array} \quad Q_3': \begin{array}{c} \curvearrowleft \\ \square \square \square \square \end{array}$

$CON(Y): \quad Q_4': \square \square$

///

We can note that any arc  $b \rightarrow c$  in a graph  $Q \in CON(X)$  starts at a synthesized attribute ( $b$ ) and ends at an inherited ( $c$ ).

Corresponding to theorem 1 we have

THEOREM 2:

Let  $t \in DOM(Z)$  and consider a subtree  $t' \in DOM(X)$  of  $t$ . We define the context dependency graph  $con(t')$  for  $X$  corresponding to  $t'$  by:

$$b \rightarrow c \text{ in } con(t') \iff b \rightarrow c \text{ in } D(t), b \in S(X), \text{ and } c \in I(X)$$

Then  $con(t') \in CON(X)$ .

If  $Q \in CON(X)$  then there is a tree  $t \in DOM(Z)$  with a subtree  $t' \in DOM(X)$  such that  $con(t') = Q$ .

PROOF: One may use induction in the length of a path from the root of  $t$  to the node  $X$ . We omit the proof.

///

### 3.2 THE WELL-DEFINEDNESS PROPERTY

We will now follow Knuth in defining a class of AGs called the well-defined (or non-circular) AGs ([Knu68]). An AG in this class is characterized by that for every tree  $t$  in  $\text{DOM}(X)$ ,  $X \in V$ , there are no cycles in  $D(t)$ .

DEFINITION 5:

An AG is called well-defined if there are no cycles in any of the dependency networks  $D(t)$  for  $t \in \text{DOM}(Z)$ .

$W\text{-AG} = \{G \mid G \text{ is well-defined}\}$

///

Well-definedness is a static property. Knuth has shown [Knu71]:

THEOREM 3:

An AG is well-defined if and only if for any production  $p: X ::= X_1 X_2 \dots X_n$  there are no cycles in the graph

$D(p) \cup \{Q_1, Q_2, \dots, Q_n\}$

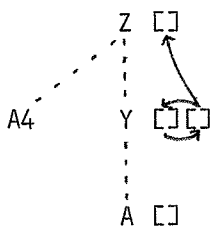
for any choice of graphs  $Q_j \in \text{SYM}(X_j)$ .

///

EXAMPLE 5:

The AG of example 1 is well-defined (see example 2).

The AG in example 5 of chapter 2 is not well-defined, we have a cycle in the dependency network for e.g. the derivation tree  $Z[A_4 Y[A]]$  for the string  $A_4 A(\text{true})$ :



///

Jazayeri, Ogden, and Rounds have shown that the test for well-definedness presented above is of exponential complexity, and furthermore they

have shown that any test for well-definedness will be exponential ([JOR75]).

Let us return to one of the topics mentioned in section 2.3. How can we find an algorithm determining the meaning of a string, i.e. the least fixpoint of the function  $F$  in definition 8 in chapter 2. Consider a well-defined AG. Since no attributes in any derivation tree depend circularly on each other we can determine an ordering of the attributes associated with the tree. The ordering can be chosen such that if an attribute  $b$  precedes an attribute  $c$  in the ordering then the value of  $b$  does not depend on the value of  $c$ . The attributes can then be evaluated in this order. A compiler writing system DELTA is based upon this idea ([Lor77]).

It is also possible to show that never mind which order of the attributes one chooses then the resulting evaluated semantic tree will be the same, the least fixpoint of the  $F$ -function. Furthermore this will be the only fixpoint of the  $F$ -function and thus the 'evaluated semantic tree' and the 'least fixpoint' approaches to language definition in chapter 2 will coincide.

Mayoh expresses the same results in a slightly modified form ([May78]). He reformulates AGs in mathematical semantics. A solution to his mathematical semantic equations specifies an evaluated semantic tree. Mayoh shows among other things that if the AG is well-defined then one needs not use the fixpoint operator when solving the semantic equations. Thus it is straight forward to find a solution as well as it is straight forward to find an order in which the attributes associated with a derivation tree can be evaluated.

In chapter 4 and 5 I will give specific algorithms that may be used to construct evaluated semantic trees for well-defined AGs.

The discussion above shows that if a string  $w$  is in e.g. the language  $\underline{L}(G, \emptyset)$  then it is possible to give values to the attributes in a derivation tree for  $w$  such that all these values are different from  $\underline{!}$  and  $\underline{?}$ . And therefore  $w$  will also be in  $\underline{L}'(G, \emptyset)$ . The two approaches traditional and translational will give the same language but of course not the same translations.

Thus the conclusion is that the four language definitions in chapter 2

coincide for the well-defined AGs.

We will assume that the external semantic rules uniquely define an initial semantic tree  $t'$  for a string  $w$  with a derivation tree  $t$ . The meaning of  $w$  determined from  $t'$  will be denoted by  $\text{meaning}(w, t)$  in the translational approach and  $\text{meaning}'(w, t)$  in the traditional.

We will now compare the class W-AG with the classes D-AG and U-AG from the previous chapter. Intuitively the classes will be incomparable since in the classes D-AG and U-AG we care about the value of the meaning of a string and not whether there are cycles in a network. On the other hand in the class W-AG we care about the cycles but not the values of the meaning. We have the result:

THEOREM 4:

The classes D-AG and W-AG are incomparable.

The classes U-AG and W-AG are incomparable.

PROOF: Let  $A_1 = A_2 = A$  in the AG of example 1. Then the AG is neither in U-AG nor in D-AG since  $\text{meaning}(B A) = \{0, 1\}$  and  $\text{meaning}(C A) = \{0, 1\}$ . But the AG is in W-AG (according to example 5).

The AG in example 3 of chapter 2 is in D-AG (and thereby in U-AG). But it is not in W-AG (according to example 5).

///

### 3.3 TRANSFORMATIONS ON AGs

In this section I define a subclass of AGs called the partly uniform AGs. A partly uniform AG has the property that the dependencies between the attributes of a symbol are independent of what the symbol may derive but may depend on the context of the symbol in a derivation tree.

DEFINITION 6:

An AG  $G = (V, B, R, Z)$  is partly uniform if it is well-defined and if for every  $X \in V$ ,  $\text{SYM}(X)$  consists of a single graph.

///

We have the result:

THEOREM 5:

Any AG  $G = (V, B, R, Z)$  over a semantic domain  $\mathcal{D}$  can be transformed into a partly uniform AG  $G' = (V', B', R', Z')$  over the same semantic domain  $\mathcal{D}$  and such that

$$\underline{I}(G, \mathcal{D}) = \underline{I}(G', \mathcal{D})$$

CONSTRUCTION:  $G'$  is defined by

$V'$ :  $Vn' = \{(X, Q) \mid X \in Vn \text{ and } Q \in \text{SYM}(X)\}$  and  $Vt' = Vt$

For any  $(X, Q) \in Vn'$  let  $A(X, Q) = A(X)$  and for any  $X' \in Vt'$  let  $A(X') = A(X)$

$B'$ : is equal to  $B$

$R'$ : if  $p: X ::= X_1 X_2 \dots X_n$  is in  $R$  and  $Q_j \in \text{SYM}(X_j)$  for  $1 \leq j \leq n$  then

$$p': X' ::= X_1' X_2' \dots X_n'$$

is in  $R'$  where  $X_j' = (X_j, Q_j)$  if  $X_j \in Vn$  and  $X_j' = X_j$  if  $X_j \in Vt$  for  $1 \leq j \leq n$ .  $X' = (X, Q)$  where  $Q$  is the graph in  $\text{SYM}(X)$  derived from  $D(p) \text{EQ1}$   $Q_2 \dots Q_n$ . The semantic rules of  $p'$  are equal to those of  $p$ .

$Z'$ : is equal to  $(Z, Q)$  where  $Q$  is the only graph in  $\text{SYM}(Z)$ .

PROOF: In order to show the correctness of the theorem we consider two assertions:

i)  $\underline{I}(G, \mathcal{D}) = \underline{I}(G', \mathcal{D})$

ii)  $G'$  is a partly uniform AG

Proof of i): We define a homomorphism  $h'$  mapping any derivation tree of  $G$  into a derivation tree of  $G'$  ( $\text{sym}(t)$  is defined in theorem 1):

- if  $t = X$  ( $\in Vt$ ) then  $h'(t) = X$  ( $\in Vt'$ )

- if  $t = X[t_1 t_2 \dots t_n]$  and the production applied at  $X$  is

$p: X ::= X_1 X_2 \dots X_n$  ( $n \geq 0$ ) then

$$h'(t) = (X, \text{sym}(t))[h'(t_1) h'(t_2) \dots h'(t_n)]$$

Another homomorphism  $h$  is defined by:

- if  $t = X$  ( $\in Vt'$ ) then  $h(t) = X$  ( $\in Vt$ )

- if  $t = (X, Q)[t_1 t_2 \dots t_n]$  then  $h(t) = X[h(t_1) h(t_2) \dots h(t_n)]$

The existence of the two homomorphisms  $h$  and  $h'$  whose composition is the identity function ensures that  $\underline{I}(G, \mathcal{D}) = \underline{I}(G', \mathcal{D})$ .

Proof of ii): We will show that for any tree  $t \in \text{DOM}(X, Q)$  ( $(X, Q) \in Vn'$ ) we have  $\text{sym}(t) = Q$ . Theorem 1 then gives that  $\text{SYM}(X, Q) = \{Q\}$  and thereby that  $G'$  is partly uniform. We use induction in the height of  $t$ .

If the height is zero then obviously  $\text{sym}(t) = Q$  where  $Q$  is the graph with no arcs.

For the induction step let  $t = (X, Q)[t_1 t_2 \dots t_n]$  and let  $p: (X, Q) ::= X_1' X_2' \dots X_n'$  be the production applied at  $(X, Q)$ . The induction hypothesis gives that  $\text{sym}(t_j) = Q_j$  where  $X_j' = (X_j, Q_j)$  for  $1 \leq j \leq n$ . The graph  $\text{sym}(t)$  is derived from the graph  $D(t)$  or (equivalently)  $D(p) \mathbb{I} Q_1 Q_2 \dots Q_n \mathbb{I}$ . But so is the graph  $Q$  and we have  $\text{sym}(t) = Q$ .

///

We have formulated and proved the theorem in the translational style, but it can easily be modified to hold in the traditional approach.

#### EXAMPLE 6:

Let us apply the construction to the AG in example 1. Let  $Q_1, Q_2, Q_3,$  and  $Q_4$  be the graphs from example 3. The new AG then has the productions:

- $p1': \langle (Z, Q_1) \uparrow b \rangle ::= B \langle (X, Q_2) \downarrow a \downarrow 0 \uparrow b \uparrow a \rangle$
- $p1'': \langle (Z, Q_1) \uparrow b \rangle ::= B \langle (X, Q_3) \downarrow a \downarrow 0 \uparrow b \uparrow a \rangle$
- $p2': \langle (Z, Q_1) \uparrow b \rangle ::= C \langle (Y, Q_4) \downarrow b \uparrow b \rangle$
- $p3': \langle (Y, Q_4) \downarrow a \uparrow b \rangle ::= \langle (X, Q_2) \downarrow 1 \downarrow a \uparrow b \uparrow c \rangle$
- $p3'': \langle (Y, Q_4) \downarrow a \uparrow b \rangle ::= \langle (X, Q_3) \downarrow 1 \downarrow a \uparrow b \uparrow c \rangle$
- $p4': \langle (X, Q_2) \downarrow a \downarrow b \uparrow a \uparrow 1 \rangle ::= A1$
- $p5': \langle (X, Q_3) \downarrow a \downarrow b \uparrow 0 \uparrow b \rangle ::= A2$

///

#### THEOREM 6:

Let  $G = (V, B, R, Z)$  be an arbitrary AG over a semantic domain  $\mathcal{D}$ . Then there exists a well-defined AG  $G' = (V', B', R', Z')$  over the same semantic domain  $\mathcal{D}$  such that

$$\underline{L}'(G, \mathcal{D}) = \underline{L}'(G', \mathcal{D})$$

and

$$\underline{I}'(G, \mathcal{D}) = h(\underline{I}'(G', \mathcal{D}))$$

where  $h$  is a homomorphism.

CONSTRUCTION: First we construct a CFG  $G'' = (Vn'', Vt'', R'', Z'')$ :

$$Vn'': \{(X, Q) \mid X \in Vn, Q \in \text{SYM}(X)\}$$

$$Vt'': Vt$$

$R''$ : Let  $p: X ::= X_1 X_2 \dots X_n$  be a production in  $R$ . If  $Q \in \text{SYM}(X)$  could be derived from the graph

$$Q' = D(p) \mathbb{I} Q_1 \ Q_2 \ \dots \ Q_n \mathbb{I}$$

where  $Q_j \in \text{SYM}(X_j)$  for  $1 \leq j \leq n$  and if there are no cycles in  $Q'$  then there is a production

$$p': X' ::= X_1' \ X_2' \ \dots \ X_n'$$

in  $R''$  where  $X' = (X, Q)$  and for  $1 \leq j \leq n$   $X_j' = (X_j, Q_j)$  if  $X_j \in V_n$  and otherwise  $X_j' = X_j$ .

$Z'' = (Z, Q)$  where  $Q$  is the only graph in  $\text{SYM}(Z)$ .

Let  $G_{u'}$  be the CFG constructed by reducing  $G''$  (for a method see e.g. [AhU72]). The AG  $G'$  is constructed in a way very similarly to that in the construction for theorem 4, the only difference being that  $G_{u'}$  is the underlying grammar.

PROOF: The homomorphism  $h$  is as in the proof for theorem 4. The proof for that theorem can easily be extended to the traditional case and thereby we have that  $\underline{L}'(G', \mathcal{D}) \subseteq \underline{L}'(G, \mathcal{D})$  and  $h(\underline{I}'(G', \mathcal{D})) \subseteq \underline{I}'(G, \mathcal{D})$ .

Assume now that  $w \in \underline{L}'(G, \mathcal{D})$  but that  $w$  is not in  $\underline{L}'(G', \mathcal{D})$ . Let  $t$  be a derivation tree for  $w$  according to  $G_u$ . If  $D(t)$  does not contain a cycle then there will be productions in  $G'$  such that  $w \in \underline{L}'(G', \mathcal{D})$ . If  $D(t)$  contains a cycle then  $w$  is not in  $\underline{L}'(G, \mathcal{D})$  and  $\underline{L}'(G', \mathcal{D})$  either. It is easy to see that  $h(\underline{I}'(G', \mathcal{D})) = \underline{I}'(G, \mathcal{D})$ .

///

EXAMPLE 7:

Consider the AG in example 2 of chapter 2. We have

$$\text{SYM}(Z): \quad Q_1: \quad \square$$

$$\text{SYM}(X): \quad Q_2: \quad \begin{array}{c} \square \square \square \\ \curvearrowright \end{array}$$

$$\text{SYM}(Y): \quad Q_3: \quad \square \square$$

The productions  $p_2$ ,  $p_4$ , and  $p_6$  give rise to circular dependency graphs (i.e.  $D(p) \mathbb{I} Q_1 \ Q_2 \ \dots \ Q_n \mathbb{I}$  graphs) and the CFG  $G''$  of the construction will contain the following productions:

$$p_1': (Z, Q_1) ::= A_1 \ A$$

$$p_3': (X, Q_2) ::= A_2$$

$$p_5': (Y, Q_3) ::= A$$

Thus the resulting AG  $G'$  has the single production:



$p1': \langle (Z, Q1) \uparrow 1 \rangle ::= A1 \langle A \uparrow a \rangle$

///

Theorem 6 shows that in the traditional approach it is not a restriction only to consider well-defined AGs because any language and any translation that can be specified by an AG can be specified by a well-defined AG over the same semantic domain.

Obviously any translation that can be specified by an AG in the traditional approach can be specified by an AG in the translational approach (by an appropriate choice of the semantic domain). The reason why theorem 6 does not hold in the translational approach is intuitively that there may be attributes in a semantic tree whose values do not influence the meaning.

### 3.4 USELESS ATTRIBUTES

When using the translational approach there may be attributes in the semantic tree whose values never will influence the meaning of a string.

#### DEFINITION 7:

Let  $G$  be an AG. An attribute  $b$  of a symbol  $X$  is called useless if there does not exist a derivation tree  $t \in \text{DOM}(Z)$  such that  $b \rightarrow^* c$  in  $D(t)$  for some  $c \in S(Z)$ .

///

I now give an algorithm which detect the useless attributes for each symbol in the AG. For each symbol  $X$  and each SYM-graph  $Q$  and CON-graph  $Q'$  for  $X$  we determine a set  $N_i(X, Q, Q')$  of attributes which not are useless.

#### ALGORITHM 1:

Input: an AG  $G = (V, B, R, Z)$

Output: A set  $U(X)$  of useless attributes for each  $X \in V$

Method:

1.  $NO(Z, Q, Q') = S(Z)$  for  $Q \in \text{SYM}(Z)$  and  $Q' \in \text{CON}(Z)$

$NO(X, Q, Q') = \emptyset$  otherwise

Let  $i = 0$

2. Let  $N_{(i+1)}(X, Q, Q') = N_i(X, Q, Q')$  for all  $X, Q \in \text{SYM}(X)$  and  $Q' \in \text{CON}(X)$ .

FOR each production  $p: X ::= X_1 X_2 \dots X_n$  and FOR each symbol  $X_k$  on

the right hand side of p D0

For each choice of graphs  $Q^1 \in \text{CON}(X)$ ,  $Q \in \text{SYM}(X)$ ,  $Q_j \in \text{SYM}(X_j)$  for  $1 \leq j \leq n$ , and  $Q_k^1 \in \text{CON}(X_k)$  such that:

i)  $Q$  is derived from the graph

$$D(p) \mathbb{E} Q_1 Q_2 \dots Q_n \mathbb{I}$$

ii)  $Q_k^1$  is derived from the graph

$$Q^1 \mathbb{E} D(p) \mathbb{E} Q_1'' Q_2'' \dots Q_n'' \mathbb{I}$$

where  $Q_j'' = Q_j$  for  $j \neq k$  and  $Q_k''$  is the graph with no arcs

Let  $Q'' = D(p) \mathbb{E} Q_1 Q_2 \dots Q_n \mathbb{I}$  and

$$N^{(i+1)}(X_k, Q_k, Q_k^1) = \{c \in A(X_k) \mid \text{there is a } b \in N_i(X, Q, Q^1) \text{ such that } c \rightarrow^* b \text{ in } Q''\}$$

$$\cup N^{(i)}(X_k, Q_k, Q_k^1)$$

3. If there exist an  $X \in V$ ,  $Q \in \text{SYM}(X)$ , and  $Q^1 \in \text{CON}(X)$  such that  $N^{(i+1)}(X, Q, Q^1) \neq N_i(X, Q, Q^1)$  then let  $i := i + 1$  and go to step 2.

4. Let  $U(X) = A(X) - \{a \mid a \in N_i(X, Q, Q^1) \text{ for some } Q \in \text{SYM}(X) \text{ and some } Q^1 \in \text{CON}(X)\}$

and stop.

///

Clearly the algorithm will stop since we have a fixed number of attributes in each of the sets  $A(X)$  and a fixed number of graphs in each of the sets  $\text{SYM}(X)$  and  $\text{CON}(X)$ .

EXAMPLE 8:

Consider the AG from example 1. Let

$$A(Z) = \{a\}$$

$$A(X) = \{a, b, c, d\}$$

$$A(Y) = \{a, b\}$$

Then we have these non-empty sets from the algorithm:

$$N_0(Z, Q_1, Q_1^1) = \{a\}$$

$$N_2(Z, Q_1, Q_1^1) = \{a\}$$

$$N_1(Z, Q_1, Q_1^1) = \{a\}$$

$$N_2(X, Q_2, Q_2^1) = \{a, c\}$$

$$N_1(X, Q_2, Q_3^1) = \{a, c, d\}$$

$$N_2(X, Q_2, Q_3^1) = \{a, c, d\}$$

$$N_1(X, Q_3, Q_3^1) = \{c\}$$

$$N_2(X, Q_3, Q_2^1) = \{c\}$$

$$N_2(X, Q_3, Q_3^1) = \{c\}$$

$$N1(Y, Q4, Q4') = \{b\}$$

$$N2(Y, Q4, Q4') = \{b\}$$

The  $N3$ -sets are equal to the  $N2$ -sets.

That is we have  $U(Z) = \emptyset$ ,  $U(X) = \{b\}$ , and  $U(Y) = \{a\}$ .

///

LEMMA 1:

An attribute  $b$  is in  $U(X)$  if and only if it is useless.

PROOF: Assume that  $b \in U(X_k)$  but that  $b$  is not useless, that is there exists a derivation tree  $t \in \text{DOM}(Z)$  and an attribute  $c \in S(Z)$  such that  $b \rightarrow^* c$  is in  $D(t)$ .

Let  $p: X ::= X_1 X_2 \dots X_n$  be the production introducing  $X_k$  in  $t$ . Determine the graphs  $Q \in \text{SYM}(X)$ ,  $Q_j \in \text{SYM}(X_j)$ ,  $Q' \in \text{CON}(X)$  and  $Q_j' \in \text{CON}(X_j)$  for  $1 \leq j \leq n$  from  $t$ . Let

$$Q'' = D(p) \{Q_1 Q_2 \dots Q_n\}$$

Let  $b = b_0 \rightarrow b_1 \rightarrow \dots \rightarrow b_m = c$  be the path in  $D(t)$ . One of these attributes  $b_r$  is in  $A(X)$ .  $b_r$  can be chosen such that  $b_0 \rightarrow^* b_r$  is a path in  $Q''$  (follows from theorem 1).  $r$  will be the minimal value  $h$  such that  $b_h$  is in  $A(X)$ . We have  $b_r \in U(X)$  since the opposite implies that  $b$  is not in  $U(X)$ . On the other hand  $b_r$  is not useless ( $b_r \rightarrow^* c$  in  $D(t)$ ) and induction in the length of the path gives that  $c \in U(Z)$ , a contradiction. We have thereby shown that if  $b$  is in  $U(X)$  then  $b$  is useless.

In order to show the rest of the theorem we use induction in ' $i$ '. The induction hypothesis will be:

If  $b \in N(i+1)(X_k, Q_k, Q_k')$  then there will be a tree  $t \in \text{DOM}(Z)$  with a subtree  $t'$  with root  $X_k$  and such that

- 1)  $\text{con}(t') = Q_k'$
- 2)  $\text{sym}(t') = Q_k$
- 3) there is a path  $b \rightarrow^* c$  in  $D(t)$  where  $c \in S(Z)$ .

$i=0$ : Let  $b \in N1(X_k, Q_k, Q_k')$ . Then  $X_k$  is a symbol on the right hand side of a production  $p: Z ::= X_1 X_2 \dots X_n$ , and there are graphs  $Q' \in \text{CON}(Z)$ ,  $Q \in \text{SYM}(X)$  and  $Q_j \in \text{SYM}(X_j)$  for  $1 \leq j \leq n$  such that i) and ii) are satisfied.

Choose trees  $t_j \in \text{DOM}(X_j)$  such that  $\text{sym}(t_j) = Q_j$  for  $1 \leq j \leq n$  (it is possible according to theorem 1) and let  $t = Z[t_1 t_2 \dots t_n]$ . The tree  $t$  satisfies the requirements 1) - 3) above:

- 1)  $\text{con}(t_k) = Q_k'$  follows from condition ii)

- 2)  $\text{sym}(tk) = Q_k$  follows from the construction of  $t$
- 3) as  $b \in M(X_k, Q_k, Q_k')$  there will be an attribute  $c \in NO(Z, Q, Q')$  ( $\in S(Z)$ ) such that  $b \rightarrow^* c$  is in  $D(p) \mathbb{I} Q_1 Q_2 \dots Q_n \mathbb{I}$  and thereby in  $D(t)$ .

The induction step: Let  $b \in N^{(i+1)}(X_k, Q_k, Q_k')$  and assume that  $b$  is not in  $N^i(X_k, Q_k, Q_k')$ . Then there is a production  $p: X ::= X_1 X_2 \dots X_n$  such that for some  $d \in N^i(X, Q, Q')$   $b \rightarrow^* d$  is in  $D(p) \mathbb{I} Q_1 Q_2 \dots Q_n \mathbb{I}$  where  $Q_j \in \text{SYM}(X_j)$  for  $1 \leq j \leq n$ ,  $Q_k' \in \text{CON}(X_k)$ ,  $Q \in \text{SYM}(X)$ , and  $Q' \in \text{CON}(X)$  satisfy i) and ii).

The induction hypothesis gives that there exists a tree  $t$  with a subtree  $t'$  such that

- 1)  $\text{con}(t') = Q'$
- 2)  $\text{sym}(t') = Q$
- 3) there is a path  $d \rightarrow^* c$  in  $D(t)$  where  $c \in S(Z)$ .

We can choose trees  $t_j \in \text{DOM}(X_j)$  such that  $\text{sym}(t_j) = Q_j$  for  $1 \leq j \leq n$ . Condition i) and ii) ensure that  $\text{con}(tk) = Q_k'$ . Since  $b \rightarrow^* d$  is in  $D(p) \mathbb{I} Q_1 Q_2 \dots Q_n \mathbb{I}$  it will also be in  $D(t')$  where  $t' = X[t_1 t_2 \dots t_n]$  and thereby  $b \rightarrow^* c$  will be a path in  $D(t)$ .

///

A consequence of this Lemma is

#### THEOREM 7:

For any AG  $G = (V, B, R, Z)$  over a semantic domain  $\mathcal{D}$  there exists an AG  $G' = (V', B, R', Z)$  over the same semantic domain without useless attributes such that  $G$  and  $G'$  have the same underlying grammar and such that

$$\underline{I}(G, \mathcal{D}) = \underline{I}(G', \mathcal{D}).$$

PROOF: Determine the useless attributes by algorithm 1. Let  $A(X') = A(X) - U(X)$  for each symbol  $X'$  in  $V' = V$  and let  $R'$  consist of the productions from  $R$  without semantic rules for the useless attributes. If attribute variables for useless attributes are used in semantic rules for not useless attributes then these attribute variables are replaced by arbitrary values from the appropriate domains. It is easy to show that the two AGs specify the same translation. The theorem then follows from the lemma.

///

#### EXAMPLE 9:

The AG from example 1 is transformed into the AG with the following productions:

$p1': \langle Z \uparrow b \rangle ::= B \langle X \psi a \uparrow b \uparrow a \rangle$   
 $p2': \langle Z \uparrow b \rangle ::= C \langle Y \uparrow b \rangle$   
 $p3': \langle Y \uparrow b \rangle ::= \langle X \psi 1 \uparrow b \uparrow c \rangle$   
 $p4': \langle X \psi a \uparrow a \uparrow 1 \rangle ::= A1$   
 $p5': \langle X \psi a \uparrow 0 \uparrow 7 \rangle ::= A2$

///

The construction in theorem 7 only removes those attributes which never will influence the value of the attributes of the root of a derivation tree. But there may still be attributes associated with a derivation tree whose values do not influence the meaning. An example of this is e.g. the production  $p5'$  of the AG in example 10. The third attribute of  $X$  is assigned an arbitrary value (7) exactly as the construction allows. But if  $X$  is expanded with this production in a derivation tree then the value of the third attribute of  $X$  will never influence the meaning. On the other hand we cannot remove the attribute because if production  $p4'$  is used to expand  $X$  then the value of the attribute may influence the meaning.

This indicates that it is more complicated to remove attributes such that the value of any attribute for any symbol in any derivation tree influences the meaning. In fact we have to change the underlying grammar of the AG. This leads to the introduction of the concept of a reduced AG.

**DEFINITION 8:**

Let  $G = (V, B, R, Z)$  be an AG over a semantic domain  $\mathcal{D}$ . Then  $G$  is reduced if and only if for every symbol  $X$  and every tree  $t \in \text{DOM}(Z)$  where  $X$  occur we have:

$$b \in A(X) \iff \text{there is a } c \in S(Z) \text{ such that } b \rightarrow^* c \text{ in } D(t)$$

///

In the following we will see how a well-defined AG can be transformed into a reduced AG. For simplicity we will give the construction for the partly uniform AGs. By application of the construction in theorem 5 the result can easily be extended to well-defined AGs.

**THEOREM 8:**

Let  $G = (V, B, R, Z)$  be a partly uniform and well-defined AG over a semantic domain  $\mathcal{D}$  containing the identity function for each lattice. Then there exists a reduced AG  $G' = (V', B', R', Z')$  over the same semantic domain  $\mathcal{D}$  such that

$$\underline{I}(G, \mathcal{D}) = \underline{I}(G', \mathcal{D})$$

CONSTRUCTION: First we construct a CFG  $G'' = (Vn'', Vt'', R'', Z'')$  where

$Vn''$ :  $\{(X, A) \mid X \in V \text{ and } A \subseteq A(X) \text{ satisfies } (*)\}$

$$A \cap I(X) = \{b \in I(X) \mid \text{there is a } c \in A \cap S(X) \text{ such that } b \rightarrow^* c \text{ is in } Q \text{ where } \text{SYM}(X) = \{Q\}\} \quad (*)$$

$Vt''$ :  $Vt$

$R''$ : If  $p: X ::= X_1 X_2 \dots X_n$  is in  $R$  then we have a production

$$p': (X, A) ::= (X_1, A_1) (X_2, A_2) \dots (X_n, A_n)$$

in  $R''$  if the following requirements are satisfied:

i)  $(X, A)$  and  $(X_j, A_j)$  for  $1 \leq j \leq n$  are in  $Vn''$

ii) for  $1 \leq k \leq n$

$$A_k \cap S(X_k) = \{c \in S(X_k) \mid \text{there is a } b \in S(X) \cap A \text{ such that } c \rightarrow^* b \text{ is in } D(p) \cap Q_1 Q_2 \dots Q_n \text{ where } \text{SYM}(X_j) = \{Q_j\} \text{ for } 1 \leq j \leq n\}$$

If  $(X, A) \in Vn''$  and  $X \in Vt$  then  $(X, A) ::= X$  is a production in  $R''$ .

$Z''$ :  $(Z, S(Z))$ .

Let  $Gu' = (Vn', Vt', Ru', Z')$  be the reduced CFG constructed from  $G''$ .  $Gu'$  will be the underlying grammar of  $G'$ . To  $(X, A)$  in  $Vn'$  we associate the set of attributes  $A$  separated into the two sets  $A \cap S(X)$  and  $A \cap I(X)$  of synthesized resp. inherited attributes.

If  $p': (X, A) ::= (X_1, A_1) (X_2, A_2) \dots (X_n, A_n)$  is a production in  $Ru'$  then the attributes in  $A \cap S(X)$  and  $A_j \cap I(X_j)$  for  $1 \leq j \leq n$  are defined by semantic rules equal to those of the production  $p$  in  $R$ . The conditions  $(*)$ , i) and ii) ensure that this is possible.

If  $p': (X, A) ::= X$  is in  $Ru'$  then the semantic rules for the attributes in  $S(X, A)$  is given as the identity function applied to the attributes of  $X$ .

PROOF: We show three things:

1.  $\text{SYM}(X, A) = Q'$  where  $Q'$  is the subgraph of  $Q$  which only involves attributes from  $A$ ,  $\text{SYM}(X) = \{Q\}$ .
2. if  $(X, A)$  is a node in  $t$  then for any  $b \in A$  there is a  $c \in S(Z)$  such that  $b \rightarrow^* c$  is in  $D(t)$
3.  $\underline{I}(G, \mathcal{D}) = \underline{I}(G', \mathcal{D})$

1.  $\text{SYM}(X, A) = \{Q'\}$ . We will show that if  $t \in \text{DOM}(X, A)$ ,  $(X, A) \in V_n$ , then  $\text{sym}(t) = Q'$  where  $Q'$  is the subgraph of  $Q$  only involving attributes from  $A$  and where  $\text{SYM}(X) = \{Q\}$ . We use induction in the height of  $t$ .

If the height is one then the production applied at  $(X, A)$  is  $p': (X, A) ::= X$  and the assertion clearly holds.

For the induction step let the production applied at  $(X, A)$  be  $p': (X, A) ::= (X_1, A_1) (X_2, A_2) \dots (X_n, A_n)$ . Then  $D(p')$  is a subgraph of  $D(p)$ . The induction hypothesis gives that  $Q_j'$  is a subgraph of  $Q_j$ , where  $\text{SYM}(X_j) = \{Q_j\}$ . Therefore we have that  $Q' = D(p') \mathbb{I} Q_1' Q_2' \dots Q_n' \mathbb{I}$  is a subgraph of the graph  $Q = D(p) \mathbb{I} Q_1 Q_2 \dots Q_n \mathbb{I}$ . From that it follows that  $\text{SYM}(X, A) = \{Q'\}$  where  $Q'$  is the subgraph of  $Q$  formed by removing nodes not in  $A$  and the involved arcs.

2. for any  $b \in A$  there is a  $c \in S(Z)$  such that  $b \rightarrow^* c$  is in  $D(t)$ . To show this part of the theorem we use induction in the length of the path from the root  $(Z, S(Z))$  of a derivation tree  $t$  and to a node  $(X, A)$  in  $t$ . Let  $m$  be the length.

$m = 1$ : The production used to expand  $(Z, S(Z))$  is  $p': (Z, S(Z)) ::= (X_1, A_1) (X_2, A_2) \dots (X_n, A_n)$ . If  $b \in A_k \cap I(X_k)$  then condition (\*) gives that there is a  $c \in A_k \cap S(X_k)$  such that  $b \rightarrow c$  is in  $Q_k$  where  $\text{SYM}(X_k) = \{Q_k\}$ . From part 1 of the proof it follows that  $b \rightarrow c$  is in  $Q_k'$ .

Thus in order to show that there is a  $d \in S(Z)$  such that  $b \rightarrow^* d$  in  $D(t)$  it is sufficient to show that  $c \rightarrow^* d$  is in  $D(t)$ .

Therefore let  $c \in A_k \cap S(X_k)$ . Condition ii) gives that there is a  $d \in S(Z)$  such that  $c \rightarrow^* d$  is in  $D(p) \mathbb{I} Q_1 Q_2 \dots Q_n \mathbb{I}$ . If the length of the path  $c \rightarrow^* d$  is equal to 1 then clearly  $c \rightarrow d$  is in  $D(p) \mathbb{I} Q_1' Q_2' \dots Q_n' \mathbb{I}$ . If the length of the path is  $2 \cdot h + 1$  (finite since  $G$  is well-defined) then we assume that the path is  $c \rightarrow e \rightarrow f \rightarrow^* d$  where  $e \in A_j \cap I(X_j)$ ,  $f \in A_j \cap S(X_j)$ , and  $e \rightarrow f$  is in  $Q_j$ . Since  $f \in A_j$  and the path  $f \rightarrow^* d$  has length  $2 \cdot h - 1$  this path will also be in  $D(p') \mathbb{I} Q_1' Q_2' \dots Q_n' \mathbb{I}$ . Furthermore  $e \rightarrow f$  is in  $Q_j'$ . Since  $c \in A_k$ ,  $e \in A_j$  and  $c \rightarrow e$  is in  $D(p)$  we have  $c \rightarrow e$  in  $D(p')$ . The conclusion is that  $c \rightarrow^* d$  is in  $D(p') \mathbb{I} Q_1' Q_2' \dots Q_n' \mathbb{I}$ .

The induction step: Let  $p': (X, A) ::= (X_1, A_1) (X_2, A_2) \dots (X_n, A_n)$  be the production introducing the node  $(X_k, A_k)$  in  $t$ . As above we can show that if  $b \in A_k$  then there is a  $c \in A$  such that  $b \rightarrow^* c$  is in  $D(p') \mathbb{I} Q_1' Q_2' \dots Q_n' \mathbb{I}$  and thereby in  $D(t)$ . The induction hypothesis gives that there exists a  $d \in$

$S(Z)$  such that  $c \rightarrow^* d$  is a path in  $D(t)$  and thereby that  $b \rightarrow^* d$  is a path in  $D(t)$ .

3.  $\underline{I}(G, \mathcal{A}) = \underline{I}(G', \mathcal{A})$  We construct a homomorphism  $h$  by

- $h(X, A) = X$  for all  $X \in Vn'$ , and
- $h(X) = X$  for  $X \in Vt'$

We then have that if  $w \in \underline{L}(G', \mathcal{A})$  then  $w \in \underline{L}(G, \mathcal{A})$  and the meaning of  $w$  is the same for the two AGs.

If  $w \in \underline{L}(G, \mathcal{A})$  and we have a derivation tree  $t$  then we can decorate the nodes in  $t$  with subsets of attributes for the respective nodes. We define a mapping  $h'$ :

$$h'(X[t_1 t_2 \dots t_n], A) = (X, A)[h'(t_1, A_1) h'(t_2, A_2) \dots h'(t_n, A_n)]$$

where  $A$  and  $A_j$  for  $1 \leq j \leq n$  satisfy condition (\*), i), and ii).

$$h'(X, A) = (X, A)[X] \text{ if } X \in Vt$$

We start by letting  $g(t) = h'(t, S(Z))$  where  $t \in \text{DOM}(Z)$ . Then we have  $w \in \underline{L}(G', \mathcal{A})$  and  $w$  has the same meaning in the two AGs.

///

EXAMPLE 10:

Let us apply the construction in the AG in example 6. Let  $A(Z) = \{a\}$ ,  $A(X) = \{a, b, c, d\}$  and  $A(Y) = \{a, b\}$ . The underlying grammar  $G_{u'}$  will have the productions

$$p1': (Z, Q1, \{a\}) ::= (B, \emptyset) (X, Q2, \{a, c, d\})$$

$$p1'': (Z, Q1, \{a\}) ::= (B, \emptyset) (X, Q3, \{c\})$$

$$p2': (Z, Q1, \{a\}) ::= (C, \emptyset) (Y, Q4, \{b\})$$

$$p3': (Y, Q4, \{b\}) ::= (X, Q2, \{a, c\})$$

$$p3'': (Y, Q4, \{b\}) ::= (X, Q3, \{c\})$$

$$p4': (X, Q2, \{a, c, d\}) ::= (A1, \emptyset)$$

$$p4'': (X, Q2, \{a, c\}) ::= (A1, \emptyset)$$

$$p5': (X, Q3, \{c\}) ::= (A2, \emptyset)$$

$$p6': (B, \emptyset) ::= B$$

$$p7': (C, \emptyset) ::= C$$

$$p8': (A1, \emptyset) ::= A1$$

$$p9': (A2, \emptyset) ::= A2$$

The reduced AG can easily be constructed now.

///



Combination of the theorems 5 and 8 shows that any translation specified by a well-defined AG can be specified by a reduced AG.

#### 4. K-VISIT ATTRIBUTE GRAMMARS

---

In chapter 2 and 3 we have briefly concerned how to construct an evaluated semantic tree for a string. We will in this chapter define a device called an evaluator which performs this construction when applied to a derivation tree for the string. We only consider well-defined AGs although some of the constructions also may be applied to non-well-defined AGs.

An evaluator may be considered as consisting of two parts called resp. a traverser and an interpreter.

The traverser specifies how to traverse a derivation tree in order to evaluate attributes associated with the nodes of the tree and thereby to construct an evaluated semantic tree. The traverser will only care about the dependencies between the attributes. When an attribute can be evaluated the traverser will call the interpreter. The interpreter will then evaluate the attribute and store its value in the semantic tree.

The action of the traverser may be dynamically influenced by the values computed by the interpreter. We will however let the traverser be quite independent of the interpreter.

Because of the close correspondance between the semantic rules and the interpreter the main problem when constructing an evaluator will be to construct the traverser. This is the reason why we in the following almost ignore the interpreter. In section 4.1 I introduce more formally the concept of an evaluator.

As a natural extension of the one-visit AGs defined by [EnF79] I introduce in section 4.2 the k-visit AGs. An evaluator for a k-visit AG may have the property that when applied to a derivation tree of the grammar each node of the tree will be visited at most k times. It turns out that the k-visit AGs perform a proper hierarchy also with respect to translations when some conditions are satisfied.

In section 4.3 I give algorithms to construct an evaluator for an arbitrary well-defined AG. It is shown that any well-defined AG is k-visit

for some  $k$ . Furthermore we consider methods that may be used to determine bounds within which the minimal  $k$  for which an AG is  $k$ -visit can be found.

#### 4.1 DEFINITION OF AN EVALUATOR

As mentioned above the main task for the evaluator is to traverse a derivation tree and evaluate attributes associated with the nodes. It turns out that different strategies for traversing a tree may give rise to different subclasses of AGs. We define (rather informally):

##### DEFINITION 1:

Let  $t \in \text{DOM}(Z)$  be a derivation tree for a string  $w$ . An evaluation strategy for  $t$  is a way of traversing the nodes of  $t$  and evaluating attributes associated with the nodes.

///

The rest of this section is divided into two parts. First the concept of an evaluator is introduced. At last I give an overview of how an evaluator may be constructed for an AG.

We will regard that of traversing the nodes of  $t$  as a recursive routine taking a node as parameter - we say that the node is visited. At each visit to a node  $X$  we may first call the interpreter to evaluate some of the inherited attributes of  $X$ , after that we may visit some of the sons by recursive calls of the routine, and at last we may call the interpreter in order to evaluate some of the synthesized attributes of  $X$ . (We might have other interpretations of a visit.) A so-called plan will tell what to be done at the visit.

For each symbol  $X$  we may have a set of plans. At each visit to a node labelled  $X$  we have to choose one of these plans. This choice is based on two types of informations:

- a subset of the inherited attributes of  $X$  that can be or already are evaluated at the start of the visit. This set is called an input set and it will be a parameter to the recursive visit routine. The parameter summarizes the activity that has taken place since the last visit to the node.
- a state for the node. The state is used to remember information between

the visits to the node. The state may contain information such as which production is applied at the node and/or which attributes have been evaluated at previous visits. For each node there is an initial state.

Initially all the nodes of the derivation tree are in their initial state. The recursive routine is called with the root of the tree and the empty input set as parameters (the start symbol has no inherited attributes). When returning from that call the attributes of the tree have been evaluated.

We are now ready to state more precisely what kind of information that will be available in a plan for a node labelled  $X$ . Let us assume that the plan is chosen on the basis of an input set  $I$  and some state  $s$ .

- i) The plan will contain a subset  $I'$  of  $I$  of those attributes of  $X$  that have not been evaluated before. Since all attributes of  $I$  can be or already are evaluated it will be possible to evaluate the attributes in  $I'$ .
- ii) The plan will contain a specification of an order in which the sons of the node have to be visited. Each son may be visited zero, one or more times in any order. For each visit to a son is specified an input set. The input set will contain those inherited attributes of the son which now either are or can be evaluated. A sequence of pairs of sons and input sets will be called a visiting sequence.
- iii) The plan will contain a subset  $S'$  of the synthesized attributes of  $X$ .  $S'$  will contain those attributes which not already have been evaluated but which can be evaluated when the attributes in  $I$  are known.

More formally we define

DEFINITION 2:

Let  $X \in V$  be a symbol in the AG  $G$  and let  $p: X ::= X_1 X_2 \dots X_n$  be a production. A plan for  $X$  and  $p$  is a triple  $(I', vs, S')$  where

- i)  $I' \subseteq I(X)$
- ii)  $vs$  is a visiting sequence for  $p$ :  
 $vs = (X_{j1}, I_1)(X_{j2}, I_2) \dots (X_{jm}, I_m)$   
 where  $I_h \subseteq I(X_{jh})$  for  $1 \leq jh \leq n$  and  $1 \leq h \leq m$
- iii)  $S' \subseteq S(X)$

///

As mentioned above a plan is chosen on the basis of an input set and a state. We will use a table called PLAN to contain this information. A visit to a node may change some of the information stored in the state of the node and then we want to update the state. A table called GOTO will be used to hold information about how the state of the node may change.

The PLAN and the GOTO tables will be parts of an evaluator. The evaluator will also contain information about how the initial states associated with a derivation tree are determined.

**DEFINITION 3:**

An evaluator  $\underline{E}(G)$  for an AG  $G = (V, B, R, Z)$  over a semantic domain  $\mathcal{D}$  is a 5-tuple:

$$\underline{E}(G) = (\underline{S}, \varepsilon_0, \underline{I}, \text{PLAN}, \text{GOTO})$$

where

- $\underline{S}$ : a finite set of (evaluation) states
- $\varepsilon_0$ : a function assigning an initial state to each node in any derivation tree for  $G_u$
- $\underline{I}$ : a set of input sets
- PLAN:  $\underline{S} \times \underline{I} \rightarrow \{\text{plans}\}$ , a function determining a plan for a given state and input set
- GOTO:  $\underline{S} \times \underline{I} \rightarrow \underline{S}$ , a function determining a new state on the basis of a given state and input set

///

**EXAMPLE 1:**

Consider an AG  $G$  over a semantic domain  $\mathcal{D} = (\underline{D}, \underline{F})$ .  $\underline{D}$  contains a flat lattice constructed by extension of the set  $\{A, B, C\}^*$  and  $\underline{F}$  contains four operations: concatenation with  $A$ ,  $B$  and  $C$  (denoted by  $A \sim x$ ,  $B \sim x$  and resp.  $C \sim x$  where  $x$  is the parameter), and the identity operation.

Let  $I(X) = \{a, b\}$ ,  $S(X) = \{c, d\}$  and  $S(Z) = \{e, f\}$ . In  $G$  we have the following productions:

- $p_1: \langle Z \uparrow s_3 \uparrow s_4 \rangle ::= \langle X \downarrow s_1 \downarrow \lambda \uparrow s_2 \uparrow s_3 \rangle \langle X \downarrow s_2 \downarrow \lambda \uparrow s_1 \uparrow s_4 \rangle$
- $p_2: \langle X \downarrow s_1 \downarrow s_2 \uparrow A \sim s_3 \uparrow s_4 \rangle ::= A \langle X \downarrow A \sim s_1 \downarrow s_2 \uparrow s_3 \uparrow s_4 \rangle$
- $p_3: \langle X \downarrow s_1 \downarrow s_2 \uparrow B \sim s_3 \uparrow s_4 \rangle ::= B \langle X \downarrow B \sim s_1 \downarrow s_2 \uparrow s_3 \uparrow s_4 \rangle$
- $p_4: \langle X \downarrow s_1 \downarrow s_2 \uparrow C \sim s_2 \uparrow C \sim s_1 \rangle ::= C$

$G$  specifies the translation

$$\underline{I}(G, \emptyset) = \{(w_1cw_2c, (cw_2'w_1c, cw_1'w_2c))\}$$

$w_1, w_2 \in \{A, B\}^+$  and  $w_j'$  is  $w_j$  reversed for  $j=1, 2$

An evaluator for  $G$  is  $\underline{E}(G) = (\underline{S}, s_0, \underline{I}, \text{PLAN}, \text{GOTO})$  where

$$\underline{S} = \{s_j \mid 0 \leq j \leq 11\}$$

$s_0$ : if the production used at the node is  $p_j$  then the state is  $s_j$  for  $1 \leq j \leq 4$ . If the node is a leaf then the state is  $s_0$

$$\underline{I} = \{\{a, b\}, \{b\}, \emptyset\}$$

The PLAN and GOTO tables are ( $X_j$  refers to the  $j$ 'th son of  $Z$  for  $j=1, 2$ ):

$\underline{S} \times \underline{I}$	PLAN	GOTO
$s_1, \emptyset$	$\emptyset, (X_2, \{b\})(X_1, \{a, b\})(X_2, \{a, b\}), \{e, f\}$	$s_5$
$s_2, \{b\}$	$\{b\}, (X, \{b\}) \{c\}$	$s_6$
$s_3, \{b\}$	$\{b\}, (X, \{b\}) \{c\}$	$s_7$
$s_4, \{b\}$	$\{b\}, \lambda, \{c\}$	$s_8$
$s_2, \{a, b\}$	$\{a, b\}, (X, \{a, b\}), \{c, d\}$	$s_9$
$s_3, \{a, b\}$	$\{a, b\}, (X, \{a, b\}), \{c, d\}$	$s_{10}$
$s_4, \{a, b\}$	$\{a, b\}, \lambda, \{c, d\}$	$s_{11}$
$s_6, \{a, b\}$	$\{a\}, (X, \{a, b\}), \{d\}$	$s_9$
$s_7, \{a, b\}$	$\{a\}, (X, \{a, b\}), \{d\}$	$s_{10}$
$s_8, \{a, b\}$	$\{a\}, \lambda, \{d\}$	$s_{11}$

///

The evaluator is used to find the meaning of a string  $w$  with derivation tree  $t$  in the following manner:

ALGORITHM 1:

Input: an evaluator  $\underline{E}(G) = (\underline{S}, s_0, \underline{I}, \text{PLAN}, \text{GOTO})$  for an AG  $G$ ,  
a string  $w$  with a derivation tree  $t$  in  $G_u$ .

Output: a semantic tree  $t''$  for  $w$ .

Method:

1. let each node in  $t$  be in its initial state determined by  $s_0$  and let  $t'$  be the initial semantic tree for  $w$  determined from  $t$
2. perform the procedure VISIT( $Z, \emptyset$ )
3. let  $t''$  be the resulting semantic tree.

PROCEDURE VISIT( $X, I$ )

/\*  $X \in V$  is a node in  $t$ ,  $I \subseteq I(X)$  is an input set \*/

1. let  $s$  be the state of the node  $X$
2. let  $PLAN(s, I) = (I', vs, S')$  where  $vs = (X_{j1}, I_1)(X_{j2}, I_2) \dots (X_{jm}, I_m)$
3. call the interpreter to evaluate the attributes in  $I'$ ;  
     FOR  $h := 1$  TO  $m$  DO VISIT( $X_{jh}, I_h$ );  
     call the interpreter to evaluate the attributes in  $S'$ ;
4. let GOTO( $s, I$ ) be the new state of the node  $X$ .

///

Let us informally describe what happens when the procedure calls the interpreter to evaluate the attributes in a set  $A$ :

- for each attribute  $b \in A$  the actual semantic rule is used to compute the value of the attribute
- if the actual node  $X$  has a corresponding label  $\langle X, (v_1, v_2, \dots, v_r) \rangle$  in the semantic tree then this label is changed to  $\langle X, (u_1, u_2, \dots, u_r) \rangle$  where  $u_j = v_j$  if the attribute is not in  $A$ , otherwise  $u_j$  is the computed value of the attribute,  $1 \leq j \leq r$ .

To each derivation tree we can associate a sequence of attributes of the nodes specifying the order in which the attributes are evaluated. An attribute  $b$  precedes an attribute  $c$  in this sequence if  $b$  is evaluated before  $c$  by the algorithm above. We define a computation sequence for  $t$  as a sequence of sets of attributes where the attributes in each set can be evaluated in parallel. These sets are determined by the various plans of the evaluator.

**DEFINITION 4:**

Let  $t$  be a derivation tree with states from  $\mathcal{S}$  associated with the nodes. A computation sequence for a visit 'VISIT( $X, I$ )' to a node  $X$  in state  $s$  is

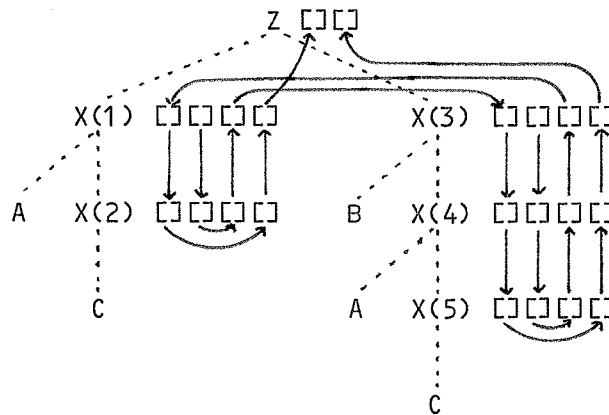
$cs(X, I, s) = I' \ cs(X_{j1}, I_1, s_1) \ cs(X_{j2}, I_2, s_2) \ \dots \ cs(X_{jm}, I_m, s_m) \ S'$   
 where  $PLAN(s, I) = (I', vs, S')$ ,  $vs = (X_{j1}, I_1)(X_{j2}, I_2) \dots (X_{jm}, I_m)$  and the state of  $X_{jh}$  is  $s_h$  for  $1 \leq jh \leq n$  and  $1 \leq h \leq m$ .

The computation sequence for  $t$  is  $cs(Z, \emptyset, s_0)$  where  $s_0$  is the initial state of the root of  $t$ ; all the nodes of  $t$  are in their initial states.

///

**EXAMPLE 2:**

Consider the string ACBAC and its derivation tree  $t$  according to the AG of example 1. The dependency network for  $t$  is



The computation sequence for  $t$  is

{b3}-{b4}-{b5}-{c5}-{c4}-{c3}-{a1, b1}-{a2, b2}-{c2, d2}-{c1, d1}-{a3}-{a4}-{a5}-  
 {d5}-{d4}-{d3}-{e, f}

///

When making an evaluator for an AG we have to consider the following problems:

1. What kind of information shall we store in the states of  $\underline{S}$
2. Given a state  $s$  and an input set  $I$  how can we determine  $PLAN(s, I)$  and  $GOTO(s, I)$
3. How can we determine a complete set of  $PLAN$  and  $GOTO$  entries.

Different choices of information to be stored in the states may lead to different subclasses of AGs.

When determining the  $PLAN$  and  $GOTO$  entries for the state  $s$  and the input set  $I$  we have the problem of finding a strategy for constructing a visiting sequence. Some methods will be discussed in this and the next chapter. Also here different choices may result in different subclasses of AGs.

The third problem, how to make a complete set of  $PLAN$  and  $GOTO$  entries, can be solved in at least two ways. One possibility is to take all combinations of a state and an input set and make a  $PLAN$  and a  $GOTO$  entry for each of them. But this may lead to many entries that never will be used. Another possibility is to simulate all events that can occur and only make those entries that may be used. We will choose that approach.



We start by constructing a PLAN and GOTO table entry for the initial visit to the root of a derivation tree. By inspection of the plans made so far one can determine what PLAN and GOTO table entries that are required. If they not already are present they must be added. This process is repeated until no new PLAN and GOTO entries can be found.

Consider a visiting sequence  $vs$  of a plan. For each pair  $(X,I)$  in  $vs$  we will call the procedure VISIT. If the node  $X$  is in the state  $s$  then the visit will cause lookups  $PLAN(s,I)$  and  $GOTO(s,I)$ . Thus if we know which states a node may be in we can easily find the PLAN and GOTO table entries that are required.

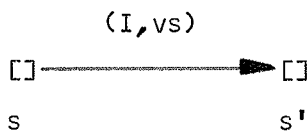
The state that a node is in at any point of time is the state it was left in by the previous visit or it is its initial state. In order to determine the state we must be able to determine which visits there have been to the node until now. That is we have to consider all the previous plans that have been 'executed' at the node. To keep track of that we introduce the concept of a history graph (inspired by [Kew76]):

DEFINITION 5:

A history graph is a directed graph whose nodes are labelled with states and whose arcs are labelled with pairs of input sets and visiting sequences.

///

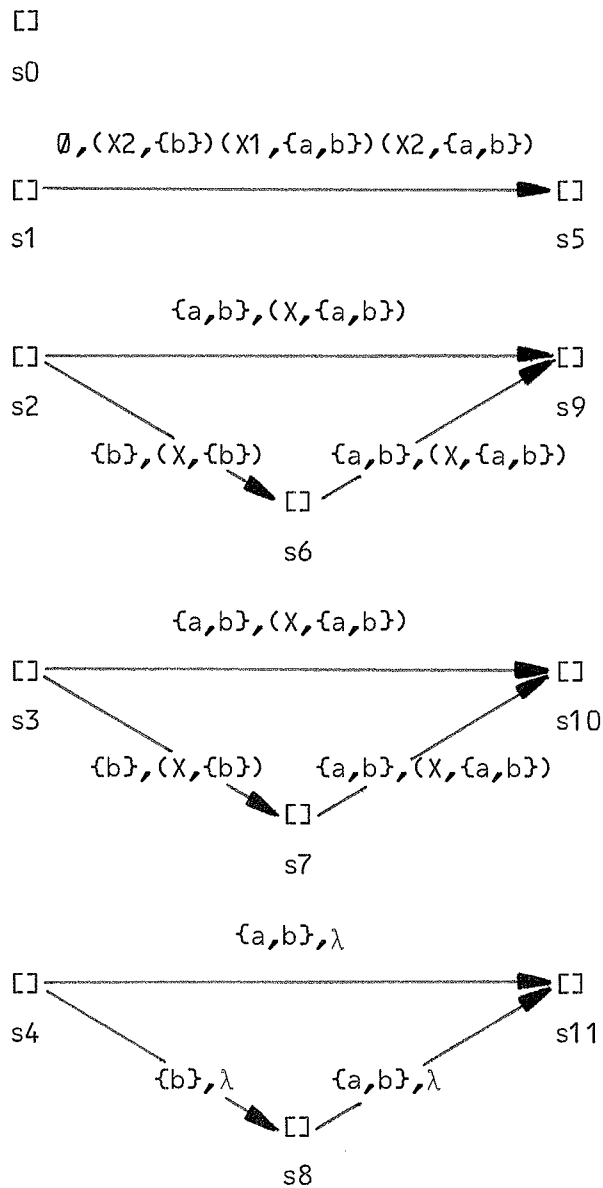
A path through the history graph represents a sequence of events which could take place at a node during the evaluation. Let us consider an arc in the history graph:



If a node in a derivation tree is in state  $s$  and we visit the node with an input set  $I$  then the sons of the node will be visited in a sequence described by  $vs$  and the node will be left in the state  $s'$ . Thus we will have  $GOTO(s,I) = s'$  and  $PLAN(s,I) = (I',vs,S')$  for some sets  $I'$  and  $S'$ .

EXAMPLE 3:

The history graph for the PLAN and GOTO tables in example 1 is:



///

The history graph is useful because it becomes easy to see in which order the sons are visited even across the boundaries of the single plans.

There may be several arcs ending at a node in the history graph and several arcs beginning at the node. If we take a path leading to a specific node in the history graph we can concatenate the visiting sequences labeling the arcs. On the basis of this composite visiting sequence it becomes possible to determine the states in which the sons may be when making a visit to their father. How it more precisely can be done depends on how the evaluation strategy is.

Initially the history graph will contain a node for each initial

state. During the construction of the PLAN and GOTO table entries it will be extended.

DEFINITION 6:

An initial history graph is a history graph with one node for each initial state and no arcs.

///

#### 4.2 THE K-VISIT PROPERTY

In this section I introduce the concept of a k-visit AG. We consider both the translational and the traditional approach. We can note that the results in this section rely very much on the concept of a visit introduced in the previous section (i.e. algorithm 1). We define:

DEFINITION 7:

A k-visit evaluation strategy for a tree t is an evaluation strategy for t where each node is visited at most k times.

///

DEFINITION 8:

An AG is a (translational) k-visit AG if for each tree  $t \in \text{DOM}(Z)$  there exists a k-visit evaluation strategy which computes  $\text{meaning}(w,t)$  where t is a derivation tree for w.

An AG is a traditional k-visit AG if for each tree  $t \in \text{DOM}(Z)$  there exists a k-visit evaluation strategy which computes  $\text{meaning}'(w,t)$  where t is a derivation tree for w.

///

The AG in example 1 is a 2-visit AG. It is easy to see that it also is traditional 2-visit.

Clearly a k-visit AG will also be a (k+1)-visit AG. I will now give an example of a translation specified by a k-visit AG which cannot be specified by any (k-1)-visit AG over the same semantic domain and with the same underlying grammar.

Let us first specify the underlying grammar  $G_u = (V_n, V_t, R_u, Z)$  of the

AG:

$V_n = \{X, Z\}$

$V_t = \{A\}$

Ru:  $Z ::= X$

$X ::= A X$

$X ::= A$

We will define the translation

$$\text{TAU}(k) = \{(w, w^{2k}) \mid w \in \{A\}^+\}$$

This translation can be specified by a  $k$ -visit AG  $G_k$  over a semantic domain  $\mathcal{D} = (\underline{D}, \underline{E})$  and with the underlying grammar  $G_u$ .  $\underline{D}$  will contain a single flat lattice STRINGS constructed as an extension of the set  $\{A\}^+$ . In  $\underline{E}$  we have two functions, the identity function and the function that concatenates a string with the symbol  $A$  (denoted  $A \sim x$  where  $x$  is the parameter).

Let  $G_k = (V, B, R, Z)$  be defined by

V:  $V_n = \{X, Z\}$

$V_t = \{A\}$

$I(X) = \{x_1, x_2, \dots, x_k\}$   $S(X) = \{y_1, y_2, \dots, y_k\}$

$I(Z) = \emptyset$   $S(Z) = \{z_1\}$

$I(A) = \emptyset$   $S(A) = \emptyset$

where for  $1 \leq j \leq k$   $x_j, y_j, z_1$ : STRINGS.

B: for  $1 \leq j \leq k$   $s_j, s_j'$ : STRINGS

R:  $p_1: \langle Z \uparrow s_k \rangle ::=$

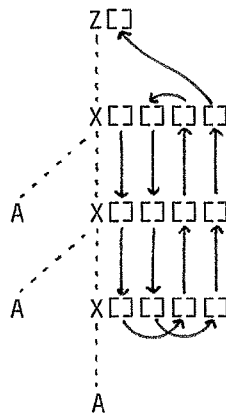
$\langle X \psi A \psi(A \sim s_1) \psi(A \sim s_2) \dots \psi(A \sim s_{(k-1)}) \uparrow s_1 \uparrow s_2 \dots \uparrow s_k \rangle$

$p_2: \langle X \psi s_1 \psi s_2 \dots \psi s_k \uparrow(A \sim s_1') \uparrow(A \sim s_2') \dots \uparrow(A \sim s_k') \rangle ::=$

$A \langle X \psi(A \sim s_1) \psi(A \sim s_2) \dots \psi(A \sim s_k) \uparrow s_1' \uparrow s_2' \dots \uparrow s_k' \rangle$

$p_3: \langle X \psi s_1 \psi s_2 \dots \psi s_k \uparrow(A \sim s_1) \uparrow(A \sim s_2) \dots \uparrow(A \sim s_k) \rangle ::= A$

A dependency network for the derivation tree for the string AAA with  $k = 2$  is:



To see that  $G_k$  is a  $k$ -visit AG we construct an evaluator

$$\underline{E}(G_k) = (\underline{S}, s_0, \underline{I}, \text{PLAN}, \text{GOTO})$$

where

$$\underline{S} = \{s_1, s_2, s_3, s_4\} \cup \{s_{ij} \mid i=2 \text{ or } i=3 \text{ and } 1 \leq j \leq k\}$$

$s_0$ : if the production applied at a node is  $p_j$  then the initial state of the node is  $s_j$ ,  $1 \leq j \leq 3$ .

$\underline{I} = \{I_j \mid 0 \leq j \leq k\}$  where  $I_j = \{x_1, x_2, \dots, x_j\}$  for  $j > 0$  and  $I_0 = \emptyset$

$\underline{S} \times \underline{I}$	PLAN	GOTO
$(s_1, I_0)$	$\emptyset, (X, I_1)(X, I_2) \dots (X, I_k)$	$\{z_1\}$ s4
$(s_2, I_1)$	$\{x_1\}, (X, I_1), \{y_1\}$	s21
$(s_{21}, I_2)$	$\{x_2\}, (X, I_2), \{y_2\}$	s22
:	:	:
$(s_{2(k-1)}, I_k)$	$\{x_k\}, (X, I_k), \{y_k\}$	s2k
$(s_3, I_1)$	$\{x_1\}, \lambda, \{y_1\}$	s31
$(s_{31}, I_2)$	$\{x_2\}, \lambda, \{y_2\}$	s32
:	:	:
$(s_{3(k-1)}, I_k)$	$\{x_k\}, \lambda, \{y_k\}$	s3k

The translation  $\text{TAU}(k)$  cannot be specified by a  $(k-1)$ -visit AG over the same semantic domain and with the same underlying grammar. To show that we assume the contrary. Let  $G_{k'}$  be a  $(k-1)$ -visit AG over the semantic domain  $\mathcal{D}$  and with underlying grammar  $G_u$  and assume that  $G_{k'}$  specifies the translation  $\text{TAU}(k)$ . Without loss of generality we can assume that there are no useless attributes in  $G_{k'}$  (according to theorem 8 in chapter 3).

Consider a string  $A^n$  with  $n > k + C$  where  $C$  is the maximal length of a constant value occurring in the semantic rules of  $G_k'$ . Let  $t$  be the derivation tree of  $A^n$ . Let the computation sequence for  $t$  be  $A_1 A_2 \dots A_m$ . Since  $G_k'$  has no useless attributes  $A_j$  will be a subset of the attributes in  $A(X)$  for  $1 \leq j < m$  and  $A_m = S(Z)$  (there are no visits to the leaves of  $t$  since inherited attributes of terminal symbols always will be useless). An attribute in a set  $A_j$  can only depend on an attribute in  $A_i$  where  $i < j$ . Thus the length of a string which is the value of an attribute in  $A_j$  can at most be  $j + C$ , and thereby the length of the translation (the value of an attribute in  $A_m$ ) can be at most  $m + C$ .

Since  $G_k'$  is a  $(k-1)$ -visit AG we have at most  $k-1$  visits to each node in  $t$ . Each visit may result in 2 sets of attributes in the computation sequence. There are  $n+1$  interior nodes in  $t$  which each is visited at most  $k-1$  times (the leaves of  $t$  will not be visited at all). Thus the computation sequence will contain at most  $2 \cdot (k-1) \cdot (n+1)$  sets. But the length of the translation of  $A^n$  is  $2 \cdot k \cdot n$  and since  $2 \cdot k \cdot n > 2 \cdot (k-1) \cdot (n+1) + C$  we have a contradiction:  $G_k'$  cannot specify the translation  $\tau(k)$ . This shows the theorem:

**THEOREM 1:**

There exists a semantic domain  $\mathcal{D}$  such that translations specified by  $k$ -visit AGs over that domain with the same underlying grammar define a proper hierarchy.

///

It may be shown that the theorem also holds in the traditional approach.

**4.3 CONSTRUCTION OF AN EVALUATOR FOR A K-VISIT AG**

In this section we will give algorithms that construct an evaluator  $E(G)$  for a well-defined AG  $G$ . We will assume that  $I(X) = \emptyset$  if  $X \notin V_t$  that is there will never be a visit to a leaf in a derivation tree. As mentioned in section 4.1 we will consider three 'problems':

1. choice of evaluation states
2. making an entry in the PLAN and GOTO tables
3. construction of the evaluator

## 1. Choice of evaluation states

At a visit to a node  $X$  in a derivation tree we choose a plan on the basis of an input set  $I$  and the state  $s$  of the node. That is we determine:

- i) which attributes of  $I(X)$  can be evaluated
- ii) in which order shall we visit the sons and with which input sets
- iii) which attributes of  $S(X)$  can be evaluated

We will let the state contain information about

- the production  $p: X ::= X_1 X_2 \dots X_n$  applied at the node
- the SYM-graphs for the nodes  $X, X_1, X_2, \dots, X_n$
- the set  $A$  of attributes of the symbols in  $p$  which are known

Let us see that this information is enough for construction of a plan.

If we have the input set  $I$  and the set  $A$  of known attributes then the attributes of i) will be  $I - (A \cap I(X))$ .

Since we know the dependencies between the attributes of the node  $X$  it is also possible to determine which synthesized attributes of  $X$  that can be evaluated when the inherited attributes of  $I$  are known. If  $Q$  is a dependency graph for  $X$  then we define:

$$\text{YIELD-}_s(X, Q, I) = \{b \in S(X) \mid \text{if } c \rightarrow b \text{ is in } Q \text{ then } c \in I\}$$

The attributes in iii) will be  $\text{YIELD-}_s(X, Q, I) - (A \cap S(X))$ . In order to have that we must require that all the attributes in the subtree with root  $X$  which depend on attributes in  $I$  are evaluated. This will be ensured in the construction of the visiting sequence.

We will require that when we visit a son  $X_j$  of  $X$  then some of the attributes of  $X_j$  which not already are evaluated will be evaluated. Since we in the state have information about which attributes of symbols in  $p$  that have been evaluated we can determine which attributes of a son  $X_j$  that can be evaluated:

$$\text{YIELD-}_i(X_j, D(p), A) = \{b \in I(X_j) \mid \text{if } c \rightarrow b \text{ is in } D(p) \text{ then } c \in A\}$$

The attributes of  $I_j = \text{YIELD-}_i(X_j, D(p), A)$  may be the input set to a visit to  $X_j$ .

After the visit to  $X_j$  we may know some synthesized attributes of  $X_j$  which may be used when visiting the other sons of  $X$ . We can use the dependency graph  $Q_j$  of  $X_j$  to determine the synthesized attributes of  $X_j$  known

after the visit to  $X_j$ . It will be the attributes in  $YIELD-s(X_j, Q_j, I_j)$ .

Formally we define

DEFINITION 9:

An evaluation state is a pair  $(p, A)$  where  $p: X ::= X_1 X_2 \dots X_n$ ,  $p+ = (p, Q, Q_1, Q_2, \dots, Q_n)$  where  $Q \in SYM(X)$  can be derived from  $D(p)$  and  $Q_1, Q_2, \dots, Q_n$  where  $Q_j \in SYM(X_j)$  for  $1 \leq j \leq n$ .  $A$  is a subset of the attributes for the symbols in  $p$ .

///

DEFINITION 10:

An initial state is an evaluation state  $(p, A)$  where  $p: X ::= X_1 X_2 \dots X_n$ ,  $A = S(X_{i1}) \cup S(X_{i2}) \cup \dots \cup S(X_{im})$  and  $\{X_{ij} \mid 1 \leq i \leq n, 1 \leq j \leq m\} = \{X_h \mid X_h \in V_t \text{ and } 1 \leq h \leq n\}$

///

It is easy to construct an algorithm which when applied to a derivation tree associates an initial state to each node (the mapping sym of theorem 1 in chapter 3 may be used).

DEFINITION 11:

A final state is an evaluation state  $(p, A)$  where  $p: X ::= X_1 X_2 \dots X_n$  and  $A = A(X_1) \cup A(X_2) \cup \dots \cup A(X_n)$ .

///

EXAMPLE 4:

For the AG in example 1 we have

$SYM(Z): Q_1: [ ] [ ]$        $SYM(X): Q_2: [ ] [ ] [ ] [ ]$ 


The states in the evaluator in example 1 are (the terminal symbols are ignored; indices refer to the positions of the symbols in a production):

Initial states

- $s_1 = ((p_1, Q_1, Q_2, Q_2), \emptyset)$
- $s_2 = ((p_2, Q_2, Q_2), \emptyset)$
- $s_3 = ((p_3, Q_2, Q_2), \emptyset)$
- $s_4 = ((p_4, Q_2), \emptyset)$

Final states

- $s_5 = ((p_1, Q_1, Q_2, Q_2), A(X_1) \cup A(X_2) \cup A(Z))$
- $s_9 = ((p_2, Q_2, Q_2), A(X_0) \cup A(X_1))$
- $s_{10} = ((p_3, Q_2, Q_2), A(X_0) \cup A(X_1))$
- $s_{11} = ((p_4, Q_2), A(X))$



Other states

s6 = ((p2,Q2,Q2),{b0,c0,b1,c1})

s7 = ((p3,Q2,Q2),{b0,c0,b1,c1})

s8 = ((p4,Q2),{b,c})

///

2. Making an entry in the PLAN and GOTO tables.

Above we have already intuitively seen how to construct a plan. Without further comment I present the algorithm:

ALGORITHM 2:

Input: a state  $s = (p+, A)$  where  $p: X ::= X_1 X_2 \dots X_n$ ,  
 $p+ = (p, Q, Q_1, Q_2, \dots, Q_n)$ , and a set  $I \subseteq I(X)$

Output: PLAN(s, I) and GOTO(s, I)

Method:

1.  $A_p := A \cup I$ ;  
 $I' := I - (A \cap I(X))$ ;  
 $vs := \lambda$
2. REPEAT
  - a: FOR  $j := 1$  TO  $n$  DO  $I_j := \text{YIELD-}i(X_j, D(p), A_p)$
  - b: choose  $X_j$  such that  
 $\text{YIELD-}s(X_j, Q_j, I_j) - (A_p \cap S(X_j)) \neq \emptyset$  or  $I_j - (A_p \cap I(X_j)) \neq \emptyset$ ;
  - c:  $vs := vs(X_j, I_j)$ ;  
 $A_p := A_p \cup I_j \cup \text{YIELD-}s(X_j, Q_j, I_j)$ ;UNTIL no choice is possible in step 2b
3.  $S' := \text{YIELD-}s(X, Q, I) - (A \cap S(X))$ ;  
PLAN(s, I) := (I', vs, S');  
GOTO(s, I) := (p+, (A\_p \cup S'));

///

The algorithm will stop since there are only a finite number of attributes for the symbols in  $p$  and since each visit to a son of  $X$  increases the set  $A_p$ .

We can furthermore note that the algorithm is nondeterministic in step 2b. Different choices will lead to different evaluators (with different properties).

It turns out that it is convenient to introduce a special property of the states, completeness:

DEFINITION 12:

A state  $(p^+, A)$  of a node  $X$  is made complete by a set  $I \subseteq I(X)$  if  
 $A = \{b \in A(p) \mid \text{if } c \rightarrow^* b \text{ is in } D(p) \text{ and } c \in I \text{ then } c \in I\}$   
 where  $p: X ::= X_1 X_2 \dots X_n$ ,  $p^+ = (p, Q, Q_1, Q_2, \dots, Q_n)$ ,  $Q \in \text{SYM}(X)$ ,  $Q_j \in \text{SYM}(X_j)$  for  $1 \leq j \leq n$  and  $A(p) = A(X) \cup A(X_1) \cup A(X_2) \cup \dots \cup A(X_n)$ .

///

LEMMA 1:

Consider a derivation tree  $t$  and a node  $X$  in  $t$ . Let  $s = (p^+, A)$  be the state of  $X$  and perform the call  $\text{VISIT}(X, I)$ . Let  $\text{GOTO}(s, I) = s'$  be constructed by algorithm 2. If  $s$  is an initial state or is made complete by a set  $I'' \subseteq I$  then  $s'$  is made complete by  $I$ .

PROOF: Let

$$A'' = \{b \in A(p) \mid \text{if } c \rightarrow^* b \text{ is in } D(p) \text{ and } c \in I \text{ then } c \in I\}$$

Let  $s' = (p^+, A')$ . We will show that  $A' = A''$ .

We use induction in the height of the subtree  $t'$  of  $t$  with root  $X$ .

Let the height of the subtree be one. Then  $\text{PLAN}(s, I) = (I', \lambda, S')$  where  $S' = \text{YIELD-}_s(X, Q, I)$  (by application of algorithm 2). We have  $A' = I' \cup S' \cup A$ . Furthermore we have

$$\begin{aligned} A'' &= \{b \in A(p) \mid \text{if } c \rightarrow^* b \text{ is in } D(p) \text{ then } c \in I\} \\ &= S(X_1) \cup S(X_2) \cup \dots \cup S(X_n) \cup I \cup \text{YIELD-}_s(X, Q, I) \\ &= A' \end{aligned}$$

For the induction step let  $t' = X[t_1 t_2 \dots t_n]$  where  $p: X ::= X_1 X_2 \dots X_n$  is the production applied at the root of  $t'$ . Let  $\text{PLAN}(s, I) = (I', \nu_s, S')$  where  $\nu_s = (X_{i1}, I_1)(X_{i2}, I_2) \dots (X_{im}, I_m)$ .

Let  $A_{pj}$  be the set of attributes of symbols of  $p$  that are known after the call ' $\text{VISIT}(X_{ij}, I_j)$ ' for  $0 \leq j \leq m$ . We have (from algorithm 2) that

$$\begin{aligned} A_{p0} &= A \cup I \\ A_{pj} &= A_{p(j-1)} \cup I_j \cup \text{YIELD-}_s(X_{ij}, \text{sym}(t_{ij}), I_j) \end{aligned}$$

Since  $I_j = \text{YIELD-}_i(X_{ij}, D(p), A_{p(j-1)})$  for  $j \geq 1$  we have (by application of the induction hypothesis for the trees  $t_{ij}$ ) that  $A_{pj} \subseteq A''$  for  $0 \leq j \leq m$ . And thereby easily that  $A' \subseteq A''$ .

In order to show the opposite inclusion let  $b \in A''$ . We will show that

$b \in A'$ .

Let  $d = b_0 \rightarrow b_1 \rightarrow b_2 \rightarrow \dots \rightarrow b_h = b$  be the longest path in the graph  $D(p) \text{ [sym}(t_1) \text{ sym}(t_2) \dots \text{sym}(t_n)]}$  ending at  $b$ . By induction in  $l$  we will show that  $b_l \in A'$ .

If  $l=0$  then the claim holds (if  $b_0 \in I$  it is obvious, otherwise it follows from the induction hypothesis) since it is an attribute of a subtree of  $t'$ .

For the induction step let  $b_l \in A'$ . If  $b_l \in I(X_j)$  for some  $j$  then  $b_{l+1} \in \text{YIELD-s}(X_j, \text{sym}(t_j), I_j')$  for some set  $I_j' \subseteq I(X_j)$  containing all attributes  $d$  such that  $d \rightarrow b_{l+1}$ . Then  $\text{YIELD-s}(X_j, \text{sym}(t_j), I_j') - (A \cap S(X_j)) \neq \emptyset$  and we will have a visit to  $X_j$  which evaluates  $b_{l+1}$  (the induction hypothesis) and  $b_{l+1} \in A'$ .

If  $b_l \in S(X_j)$  and  $b_{l+1} \in I(X_k)$  then  $b_{l+1} \in \text{YIELD-i}(X_k, D(p), B)$  for some set  $B$  of attributes that includes all attributes  $d$  such that  $d \rightarrow b_{l+1}$  is in  $D(p)$  and we will have a visit to  $X_k$  where  $b_{l+1}$  is evaluated (and  $b_{l+1} \in A'$ ).

If  $b_l \in S(X_j)$  and  $b_{l+1} \in S(X)$  (i.e.  $l = h-1$ ) then it is easy to see that  $b_{l+1} \in A'$ .

///

### 3. Construction of the evaluator

As mentioned in section 4.1 we will construct a history graph in order to simulate all possible events. We have to consider:

- how can we determine the state of a son of a node
- how do we keep track of the pairs  $(s, I)$  for which we not already have made a PLAN and a GOTO table entry.

Let us discuss how to determine the state in which a son  $X$  of a node can be. If we know the fixed part of the state for the son (i.e. the  $p^+$ -part) and the input set  $I$  then we can determine the state of the node  $X$  as  $s = (p^+, A)$  where  $I$  makes  $s$  complete (according to lemma 1).

For each composite visiting sequence for a node (see section 4.1) we can determine the input set for the last visit to the son if any. On the basis of the state for the father we can determine a part of a state (the SYM-graph for the left hand side of a production) for the son. The set of pairs  $(s, I)$  for which a PLAN and GOTO table entry is required by a specific

visiting sequence can be determined by the following algorithm.

ALGORITHM 3:

Input: A (finite) history graph  $H$ , a visiting sequence  $vs$  for  
PLAN( $s, I$ ) where  $s$  is the label of a node in  $H$

Output: REQUIRED: a queue of pairs ( $s', I'$ ) for which a PLAN  
and a GOTO table entry is required by  $vs$

Method:

1. Let REQUIRED be empty;  
Let  $s = (p+, A)$  and  $p: X ::= X_1 X_2 \dots X_n$ ;  
Let  $vs = (X_{i1}, I_1)(X_{i2}, I_2) \dots (X_{im}, I_m)$ ;  
FOR each path in  $H$  beginning at an initial state and ending at  $s$   
DO construct the composite visiting sequence  $cvs$  for the path;  
FOR  $j := 1$  TO  $m$  DO perform step 2;
2. Let  $Y = X_{ij}$  and Let  $Q$  be the SYM-graph for  $X_{ij}$  determined from  $p+$ ;  
a: IF  $j > 1$  THEN append  $(X_{i(j-1)}, I_{(j-1)})$  to  $cvs$ ;  
b: determine the pair  $(Y, I')$  in  $cvs$  such that  
-  $cvs = cvs' (Y, I') cvs''$   
- for all  $(Y', I'')$  in  $cvs''$  we have  $Y' \neq Y$ ;  
IF  $(Y, I')$  does not exist THEN go to step 2d;  
c: FOR each  $q+$  where  $q: Y ::= Y_1 Y_2 \dots Y_h$ ,  
 $q+ = (q, Q, Q_1, Q_2, \dots, Q_h)$  and where  
 $Q$  may be derived from  $D(q) \{Q_1 Q_2 \dots Q_h\}$  and  
 $Q_L \in SYM(Y_L)$  for  $1 \leq L \leq h$   
DO let  $s'' = (q+, A')$  be the state made complete by  $I'$   
and append  $(s'', I_j)$  to REQUIRED;  
stop performance of step 2;  
d: FOR each  $q+$  where  $q: Y ::= Y_1 Y_2 \dots Y_h$ ,  
 $q+ = (q, Q, Q_1, Q_2, \dots, Q_h)$  and where  
 $Q$  may be derived from  $D(q) \{Q_1 Q_2 \dots Q_h\}$  and  
 $Q_L \in SYM(Y_L)$  for  $1 \leq L \leq h$   
DO append  $((q+, A'), I_j)$  to REQUIRED  
where  $(q+, A')$  is an initial state
3. Stop

///

The algorithm will stop since the history graph is a directed acyclic graph and since the number of productions and SYM-graphs are finite.

In order to keep track of the pairs  $(s, I)$  for which we have to construct PLAN and GOTO table entries we introduce a queue called REMEMBER. Initially REMEMBER will consist of the pairs  $(s, \emptyset)$  where  $s$  is a possible initial state for the root of a derivation tree. A pair is removed from the queue when it is possible to make a PLAN and a GOTO table entry for it. Whenever a visiting sequence is made, the possible states for the sons are found and pairs  $(s, I)$  are added to REMEMBER. When REMEMBER is empty we have a complete set of PLAN and GOTO table entries.

ALGORITHM 4:

Input: An AG  $G = (V, B, R, Z)$

Output: An evaluator  $\underline{E}(G)$  for  $G$

Method:

1. Let  $H$  be the initial history graph;  
 Let REMEMBER consist of the pairs  $(s, \emptyset)$  where  
 $s = (p+, A)$ ,  $p$  has left hand side  $Z$  and  $s$  is an initial state;
2. IF REMEMBER is not empty  
 THEN let  $(s, I)$  be the front element of REMEMBER  
 and remove  $(s, I)$  from REMEMBER  
 ELSE go to step 4
3. IF a PLAN and GOTO table entry for  $(s, I)$  not already  
 have been constructed  
 THEN
  - a: apply algorithm 2 to  $(s, I)$  to yield  
 $PLAN(s, I) = (I'', vs, S'')$  and  $GOTO(s, I) = s'$ ;
  - b: apply algorithm 3 to  $H$  and  $vs$  to construct the queue  
 REQUIRED and append REMEMBER to REQUIRED;
  - c: add a node labelled  $s'$  to  $H$  if it is not present  
 and add an arc  $s \rightarrow s'$  labelled with  $(I, vs)$ ;
4. go to step 2;
5. let  $\underline{S}$  consist of the states constructed above, and  
 let  $\underline{I}$  consist of the input sets constructed above.  
 Then  $\underline{E}(G) = (\underline{S}, \emptyset, \underline{I}, PLAN, GOTO)$

///

The algorithm will stop since there are a finite number of possible states and input sets. The following lemmas will be used to show that any

well-defined AG is  $k$ -visit for some  $k$ .

LEMMA 2:

If in algorithm 4  $(s, I)$  is in REMEMBER at some time during the performance then  $PLAN(s, I)$  and  $GOTO(s, I)$  are constructed.

PROOF: The lemma follows easily from algorithm 4.

///

LEMMA 3:

If algorithm 4 calls algorithm 2 to make  $PLAN$  and  $GOTO$  table entries for  $(s, I)$  then there will be a node in the history graph labelled with  $s$ .

PROOF: Assume that there is no node labelled  $s$  in the history graph when algorithm 2 is applied to  $(s, I)$ . If  $s$  is an initial state we obviously have a contradiction. Therefore let  $s$  be made complete by a set  $I'$  (lemma 1). The pair  $(X, I')$  will be in a visiting sequence and  $GOTO(s', I') = s$  (for some state  $s'$ ) has been determined because REMEMBER is a queue. But then  $s'$  will not be in the history graph. Since all states originate in initial states we have a contradiction.

///

LEMMA 4:

Let  $G$  be an AG and let  $\underline{E}(G)$  be an evaluator constructed by algorithm 4. Let  $t \in \text{DOM}(Z)$  and apply  $\underline{E}(G)$  to  $t$ . If there is a call  $VISIT(X, I)$  to a node  $X$  in  $t$  in the state  $s$  then at some time during the performance of algorithm 4  $(s, I)$  is in REMEMBER.

PROOF: We prove the lemma by induction in the length of a path from the root of  $t$  to the node  $X$ .

If the length is zero then we have a call  $VISIT(Z, \emptyset)$  and  $s$  is an initial state. Obviously  $(s, \emptyset)$  is in REMEMBER at the start.

For the induction step assume that the father of  $X$  is  $Y$  and that a call  $VISIT(Y, I')$  implies the call  $VISIT(X, I)$  where  $X$  is in the state  $s$ . Let the state of  $Y$  before the call be  $s'$ . The induction hypothesis gives that  $(s', I')$  is in REMEMBER at some point of time. From lemma 2 it follows that  $PLAN(s', I')$  and  $GOTO(s', I')$  are constructed. Since the visit to  $Y$  implies the visit to  $X$  we have that the visiting sequence of  $PLAN(s', I')$  contains the pair  $(X, I)$ .

From lemma 3 it follows that there is a node in the history graph

labelled  $s'$ . If the node  $X$  is in state  $s$  and  $s$  is made complete by  $J$  then it is easy to show that  $(X, J)$  will be in the composite visiting sequence of algorithm 3 and that  $(s, I)$  will be in REQUIRED. If  $s$  is an initial state then it is easy to show that there will not be any pairs  $(X, I')$  in the composite visiting sequence.

///

LEMMA 5:

Let  $G$  be an AG and let  $\underline{E}(G)$  be the evaluator constructed by algorithm 4. If  $\underline{E}(G)$  is applied to a tree  $t \in \text{DOM}(Z)$  and there is a call  $\text{VISIT}(X, I)$  to a node  $X$  in  $t$  in a state  $s$  then there exist entries  $\text{GOTO}(s, I)$  and  $\text{PLAN}(s, I)$  in  $\underline{E}(G)$ .

PROOF: In order to show the lemma we have to consider two points:

- if there is a call  $\text{VISIT}(X, I)$  to a node  $X$  in  $t$  and  $X$  is in the state  $s$  then at some time during the performance of algorithm 4  $(s, I)$  is in REMEMBER
- if  $(s, I)$  is in REMEMBER at some time during the performance of algorithm 4 then  $\text{PLAN}(s, I)$  and  $\text{GOTO}(s, I)$  are constructed.

The first part follows from lemma 4 and the second part from lemma 2.

///

LEMMA 6:

Let  $G$  be an AG and let  $\underline{E}(G)$  be the evaluator constructed by algorithm 4. Apply  $\underline{E}(G)$  to a tree  $t \in \text{DOM}(Z)$ . Let a node  $X$  in  $t$  initially be in the state  $s_0$  and let a sequence of visits change the state to  $s_1, s_2, \dots,$  and  $s_k$ . Then there are nodes  $s_j$  for  $0 \leq j \leq k$  in the final history graph of algorithm 4 and there are arcs  $s_j \rightarrow s_{j+1}$  for  $0 \leq j \leq k-1$ .

PROOF: If the state of a node is changed from  $s_j$  to  $s_{j+1}$  by a call  $\text{VISIT}(X, I)$  then  $\text{GOTO}(s_j, I) = s_{j+1}$ . From lemma 3 it follows that there will be an arc  $s_j \rightarrow s_{j+1}$  in the history graph.

///

THEOREM 2:

Let  $G$  be a well-defined AG and let  $\underline{E}(G)$  be an evaluator for  $G$  constructed by algorithm 4. Then  $\underline{E}(G)$  will specify a  $k$ -visit evaluation strategy when applied to a tree  $t \in \text{DOM}(Z)$  for some fixed  $k$  independent of  $t$ .

PROOF: Let  $t \in \text{DOM}(Z)$ . In order to show that  $\underline{E}(G)$  specifies a  $k$ -visit

evaluation strategy when applied to  $t$  we have to show:

- if there is a call  $VISIT(X,I)$  to a node  $X$  in  $t$  and  $X$  is in the state  $s$  then there exist entries  $PLAN(s,I)$  and  $GOTO(s,I)$
- each node is visited at most  $k$  times.

The first part follows directly from Lemma 5. Let  $k$  be the maximal length of a path in the history graph constructed by algorithm 4. Each visit to a node will change its state and from Lemma 6 it follows that a state can at most be changed  $k$  times. Thus the evaluator will specify a  $k$ -visit evaluation strategy.

///

THEOREM 3:

Every well-defined AG  $G$  is  $k$ -visit for some  $k$ .

PROOF: Let  $\underline{E}(G)$  be the evaluator for  $G$  constructed by algorithm 4. Apply  $\underline{E}(G)$  to a tree  $t \in \text{DOM}(Z)$ . In order for  $G$  to be  $k$ -visit we have to show:

- $\underline{E}(G)$  specifies a  $k$ -visit evaluation strategy for  $t$
- the synthesized attributes of the root of  $t$  are evaluated.

The first part follows from theorem 2. There will be a call  $VISIT(Z,\emptyset)$  to the root of  $t$  and this visit will leave the root of  $t$  in a state made complete by  $\emptyset$  (follows from Lemma 1) and thereby all the attributes of  $Z$  have been evaluated. And the second part of the proof is completed.

///

THEOREM 4:

Every well-defined AG  $G$  is traditional  $k$ -visit for some  $k$ .

PROOF (outline): Let  $G$  be a well-defined AG.  $G$  is augmented in that every nonterminal symbol is supplied with an extra synthesized attribute. This attribute will be a dummy attribute in the sense that it will never be computed when applying the evaluator to a derivation tree. It is only used during the construction of the evaluator in order to enforce that all the attributes in every derivation tree is evaluated.

The dummy attributes are used in the following way. If  $p: X ::= X_1 X_2 \dots X_n$  is a production then the dummy attribute of  $X$  will depend on all the attributes in  $I(X)$ ,  $S(X_1)$ ,  $S(X_2)$ , ..., and  $S(X_n)$  including the dummy attribute of  $X_j$  for  $1 \leq j \leq n$ .

If  $G'$  is the AG constructed from  $G$  in this way then it may be shown that  $G'$  will be translational  $k$ -visit if and only if  $G$  is traditional



k-visit. Thereby theorem 3 easily gives the theorem.

///

As we have seen algorithm 4 determines (indirectly) a  $k$  such that an AG  $G$  is  $k$ -visit. Furthermore different choices in the nondeterministic step in algorithm 2 may result in different  $k$ 's. The nondeterminism can be used to give the resulting evaluator special properties. One can use the freedom to force the left-most son with evaluable attributes to be visited each time. Another possibility is to force the sons to be visited in order from left to right.

Still another possibility will be to use the freedom to make  $k$  minimal. It is possible nondeterministically to find the minimal  $k$  such that an AG is  $k$ -visit. I have not been able to find a deterministic algorithm doing that. In the rest of this section we will consider some simple methods that may be used to find an upper and a lower bound between which the minimal  $k$  will be.

An upper bound for  $k$  will be the maximal number of attributes for any symbol in  $G$ . The reason for this is that we only makes a visit to a son when some new attributes can be evaluated (see algorithm 2). However it is easy to change the algorithm such that the maximal number of synthesized attributes for any symbol of  $G$  becomes an upper bound (we then only visit a son when some new synthesized attributes can be evaluated).

It is also possible to determine a lower bound for the  $k$ . We then use the SYM- and CON-graphs for the symbols. Let  $Q' \in \text{SYM}(X)$  and  $Q'' \in \text{CON}(X)$  for a symbol  $X$  of  $G$ . Consider a tree  $t$  with a node  $X$  with  $Q'$  and  $Q''$  as resp. the corresponding SYM- and CON-graphs. At the first visit to  $X$  we can at most evaluate the attributes in the set  $I1(X,Q) \cup S1(X,Q)$  where  $Q = Q' \sqcup Q''$  and

$$I1(X,Q) = \{b \in I(X) \mid \text{there are no arcs in } Q \text{ ending at } b\}$$

$$S1(X,Q) = \{b \in S(X) \mid \text{if } c \rightarrow b \text{ is in } Q \text{ then } c \in I1(X,Q)\}$$

After the  $m$ 'th visit to the node  $X$  ( $m > 1$ ) we will at most have evaluated the attributes in  $I_m(X,Q) \cup S_m(X,Q)$  where

$$I_m(X,Q) = \{b \in I(X) \mid \text{if } c \rightarrow b \text{ is in } Q \text{ then } c \in S_{(m-1)}(X,Q)\}$$

$$S_m(X,Q) = \{b \in S(X) \mid \text{if } c \rightarrow b \text{ is in } Q \text{ then } c \in I_m(X,Q)\}$$

The minimal number of visits to the node  $X$  which is required in order to evaluate all the attributes of  $X$  is therefore the least  $m$  such that  $I_m(X,Q) \cup S_m(X,Q) = A(X)$ . This value can be determined from the graph  $Q$ .

Let  $l$  be the length of the longest path in  $Q$  ( $l$  is finite since  $G$  is assumed to be well-defined). The minimal value  $m$  can be determined by the following formulas:

- if  $l = 0$  then  $m = 1$
- if the longest path begins at an inherited attribute then
 
$$m = l \text{ DIV } 2 + l \text{ MOD } 2$$
- if the longest path begins at a synthesized attribute then
 
$$m = (l+1) \text{ DIV } 2 + (l+1) \text{ MOD } 2.$$

For all symbols  $X$  and all graphs  $Q = Q' \cup Q''$  where  $Q' \in \text{SYM}(X)$  and  $Q'' \in \text{CON}(X)$  we can determine the minimal number of visits required in order to evaluate all the attributes of  $X$ . The maximal of these numbers will be the minimal value for  $k$  for which the AG may be  $k$ -visit.

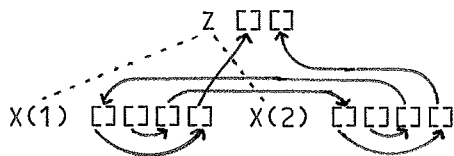
The following example shows that there are AGs which are  $k$ -visit but where  $k$  cannot be equal to the lower bound determined above.

EXAMPLE 5:

Let  $G$  be the AG in example 1. We have these CON-graphs:

CON(Z):     $\square\square$             CON(X):     $\square\square\square\square$

The lower bound for which  $G$  is  $k$ -visit is 1 whereas the upper bound is 2. Inspection of the graph  $D(p_1) \cap Q$  where  $Q \in \text{SYM}(X)$  shows that  $G$  is 2-visit but not 1-visit:



In order for  $G$  to be 1-visit we shall evaluate all the attributes of  $X(1)$  at the first visit and all the attributes of  $X(2)$  at the first visit. It is easy to see that this is not possible.

///

The evaluator constructed in this section is an extension of the one in [Kew76]. In that paper there are put some restrictions on the dependencies that are allowed between the attributes. These restrictions imply that the states of an evaluator can be chosen simpler than here: a state is a pair  $(p, A)$  where  $p$  is a production and  $A$  is a subset of the attributes of the symbols in the production  $p$ .

In [Kas78] another subclass of AGs are considered. Here the states can

be further simplified: a state has the form  $(p,m)$  where  $p$  is a production and  $m$  is an integer denoting the number of visits that has been performed at the node. The restrictions on the AGs ensure that it is possible to determine the set of known attributes (i.e.  $A$ ) on the basis of  $p$  and  $m$ . Also the input sets of the evaluator can be simplified: an integer tells the number of the visit that now will be performed.

The evaluator constructed in this section has some resemblance with that of [CoH79] although a slightly different interpretation of a visit is used in that paper.

## 5. K LEFT-TO-RIGHT PASS ATTRIBUTE GRAMMARS

In the previous chapter we extended the one-visit property defined by [EnF79]. A subclass of the one-visit AGs is the L-attribute grammars defined by [Boc76]. An AG in this class can be characterized by that all the attributes of any derivation tree can be evaluated by one left-to-right depth-first pass over the tree. In this chapter we will extend this subclass in a way similarly to that of the one-visit AGs in chapter 4 and thereby we will define a class of AGs called the k left-to-right pass AGs.

In [Boc76] a class of AGs called multipass AGs are introduced. An AG in this class has the property that the attributes associated with a derivation tree can be evaluated during a fixed number (for the AG) of left-to-right depth-first passes over the tree. But the algorithm checking the multipass property rejects some very simple grammars which really have the multipass property. It is required that all nodes in a tree with the same label are treated in the same way i.e. corresponding attributes are evaluated in the same pass. Consider for instance the AG with the following productions (from [ALb79]):

$$\begin{aligned} p1: & \langle Z \uparrow a \rangle ::= \langle X \psi b \uparrow a \rangle \langle X \psi 1 \uparrow b \rangle \\ p2: & \langle X \psi a \uparrow a \rangle ::= A \end{aligned}$$

All the attributes of the only derivation tree for this grammar can be evaluated by two left-to-right passes over the tree. But the AG is rejected by the algorithm given by [Boc76].

In section 5.1 I introduce the concept of a k left-to-right pass AG. It is shown that there are translations that can be defined by one-visit AGs but which cannot be specified by a k left-to-right pass AG for any k when some conditions are satisfied.

In section 5.2 is given an algorithm that for any AG and any k tests whether the AG has the k left-to-right pass property.

In section 5.3 I give algorithms that may be used to construct an evaluator for an AG which evaluates the attributes of a derivation tree by k left-to-right passes over the tree if the AG is k left-to-right pass.

## 5.1 THE K LEFT-TO-RIGHT PASS PROPERTY

We give definitions similarly to those of section 4.2:

### DEFINITION 1:

An evaluation strategy for a tree  $t$  is called  $k$  left-to-right pass if  $t$  is traversed  $k$  times in a left-to-right depth-first order.

///

### DEFINITION 2:

An AG  $G$  is (translational)  $k$  left-to-right pass ( $k$ -LR-pass) if for any derivation tree  $t \in \text{DOM}(Z)$  for a string  $w$  there exists a  $k$  left-to-right pass evaluation strategy computing  $\text{meaning}(w, t)$ .

An AG  $G$  is traditional  $k$  left-to-right pass (traditional  $k$ -LR-pass) if for any derivation tree  $t \in \text{DOM}(Z)$  for a string  $w$  there exists a  $k$  left-to-right pass evaluation strategy computing  $\text{meaning}'(w, t)$ .

///

Clearly a  $k$ -LR-pass AG will also have the  $(k+1)$ -LR-pass property. Furthermore there are translations that can be specified by  $k$ -LR-pass AGs over a semantic domain  $\mathcal{D}$  but which cannot be specified by  $(k-1)$ -LR-pass AGs over the same domain and with the same underlying grammar. An example is the translation  $\text{TAU}(k)$  defined in section 4.2.

Clearly a  $k$ -LR-pass AG will also have the  $k$ -visit property. The opposite is not the case. In fact there is a translation specified by a 1-visit AG  $G$  over a semantic domain  $\mathcal{D}$  which cannot be specified by a  $k$ -LR-pass AG over the same domain and with the same underlying grammar.

Consider the semantic domain  $\mathcal{D} = (\underline{D}, \underline{E})$ .  $\underline{D}$  contains the flat lattice constructed from the set  $\{A, C\}^*$ . In  $\underline{E}$  we have the identity function and the functions making string concatenation with  $A$  and  $C$  (denoted by resp.  $x \sim A$  and  $x \sim C$  where  $x$  is the parameter).

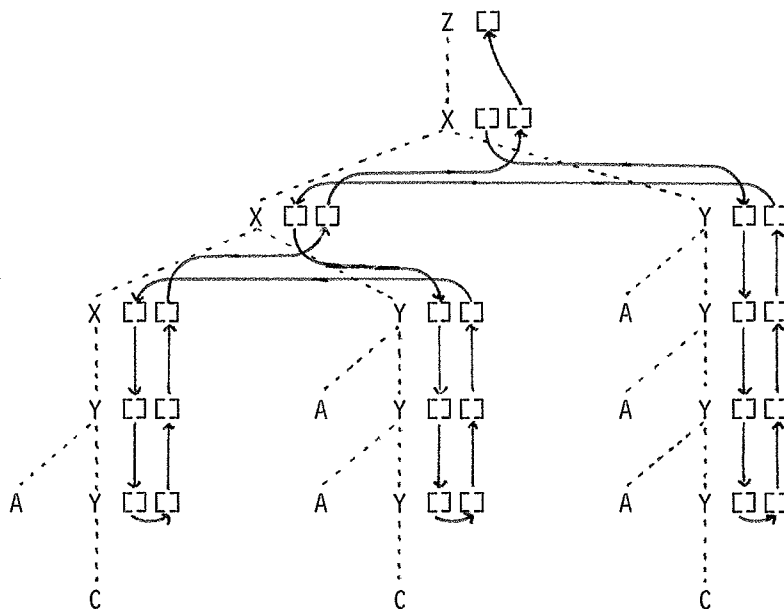
Consider this AG  $G$  over the semantic domain  $\mathcal{D}$  with the productions:

- $p1: \langle Z \uparrow s \rangle ::= \langle X \psi \lambda \uparrow s \rangle$
- $p2: \langle X \psi s1 \uparrow s2 \rangle ::= \langle X \psi s3 \uparrow s2 \rangle \langle Y \psi s1 \uparrow s3 \rangle$
- $p3: \langle X \psi s1 \uparrow s2 \rangle ::= \langle Y \psi s1 \uparrow s2 \rangle$
- $p4: \langle Y \psi s1 \uparrow s2 \rangle ::= A \langle Y \psi s1 \sim A \uparrow s2 \rangle$
- $p5: \langle Y \psi s1 \uparrow s1 \sim C \rangle ::= C$

We have

$$\begin{aligned} \mathcal{I}(G, \mathcal{D}) = \{ (w, w') \mid w = w_1 C w_2 C \dots C w_n C, w' = w_n C w_{(n-1)} C \dots C w_1 C \\ \text{and } w_j \in \{A\}^* \text{ for } 1 \leq j \leq n \} \end{aligned}$$

It is easy to see that  $G$  is well-defined. We can use the analysis at the end of section 4.3 to deduce that  $G$  is 1-visit since each symbol of  $G$  has at most one synthesized attribute. In order to give an intuitive feeling about how the flow of attribute values are we may consider the dependency network for the derivation tree for the string ACAACAAAC:



We will show that the translation  $\mathcal{I}(G, \mathcal{D})$  cannot be specified by any  $k$ -LR-pass AG with the same underlying grammar and over the semantic domain  $\mathcal{D}$ .

Assume the contrary. Let  $G_k$  be a  $k$ -LR-pass AG with the underlying grammar  $G_u$ .  $G_k$  is over the semantic domain  $\mathcal{D}$  and  $\mathcal{I}(G, \mathcal{D}) = \mathcal{I}(G_k, \mathcal{D})$ .

Consider the string  $w = w_1 C w_2 C \dots C w_n C$  where  $n > k$  and the length of  $w_n C$  is greater than the longest constant value appearing in the semantic rules of  $G_k$ . Let  $t$  be the derivation tree of  $w$ . In  $t$  we can find subtrees  $t_j \in \text{DOM}(Y)$  such that the yield of  $t_j$  is  $w_j C$  for  $1 \leq j \leq n$ . Furthermore  $w$  is chosen such that there are infinitely many trees  $t_j' \in \text{DOM}(Y)$  with the property the  $\text{sym}(t_j') = \text{sym}(t_j)$  for  $1 \leq j \leq n$ . (This is possible since there is a finite number of graphs in  $\text{SYM}(Y)$ ).

We can find attributes  $b_1, b_2, \dots, b_n$  of nodes in  $t$  such that

- i)  $b_n \rightarrow *b_{(n-1)} \rightarrow * \dots \rightarrow *b_1 \rightarrow *c$ ,  $c \in S(Z)$ , is a path in  $D(t)$
- ii) the value of  $b_j$  is  $w_n c w_{(n-1)} c \dots c w_j c$  for  $1 \leq j \leq n$
- iii) if  $d \rightarrow b_j$  is in  $D(t)$  then the value of  $d$  is different from the value of  $b_j$  for  $1 \leq j \leq n$

We will show:

In order to evaluate  $b_j$  when  $b_{(j+1)}$  has been evaluated we have to visit some nodes in  $t_j$ .

Let  $b_{(j+1)}$  be evaluated in some pass and assume that  $b_j$  is evaluated in the same pass.

Let us first consider the case where all the nodes of  $t_j$  have been visited in the pass before  $b_{(j+1)}$  and thereby  $b_j$  are evaluated. We will now change the subtree  $t_j$  of  $t$  to a tree  $t'_j \in \text{DOM}(Y)$  with the property that  $\text{sym}(t_j) = \text{sym}(t'_j)$  but where  $t_j$  and  $t'_j$  have different yields. Let  $t'$  be the new derivation tree and let its yield be  $w'$ . There will be attributes  $c_i$  at the nodes of  $t'$  satisfying:

- i)  $c_n \rightarrow *c_{(n-1)} \rightarrow * \dots \rightarrow *c_1 \rightarrow *c$ ,  $c \in S(Z)$ , is a path in  $D(t')$
- ii) the value of  $c_i$  is  $w_n c w_{(n-1)} c \dots c w_i c$  for  $j \leq i \leq n$
- iii) if  $d \rightarrow c_i$  is in  $D(t')$  then the value of  $d$  is different from that of  $c_i$  for  $j \leq i \leq n$

Thus the value of  $c_i$  is equal to that of  $b_i$  for  $j \leq i \leq n$ . But the value of  $c_j$  will be equal to that of  $b_j$  since no nodes in  $t_j$  (or  $t'_j$ ) are visited when constructing  $c_j$  from  $c_{(j+1)}$  and we get the wrong output for  $w'$ .

The case where none of the nodes of  $t_j$  have been visited in the pass before  $b_{(j+1)}$  and  $b_j$  are evaluated can be handled in a almost similar manner. The subtree  $t_j$  is changed and it is shown that the output will not be changed.

The conclusion is that in order to evaluate  $b_j$  from  $b_{(j+1)}$  we have to visit the nodes in  $t_j$ . Since  $b_n$  cannot be evaluated before the first pass at least  $n$  passes over  $t$  is required in order to determine the translation of the string  $w$ . But  $k < n$  and we have a contradiction:  $G_k$  cannot specify the same translation as  $G$ . This shows the theorem:

#### THEOREM 1:

There exists a translation specified by a 1-visit AG over a semantic domain  $\mathcal{D}$  which cannot be specified by any  $k$ -LR-pass AG over the same seman-

tic domain and with the same underlying grammar.

///

A similar result can be obtained for the traditional approach.

## 5.2 TESTING THE K LEFT-TO-RIGHT PASS PROPERTY

Let  $G$  be a well-defined AG. For a given  $k$  will  $G$  be (traditional)  $k$ -LR-pass? This question will be answered in the rest of this section.

Consider a tree  $t \in \text{DOM}(X)$  and let  $\text{sym}(t) = Q$ . The graph  $Q$  tells how the attributes of the root of  $t$  depend on each other in  $t$ , but the graph does not tell how many left-to-right passes over  $t$  there will be necessary in order to evaluate some specific synthesized attribute when some inherited attributes are known. This is however the kind of information that are required when we want to test whether the AG is (traditional)  $k$ -LR-pass for some  $k$ .

One may choose to associate weights with the arcs of the SYM-graphs. This will be done in a way that gives the weighted graphs some desirable properties. If  $b \rightarrow c$  is an arc in the graph  $Q$  above and if the arc has associated e.g. the weight three then at least three left-to-right passes over the tree will be required in order to evaluate the attribute  $c$  when  $b$  becomes known. More than three passes may be necessary if  $c$  depends on other attributes than  $b$  in  $Q$ .

### DEFINITION 3:

Let  $G$  be an AG. A weighted SYM-graph for a symbol  $X$  of  $G$  is a graph  $Q \in \text{SYM}(X)$  with non-negative weights associated with the arcs.

///

There may be synthesized attributes of the root of the tree that do not depend on any attribute in  $Q$ . They cause a little problem since they may be evaluated after any number of passes. In order to keep track of these attributes we extend the AG. To each nonterminal symbol is associated an extra (dummy) inherited attribute and all synthesized attributes of the symbol will depend on this attribute. This is ensured in this way:

If  $p: X ::= X_1 X_2 \dots X_n$  is a production in  $G$  then all inherited attributes of  $X_j$  (including the dummy inherited attribute of  $X_j$  if  $X_j \in V_n$ ) will depend on the dummy inherited attribute of  $X$  for  $1 \leq j \leq n$ . Furthermore



any synthesized attribute of X will depend on the dummy inherited attribute of X.

We will in the rest of this chapter assume that the AGs are extended in this way.

Let  $Q_j'$  be a weighted graph corresponding to the SYM-graph  $Q_j$  of  $X_j$  for  $1 \leq j \leq n$ . Let us see how to construct a weighted graph  $Q'$  corresponding to the SYM-graph  $Q$  of  $X$  where  $Q$  is derived from the graph  $D(p) \bowtie Q_1 Q_2 \dots Q_n$ .

The arc  $b \rightarrow c$  is in  $Q$  if there is a path  $b \rightarrow^* c$  in  $D(p) \bowtie Q_1 Q_2 \dots Q_n$ . Let  $b = b_0 \rightarrow b_1 \rightarrow b_2 \rightarrow \dots \rightarrow b_m = c$  ( $m > 0$ ).

Consider an arc  $b_{(i-1)} \rightarrow b_i$  in one of the graphs  $Q_j$  where  $1 \leq j \leq n$ . If the weight of this arc in  $Q_j'$  is  $v_i$  then we must make a pass over the tree at least  $v_i$  times in order to evaluate  $b_i$  when  $b_{(i-1)}$  becomes known.

If  $b_{(i-2)}$  is an attribute of  $X_h$  and  $h < j$  then  $b_{(i-1)}$  and  $b_{(i-2)}$  can be evaluated in the same pass and thus the number of extra passes required in order to evaluate  $b_i$  will be at least  $v_i - 1$ .

On the other hand if  $b_{(i-2)}$  is an attribute of  $X_h$  for  $h \geq j$  then we cannot evaluate  $b_{(i-2)}$  and  $b_{(i-1)}$  at the same pass and thus at least  $v_i$  extra passes are required in order to evaluate  $b_i$ .

Let us define some weights  $u_i$  for the arcs  $b_{(i-1)} \rightarrow b_i$  in the path:

- if  $i = 1$  then  $u_i = 1$
- if  $i = m$  then  $u_i = 0$
- if  $b_{(i-1)} \in I(X_j)$  and  $b_i \in S(X_j)$  then  $u_i = v_i - 1$
- if  $b_{(i-1)} \in S(X_j)$ ,  $b_i \in I(X_h)$  and  $j < h$  then  $u_i = 0$
- if  $b_{(i-1)} \in S(X_j)$ ,  $b_i \in I(X_h)$  and  $j \geq h$  then  $u_i = 1$

Thus the cost of going from right to left along an arc in  $D(p)$  is one corresponding to that an extra pass is required. Going from left to right along an arc in  $D(p)$  is free since this can be done in the same pass. If  $v_i$  passes is required in order to evaluate  $b_i$  from  $b_{(i-1)}$  then  $u_i$  is the number of passes required more than the one that currently is performed.

The total number of passes required in order to evaluate the attributes  $b_i$  for  $0 \leq i \leq m$  is  $\sum_{i=1}^m u_i$ .

If  $m=1$ , i.e.  $b \rightarrow c$  is in  $D(p)$ , then the weight of the path is one.

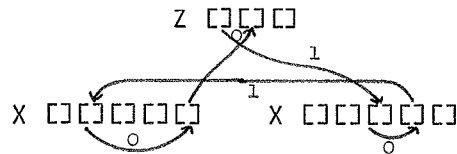
Thus the weight of the arc  $b \rightarrow c$  in  $Q$  will be the maximal of the weights for the paths  $b \rightarrow^* c$  in  $D(p) \bowtie Q_1 Q_2 \dots Q_n$ . We say that  $Q'$  is derived from the graph  $D(p) \bowtie Q_1' Q_2' \dots Q_n'$ .

EXAMPLE 1:

Consider the AG of example 1 in chapter 4. We have



And thereby we have a composite graph for p1 with (among others) this path; the labels are the  $u_i$ -weights defined above:



And we get this graph with weights for Z:



///

As we associate SYM-graphs with the nodes of a derivation tree we will associate weighted SYM-graphs with the nodes. Let  $lr\text{-sym}(t)$  denote the weighted SYM-graph associated with the root of the tree  $t$ . The prefix 'lr' refers to that the weights are computed for left-to-right passes over the tree.

We define for  $t \in \text{DOM}(X)$ :

- if  $X \in V_t$  then  $lr\text{-sym}(t) = \text{sym}(t)$
- Let  $X \in V_n$  and let  $p: X ::= X_1 X_2 \dots X_n$  ( $n > 0$ ) be the production applied at the root of  $t$ , i.e.  $t = X[t_1 t_2 \dots t_n]$  for trees  $t_j \in \text{DOM}(X_j)$  for  $1 \leq j \leq n$ . Then  $lr\text{-sym}(t)$  is the weighted graph derived from the graph  $D(p)[lr\text{-sym}(t_1) lr\text{-sym}(t_2) \dots lr\text{-sym}(t_n)]$  as described above
- if  $X \in V_n$  and  $p: X ::= \lambda$  is the production applied at the root of  $t$  then  $lr\text{-sym}(t)$  is the graph  $\text{sym}(t)$  with the weight one associated with each arc.

LEMMA 1:

Let  $t \in \text{DOM}(X)$  and let  $Q = lr\text{-sym}(t)$ . If  $d$  is the dummy inherited attribute of  $X$  then the weight of the arc  $d \rightarrow b$  in  $Q$  will be greater than or equal to the weight of any arc  $c \rightarrow b$  in  $Q$ .

PROOF: The lemma will be shown by induction in the height of the tree  $t$ .

If the height is one obviously the lemma hold.

For the induction step assume that the weight of  $c \rightarrow b$  is greater than the weight of  $d \rightarrow b$  in  $Q$ . The weight of  $c \rightarrow b$  will be derived from a path  $c = c_0 \rightarrow c_1 \rightarrow \dots \rightarrow c_m = b$  in  $D(p) \mathbb{E} Q_1 Q_2 \dots Q_n \mathbb{I}$  where  $p: X ::= X_1 X_2 \dots X_n$  is the production applied at the root of  $t$ ,  $t = X[t_1 t_2 \dots t_n]$  and  $lr\text{-sym}(t_j) = Q_j$  for  $1 \leq j \leq n$ . This path has the same weight as the arc  $c \rightarrow b$  in  $Q$ . Since  $c_1$  is an inherited attribute of a symbol  $X_j$  (if  $m > 1$ ) then  $d \rightarrow c_1$  will be in  $D(p)$  and the path  $d \rightarrow c_1 \rightarrow c_2 \rightarrow \dots \rightarrow c_m$  will have the same weight as the path  $c \rightarrow c_1 \rightarrow c_2 \rightarrow \dots \rightarrow c_m$  (the induction hypothesis) and we have a contradiction. If  $m = 1$  the lemma obviously hold.

///

LEMMA 2:

Let  $t \in \text{DOM}(X)$  and let  $Q = lr\text{-sym}(t)$ . Let  $c \rightarrow b$  have the weight  $v$  in  $Q$ . If  $k$  passes are used to evaluate the attribute  $b$  then the value of  $c$  need not be evaluated before the  $(k-v+1)$ 'th pass over  $t$ .

PROOF: We use induction in the height of  $t$ .

If the height is one the lemma obviously holds since any arc in  $Q$  will have the weight one.

For the induction step let  $t = X[t_1 t_2 \dots t_n]$  and let  $p: X ::= X_1 X_2 \dots X_n$  be the production applied at the root of  $t$ , i.e.  $t_j \in \text{DOM}(X_j)$  for  $1 \leq j \leq n$ . Let  $lr\text{-sym}(t_j) = Q_j$  for  $1 \leq j \leq n$ .

Let the weight of the arc  $c \rightarrow b$  in  $Q$  be equal to the weight associated with the path  $c = b_0 \rightarrow b_1 \rightarrow \dots \rightarrow b_m = b$  in  $D(p) \mathbb{E} Q_1 Q_2 \dots Q_n \mathbb{I}$ .

Let the weights  $u_i$  of the arcs  $b_{(i-1)} \rightarrow b_i$  be determined as before:

- if  $i = 1$  then  $u_i = 1$
- if  $i = m$  then  $u_i = 0$
- if  $b_{(i-1)} \in I(X_j)$  and  $b_i \in S(X_j)$  then  $u_i = v_{i-1}$
- if  $b_{(i-1)} \in S(X_j)$ ,  $b_i \in I(X_h)$  and  $j < h$  then  $u_i = 0$
- if  $b_{(i-1)} \in S(X_j)$ ,  $b_i \in I(X_h)$  and  $j \geq h$  then  $u_i = 1$

By induction in ' $i$ ' ( $i > 0$ ) we will show:

If  $b$  is evaluated in the  $k$ 'th pass then  $b_i$  must be evaluated in the  $(k - \sum_{l=i+1}^m u_l)$ 'th pass over  $t$ .

If  $i = m$  then obviously the claim holds.

Assume that the claim holds for  $i+1$ . We will show that it also holds for  $i$ .

Let first the arc  $b_i \rightarrow b_{(i+1)}$  be in  $Q_j$  for some  $j$  and assume that  $b_{(i+1)}$  are evaluated in the  $r$ 'th pass over the tree  $t_j$ . The induction

hypothesis (the lemma) gives that  $b_i$  need not be evaluated before the  $(r - v(i+1) + 1)$ 'th pass over  $t_j$ . From the claim it follows that  $r = k - \sum_{l=i+2}^m u_l$  and thus that  $b_i$  need not be evaluated before the  $(k - \sum_{l=i+1}^m u_l)$ 'th pass. And the claim holds for  $i$ .

Let now  $b_i \rightarrow b(i+1)$  be an arc in  $D(p)$  and let  $b_i \in S(X_j)$  and  $b(i+1) \in I(X_h)$  for some  $j$  and  $h$ .

If  $j < h$  then  $b_i$  need not be evaluated in a pass before  $b(i+1)$  and since  $u(i+1) = 0$  we see that the claim holds for  $i$ .

On the other hand if  $j \geq h$  then  $b_i$  has to be evaluated in a pass over  $t_j$  before the one in which  $b(i+1)$  are evaluated. Obviously the claim also hold here since  $u(i+1) = 1$ .

If  $b(i+1) = b$  then it is easy to see that the claim holds.

This proves the claim. It is now easy to see that the value of  $b_0$  needs not be used before the  $(k - \sum_{l=2}^m u_l)$ 'th pass. But  $\sum_{l=2}^m u_l = v-1$  and we have shown the lemma.

///

### LEMMA 3:

Let  $t \in \text{DOM}(X)$  and let  $Q = \text{lr-sym}(t)$ . If the maximal weight of an arc is  $k$  then exactly  $k$  passes over are required in order to evaluate the attributes in  $S(X)$  (all the attributes in  $I(X)$  are known in the first pass).

PROOF: Assume that the attributes of  $S(X)$  are evaluated at the  $m$ 'th pass over  $t$ . Let  $c \rightarrow b$  be in  $Q$  and have weight  $v$ . From lemma 2 we have that  $b$  has to be known in the  $(m-v+1)$ 'th pass over  $t$ . Let for fixed  $c$ :

$$r_c = \min \{ (m-v+1) \mid c \rightarrow b \text{ is an arc in } Q \text{ and has weight } v \}$$

Thus in order to evaluate all attributes of  $X$  that depend on the attribute  $c$  we have to know  $c$  in the  $r_c$ 'th pass.

Let  $r = \min \{ r_c \mid c \in I(X) \}$ . In order to evaluate the attributes in  $S(X)$  we need in fact only  $m-r+1$  passes over the tree.

From the definition of  $r$  it follows that there is an arc  $b \rightarrow c$  in  $Q$  with the weight  $m-r+1$  and that this will be the maximal weight of an arc in  $Q$ . This completes the proof.

///

### THEOREM 2:

Let  $G$  be an AG augmented with dummy inherited attributes. Let  $t \in \text{DOM}(Z)$ . If  $k$  is the maximal weight of an arc in  $\text{lr-sym}(t)$  then there is a  $k$  left-to-right pass evaluation strategy computing the synthesized attributes

at the root.

PROOF: The theorem follows directly from Lemma 3.

///

The following algorithm may be used to test whether an AG is  $k$ -LR-pass for some fixed  $k$ . The weighted SYM-graphs called LR-SYM( $X$ ) for  $X \in V$  are constructed for each symbol  $X$  if the AG is accepted.

ALGORITHM 1:

Input: an AG  $G = (V, B, R, Z)$  and an integer  $k$

Output: the sets LR-SYM( $X$ ) for  $X \in V$  if  $G$  is  $k$ -LR-pass


Method:


1. LR-SYM $_0$ ( $X$ ) =  $\emptyset$  for  $X \in V_n$ ;  
 LR-SYM $_0$ ( $X$ ) =  $\{Q\}$  for  $X \in V_t$  where  $Q$  is the graph with no arcs;  
 let  $i = 0$
2. let LR-SYM( $i+1$ )( $X$ ) = LR-SYM $_i$ ( $X$ ) for all  $X \in V$ ;  
 FOR each production  $p: X ::= X_1 X_2 \dots X_n$  DO  
 FOR each choice of graphs  $Q_1, Q_2, \dots, Q_n$  where  
 $Q_j \in \text{LR-SYM}(X_j)$  for  $1 \leq j \leq n$   
 DO let  $Q \in \text{LR-SYM}(i+1)(X)$  where  $Q$  is derived from the  
 graph  $D(p) \{Q_1 Q_2 \dots Q_n\}$
3. IF there is a graph  $Q \in \text{LR-SYM}(i+1)(Z)$  with a weight greater than  $k$   
 THEN stop,  $G$  is not  $k$ -LR-pass
4. IF there is an  $X \in V$  such that LR-SYM $_i$ ( $X$ )  $\neq$  LR-SYM( $i+1$ )( $X$ )  
 THEN let  $i := i+1$  and go to step 2;
5.  $G$  is  $k$ -LR-pass;  
 let LR-SYM( $X$ ) = LR-SYM $_i$ ( $X$ ) for all  $X \in V$  and stop.

///

EXAMPLE 2:

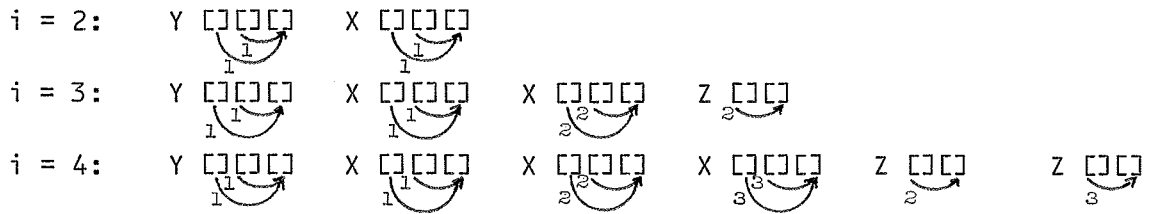
For the AG in example 1 of chapter 4 we have for  $k = 2$ :

LR-SYM( $Z$ ):  $Q_1: Z$  

LR-SYM( $X$ ):  $Q_2: X$  

For the AG considered in section 5.1 we have these LR-SYM $_i$  graphs for  $k = 2$ :

$i = 1$ :  $Y$  



and the algorithm will stop, the AG is not 2-LR-pass.

///

LEMMA 4:

Apply algorithm 1 to an AG  $G$  and an integer  $k$ . If  $Q \in LR\text{-SYM}_i(X)$  for some  $i$  and some  $X$  then there is a tree  $t \in \text{DOM}(X)$  where  $lr\text{-sym}(t) = Q$ .

PROOF: The lemma can easily be shown by induction in ' $i$ '. We will omit the proof.

///

THEOREM 3:

Let  $G$  be an AG augmented with dummy inherited attributes and apply algorithm 1 to  $G$  and an integer  $k$ . Then  $G$  is  $k$ -LR-pass if and only if  $G$  is accepted by the algorithm.

PROOF: Assume that  $G$  is not  $k$ -LR-pass. Then there exist a tree  $t \in \text{DOM}(Z)$  where an attribute  $b \in S(Z)$  requires more than  $k$  passes in order to be evaluated. Then the arc  $d \rightarrow b$  in  $lr\text{-sym}(t)$  has a weight greater than  $k$  where  $d$  is the dummy inherited attribute of  $X$  (lemma 2). Obviously  $G$  is rejected by algorithm 1.

Assume that  $G$  is  $k$ -LR-pass but is rejected by algorithm 1. Then there exist a graph  $Q \in LR\text{-SYM}_i(Z)$  with an arc  $d \rightarrow b$  with a weight greater than  $k$  for some  $i$ . Lemma 3 gives that there is a tree  $t \in \text{DOM}(Z)$  such that  $lr\text{-sym}(t) = Q$ . From theorem 2 it follows that  $G$  cannot be  $k$ -LR-pass - a contradiction.

///

In order to test whether an AG is traditional  $k$ -LR-pass one may extend each nonterminal symbol with a dummy synthesized attribute as described in the proof for theorem 4 in chapter 4. We will not consider that further.

### 5.3 AN EVALUATOR FOR A K LEFT-TO-RIGHT PASS AG

In this section I will give algorithms that construct an evaluator for a k-LR-pass AG. The evaluator will specify a k left-to-right pass evaluation strategy when applied to a derivation tree. The construction of the algorithms are divided into four parts:

1. choice of evaluation states
2. making an entry in the PLAN and GOTO tables
3. simulation of a single pass
4. construction of the evaluator

#### 1. Choice of evaluation states

The analysis of section 4.3 shows that if we have the following information in a state then it is possible to construct a plan in a k-visit evaluator on the basis of the state and an input set.

- the production applied at the node p:  $X ::= X_1 X_2 \dots X_n$
- the SYM-graphs for the nodes  $X, X_1, X_2, \dots, X_n$
- the set A of the attributes of the symbols in p which are known.

The SYM-graphs are used to find those synthesized attributes of the symbol X that can be evaluated when some inherited attributes of X are known. In the k-LR-pass case we will instead use the weighted SYM-graphs for the nodes  $X, X_1, X_2, \dots, X_n$ . That is we have:

#### DEFINITION 4:

An evaluation state is a tuple  $(p+, A)$  where  $p: X ::= X_1 X_2 \dots X_n$  is a production,  $p+ = (p, Q, Q_1, Q_2, \dots, Q_n)$  where Q is a weighted SYM-graph for X and  $Q_j$  is a weighted SYM-graph for  $X_j$  for  $1 \leq j \leq n$ . A is a subset of the attributes of the symbols in p.

///

#### DEFINITION 5:

An initial state is an evaluation state  $(p+, A)$  where  $p: X ::= X_1 X_2 \dots X_n$ ,  $p+ = (p, Q, Q_1, Q_2, \dots, Q_n)$  where  $Q \in \text{LR-SYM}(X)$  can be derived from  $D(p)$  and the graphs  $Q_j \in \text{LR-SYM}(X_j)$  for  $1 \leq j \leq n$ .  $A = S(X_{i1}) \cup S(X_{i2}) \cup \dots \cup S(X_{im})$  where

$$\{X_{ij} \mid 1 \leq i \leq n, 1 \leq j \leq m\} = \{X_h \mid X_h \in V_t \text{ and } 1 \leq h \leq n\}$$

///

EXAMPLE 3:

For the AG of example 1 in chapter 4 we have the following initial states:

- s1 = ((p1, Q1, Q2, Q2), 0)
- s2 = ((p2, Q2, Q2), 0)
- s3 = ((p3, Q2, Q2), 0)
- s4 = ((p4, Q2), 0)

where Q1 and Q2 are the weighed graphs constructed in example 2.

///

The initial states of the nodes in a derivation tree can be determined in a manner very similar to that in the k-visit case (e.g. by use of the lr-sym mapping).

DEFINITION 6:

A final state is an evaluation state (p+, A) where  $p: X ::= X_1 X_2 \dots X_n$  and  $A = A(X) \cup A(X_1) \cup A(X_2) \cup \dots \cup A(X_n)$ .

///

2. Making an entry in the PLAN and GOTO tables

A visiting sequence for a production  $p: X ::= X_1 X_2 \dots X_n$  will have a special form in an evaluator specifying a k left-to-right pass evaluation strategy:

$$vs = (X_1, I_1)(X_2, I_2) \dots (X_n, I_n)$$

The sons of a node is visited in order from left to right and each of them exactly one time.

Let the state of a node X be  $s = (p+, A)$  where  $p: X ::= X_1 X_2 \dots X_n$  and  $p+ = (p, Q, Q_1, Q_2, \dots, Q_n)$ . Let I be the input set to a visit to the node X when it is in the state s and let us see how we can construct a plan for the visit. That is we have to determine:

- i) which attributes of I(X) can be evaluated
- ii) with which input sets shall we visit the sons of X
- iii) which attributes of S(X) can be evaluated

As in the k-visit case the attributes in i) can be determined as  $I - (A \cap I(X))$ .

We will change the weights of the arcs of the SYM-graphs after each visit. If the weight of an arc  $b \rightarrow c$  in a graph is v before the visit and



if  $b$  is in the input set for the visit then the weight of the arc will be changed to  $v-1$  since now only  $v-1$  visits are required in order to evaluate  $c$ . However if  $v=0$  we will not change the weight any more. Thus the  $p^+$  part of a state will be changed in the  $k$ -LR-pass case in contradiction to the  $k$ -visit case.

We will redefine the YIELD-s operation of section 4.3 to handle the weighted graphs. Let

$$\text{YIELD-s}(X, Q, I) = \{b \in S(X) \mid \text{if } c \rightarrow b \text{ is in } Q \text{ and has weight } \\ 0 \text{ or } 1 \text{ then } c \in I\}$$

where  $Q$  is a weighted SYM-graph for  $X$ .

Thus if  $Q$  is the weighted SYM-graph obtained from the state then  $\text{YIELD-s}(X, Q, I) - (A \cap S(X))$  will be the attributes in iii).

If  $A_p$  is the set of attributes known after the visit to  $X(j-1)$  then the input set to the visit to  $X_j$  can be determined as

$$\text{YIELD-i}(X_j, D(p), A_p) = \{b \in I(X_j) \mid \text{if } c \rightarrow b \text{ is in } D(p) \text{ then } c \in A_p\}$$

just as in the  $k$ -visit case. The attributes known after the visit will be  $\text{YIELD-s}(X_j, Q_j, I_j)$  where  $I_j$  is the input set and  $Q_j$  is the weighted SYM-graph for  $X_j$  obtained from the state.

#### ALGORITHM 2:

Input: a state  $s = (p^+, A)$  where  $p^+ := X_1 X_2 \dots X_n$ ,  
 $p^+ = (p, Q, Q_1, Q_2, \dots, Q_n)$ , and a set  $I \subseteq I(X)$ .

Output:  $\text{PLAN}(s, I)$  and  $\text{GOTO}(s, I)$

Method:

1.  $A_p := A \cup I$ ;  
 $I' := I - (A \cap I(X))$
2. FOR  $j := 1$  TO  $n$  DO  
 $I_j := \text{YIELD-i}(X_j, D(p), A_p)$ ;  
 $A_p := A_p \cup I_j \cup \text{YIELD-s}(X_j, Q_j, I_j)$ ;  
FOR all arcs  $b \rightarrow c$  in  $Q_j$  where  $b \in I_j$   
DO change the weight  $v$  of  $b \rightarrow c$  to  $v-1$  if  $v > 0$ ;  
Let the resulting graph be  $Q_j'$
3.  $v_s := (X_1, I_1)(X_2, I_2) \dots (X_n, I_n)$ ;  
 $S' := \text{YIELD-s}(X, Q, I) - (A \cap S(X))$ ;  
FOR all arcs  $b \rightarrow c$  in  $Q$  where  $b \in I$   
DO change the weight  $v$  of the arc  $b \rightarrow c$  to  $v-1$  if  $v > 0$ ;  
Let the resulting graph be  $Q'$ ;

$PLAN(s,I) := (I',vs,S')$ ;  
 $GOTO(s,I) := ((p,Q',Q1',Q2',\dots,Qn'), (Ap \cup S'))$

///

EXAMPLE 4:

Consider the AG of example 3 and apply algorithm 2 to the pair  $(s1, \{z\})$  where  $z$  is the dummy inherited attribute of the symbol  $Z$ . We get

$PLAN(s1, \{z\}) = (\{z\}, (X1, \{x1, b1\})(X2, \{x2, a2, b2\}), \{f\})$

where  $x$  is the dummy inherited attribute of  $X$ . The indices refer to resp. the first and the second son of a node labelled  $Z$ . The names of the attributes are introduced in example 1 of chapter 4.

Furthermore we get  $GOTO(s1, \{z\}) = s5$  where

$s5 = ((p1, Q3, Q4, Q5), \{z, f, x1, b1, c1, x2, a2, b2, c2, d2\})$

and



///

3. Simulation of a single left-to-right pass

As in section 4.3 we will consider two problems:

- how can we determine the state of a son of a node
- how can we keep track of the pairs  $(s,I)$  for which a PLAN and GOTO table entry not already have been made.

It is not quite easy to determine the state of a son of a node, since the states of the nodes in the derivation tree seems not to be complete in a sence similarly to that in section 4.3. We have to find a sequence of visits which have been performed at the son.

On the basis of a composite visiting sequence for a node in the history graph we can determine some visiting sequences for each of the sons of a node in a derivation tree in the actual state. For each possible initial state of a son we can simulate the sequence of visits in the composite visiting sequence. The following algorithm will determine the pairs  $(sj, Ij)$  for which a PLAN and a GOTO table entry are required for a specific visiting sequence:

ALGORITHM 3:

Input: A history graph H with a corresponding set of GOTO table entries and a visiting sequence vs for PLAN(s,I) where s is the label of a node in H

Output: REQUIRED: a set of pairs (s',I') for which a PLAN and a GOTO table entry are required

Method:

1. REQUIRED :=  $\emptyset$   
let vs = (X1,I1)(X2,I2)...(Xn,In)  
FOR each path in H beginning at an initial state s0 and ending at the node labelled s  
DO perform step 2.
2. construct the composite visiting sequence cvs for the path;  
a: FOR j:=1 TO n  
DO let cvs(j) contain the pairs (Y,I'') from cvs where Y = Xj and let the ordering of the pairs in cvs(j) be as that in cvs;  
b: FOR j:=1 TO n  
DO FOR each possible state initial state s0' for Xj where the LR-SYM-graph of Xj is Qj (determined from s0)  
DO let cvs(j) = (Xj,I1')(Xj,I2')...(Xj,Ir');  
FOR h:=1 TO r DO sh' := GOTO(s(h-1)',Ih');  
REQUIRED := REQUIRED  $\cup$  {(sr',Ij)};
3. Stop.

///

EXAMPLE 5:

Let us apply algorithm 3 to the initial history graph of the AG in example 4 and the visiting sequence (X1,{x1,b1})(X2,{x2,a2,b2}) of PLAN(s1,{z}). We get

REQUIRED = { (s2,{x,b}), (s3,{x,b}), (s4,{x,b}),  
(s2,{x,a,b}), (s3,{x,a,b}), (s4,{x,a,b}) }

///

In order to keep track of the pairs (s,I) for which a PLAN and a GOTO table entry have to be constructed we use a set REMEMBER. This set is used in the same manner as the queue with the same name in section 4.3.

ALGORITHM 4:

Input: a history graph and a set of PLAN and GOTO table entries corresponding to m passes over a tree.

Output: a history graph and a set of PLAN and GOTO table entries corresponding to m+1 passes over a tree.

Method:

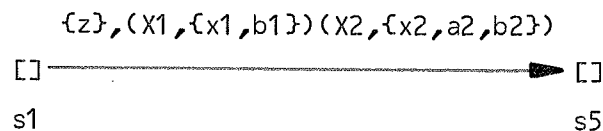
1. REMEMBER :=  $\{(s, \{d\}) \mid s = (p+, A), p \text{ has left hand side } Z, s \text{ is in } H \text{ and } d \text{ is the dummy inherited attribute of } Z\}$
2. IF REMEMBER  $\neq \emptyset$   
THEN choose (s,I) from REMEMBER and remove the pair from REMEMBER  
ELSE go to step 4
3. IF a PLAN and GOTO entry for (s,I) not already have been constructed  
THEN
  - a: apply algorithm 2 to (s,I) to yield  
PLAN(s,I) = (I',vs,S') and GOTO(s,I) = s';
  - b: apply algorithm 3 to H and vs to construct the set REQUIRED and  
let REMEMBER := REMEMBER  $\cup$  REQUIRED
  - c: add an arc s  $\rightarrow$  s' labelled (I,vs) to H
4. let H' be equal to H with the added nodes and arcs.

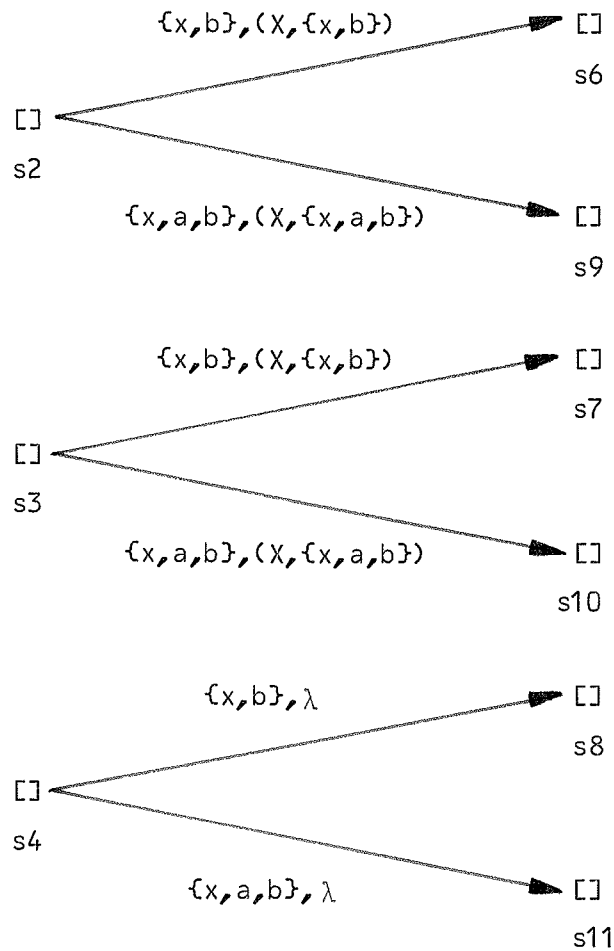
///

The algorithm will stop since there is a finite set of states in H and a finite set of input sets.

EXAMPLE 6:

Let us apply algorithm 4 to the AG of the previous examples. We start by the initial history graph (that is no passes has been simulated). We get the following history graph:





Corresponding to this history graph we have the following entries in the PLAN and GOTO table entries:

<u>S</u> X <u>I</u>	PLAN	GOTO
(s1,{z})	{z}, (X1,{x1,b1})(X2,{x2,a2,b2}), {f}	s5
(s2,{x,b})	{x,b}, (X,{x,b}), {c}	s6
(s3,{x,b})	{x,b}, (X,{x,b}), {c}	s7
(s4,{x,b})	{x,b}, λ , {c}	s8
(s2,{x,a,b})	{x,a,b}, (X,{x,a,b}), {c,d}	s9
(s3,{x,a,b})	{x,a,b}, (X,{x,a,b}), {c,d}	s10
(s4,{x,a,b})	{x,a,b}, λ , {c,d}	s11

The new states are

s5 = ((p1,Q3,Q4,Q5),{z,f,x1,b1,c1,x2,a2,b2,c2,d2})

s6 = ((p2,Q4,Q4),{x0,b0,c0,x,b,c})

s7 = ((p3,Q4,Q4),{x0,b0,c0,x,b,c})

$s_8 = ((p_4, Q_4), \{x, b, c\})$   
 $s_9 = ((p_2, Q_5, Q_5), A(X_0) \cup A(X))$   
 $s_{10} = ((p_3, Q_5, Q_5), A(X_0) \cup A(X))$   
 $s_{11} = ((p_4, Q_5), A(X))$

where the graphs  $Q_3$ ,  $Q_4$  and  $Q_5$  are defined in example 4.

///

#### 4. Construction of the evaluator.

When constructing an evaluator specifying a  $k$  left-to-right pass evaluation strategy we will apply algorithm 4  $k$  times. Each application corresponds to the simulation of yet another pass over a tree.

##### ALGORITHM 5:

Input: an AG  $G = (V, B, R, Z)$  and an integer  $k$

Output: if  $G$  is  $k$ -LR-pass then an evaluator  $\underline{E}(G)$

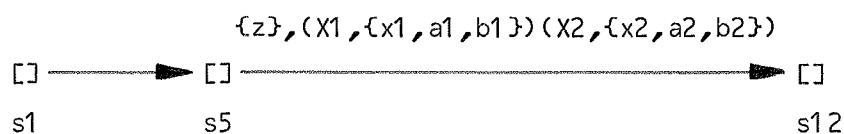
Method:

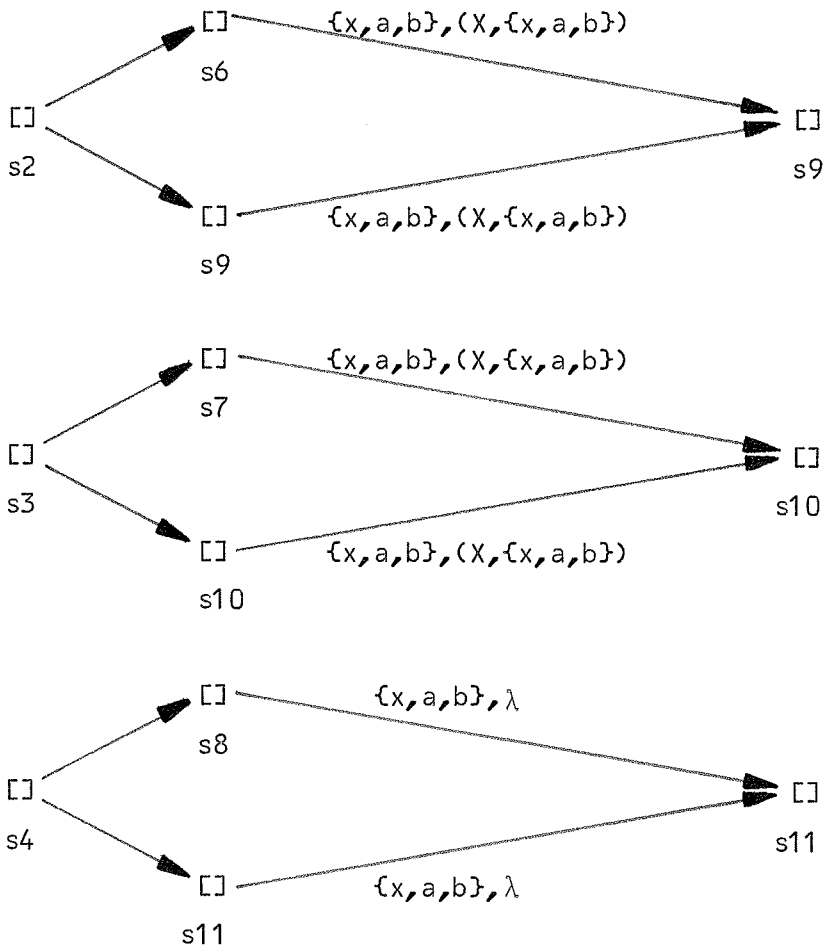
1. Apply algorithm 1 to  $G$  and  $k$ ;  
     IF the algorithm fails THEN stop,  $G$  is not  $k$ -LR-pass.
2. Construct the initial history graph  $H_0$
3. FOR  $m := 1$  TO  $k$  DO  
     apply algorithm 4 to  $H_{(m-1)}$  and  
     let  $H_m$  be the resulting history graph
4. Let  $\underline{S}$  consist of the states of  $H_k$  and  
     let  $\underline{I}$  be the set of input sets labelling the arcs;  
     let  $\underline{E}(G) = (\underline{S}, s_0, \underline{I}, \text{PLAN}, \text{GOTO})$  be the evaluator

///

##### EXAMPLE 7:

Let us apply algorithm 5 to the AG of the previous examples for  $k = 2$ . The history graph  $H_1$  was given in example 6. The history graph  $H_2$  is (we have not mentioned the labels on the arcs that was present in  $H_1$ )





Corresponding to this history graph we have the PLAN and GOTO entries from example 6 and these new ones:

$\underline{S} \ \underline{X} \ \underline{I}$	PLAN	GOTO
$(s5, \{z\})$	$\emptyset, (X1, \{x1, a1, b1\}) (X2, \{x2, a2, b2\}), \{e\}$	s12
$(s6, \{x, a, b\})$	$\{a\}, (X, \{x, a, b\}), \{d\}$	s9
$(s7, \{x, a, b\})$	$\{a\}, (X, \{x, a, b\}), \{d\}$	s10
$(s8, \{x, a, b\})$	$\{a\}, \lambda, \{d\}$	s11
$(s9, \{x, a, b\})$	$\emptyset, (X, \{x, a, b\}), \emptyset$	s9
$(s10, \{x, a, b\})$	$\emptyset, (X, \{x, a, b\}), \emptyset$	s10
$(s11, \{x, a, b\})$	$\emptyset, \lambda, \emptyset$	s11

We have the new state  $s12 = ((p1, Q5, Q5), A(Z) \cup A(X1) \cup A(X2))$ .

///

Ofcause several lemmas and theorems have to be proved in order to show

that these algorithms construct an evaluator correctly. The two main things to be shown are

- the evaluator specifies a  $k$  left-to-right pass evaluation strategy when applied to a derivation tree
- all the attributes at the root of the tree have been evaluated when returning from the last call of VISIT to the root of the tree.

If the algorithms are applied to an AG augmented with dummy synthesized attributes as described at the end of section 5.1 then all the attributes associated with the nodes of a derivation tree will be evaluated when the evaluator is applied to the tree. But ofcourse it requires a formal proof.

As we put restrictions on the AGs in chapter 4 in order to obtain some simplified states of the evaluators we may do the same here.

One possibility is to consider the AGs with the  $k$  left-to-right pass property and where there can be constructed an evaluator with states  $(p^+, A)$  where  $p^+ = (P, Q, Q_1, Q_2, \dots, Q_n)$ ,  $p: X ::= X_1 X_2 \dots X_n$  and  $Q$  and  $Q_j$  are SYM-graphs for  $X$  and resp.  $X_j$  for  $1 \leq j \leq n$ . That is we assume that the weights are fixed when the SYM-graphs are known.

Another possibility is to remove all information about the SYM-graphs from the state, i.e. the states have the form  $(p, A)$ . This class of AGs has some resemblance with the benign (and adiquate) AGs.

Also the  $A$ -part of the state may be simplified and we can obtain subclasses of AGs with evaluators with states of the form  $(p^+, m)$  or  $(p, m)$  where  $m$  is the number of passes that have been performed. The subclass of AGs with evaluators with states  $(p, m)$  are (presumable) equivalent to the multipass AGs of [Boc76].

However further research are required in order to investigate these and other subclasses of AGs.



## 6. CONCLUSION

In this chapter I will briefly review the main results discussed in this thesis and I will give some proposals for further research.

The chapter consists of three sections. In the first one I briefly mention the main results and outline some results that have to be improved.

In section 6.2 I discuss how some of the results developed above can be considered as the very beginning of a theory about how to compose and decompose AGs.

And at last, in section 6.3, I give my final comments.

### 6.1 REVIEW

As mentioned in chapter 1 the AGs have not been considered very much from a theoretical point of view. In fact no commonly accepted definition of an AG, its language and its translation seem to exist.

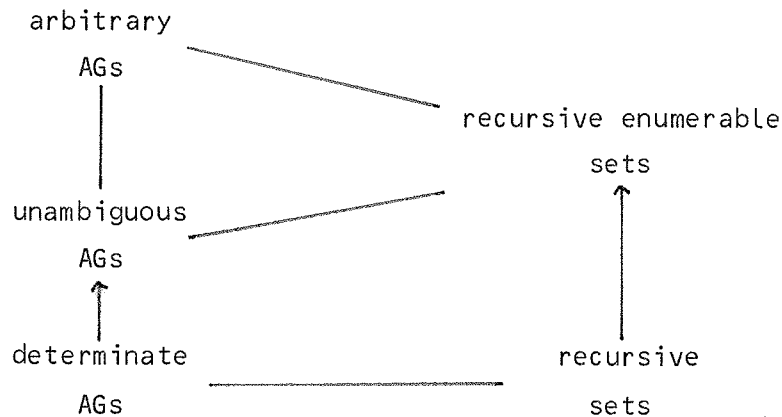
The first part of this thesis (chapter 2 and 3) has been an attempt to clarify concepts concerning how to define the language and the translation specified by an AG. Let me mention three of the aspects from these chapters that I think are among the most interesting and promising ones.

The translation of a string can be defined as the minimal fixpoint of a certain function. This is ensured by requiring that the domains of the attributes are complete lattices and that the semantic functions are continuous functions operating on the complete lattices. The definition makes it much easier to prove the equivalence and correctness of various evaluation methods. Another important property of the definition is that it constitutes a descriptive rather than an algorithmic definition of the translation specified by an AG. An interesting observation is, however, that most of the alternative definitions coincide for the well-defined AGs.

As the second point I think that both the translational and the traditional approach to language definition are legitimate. In a compiler writing system one will (presumably) not let the translation of a program be an attribute of the start symbol as in the translational approach since that may give considerable space problems. Several proposals to overcome

the problem exist e.g. in [Rhi77], [JMR78], and [Wil78]. These three proposals have some resemblance with the traditional approach. One can, however, note that some of the proposals put restrictions on the type of translations that can be specified by the various systems.

The third aspect that I want to emphasize is the introduction of the subclasses of unambiguous and determinate AGs. I think that these subclasses will turn out to be very natural although some research have to be done. For instance the AGs considered by [Maj79] are subclasses of the unambiguous AGs whereas one usually only consider subclasses of the determinate AGs. An interesting observation is, however, that the determinate AGs are able to specify exactly the recursive sets whereas the unambiguous AGs are able to define any recursive enumerable set. Thus we have for languages recognized by AGs (relying on the Church-Turing thesis)

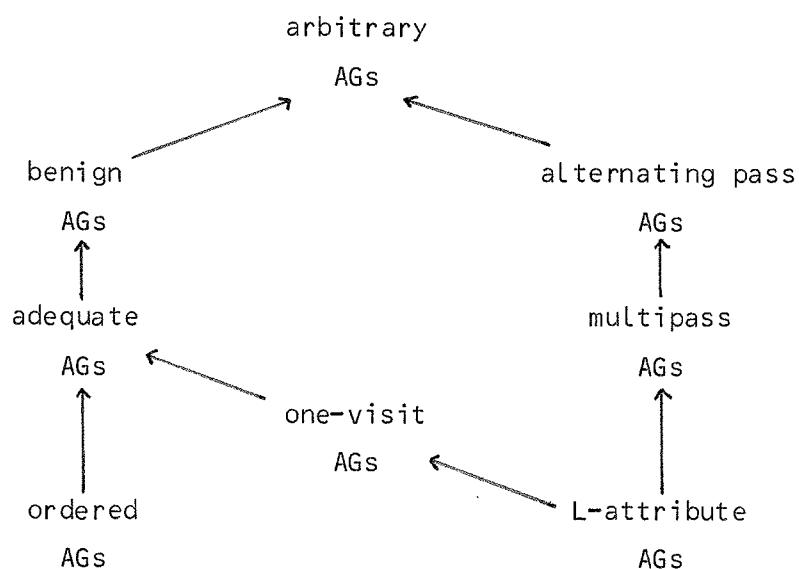


In the second part of the thesis (chapter 4 and 5) I introduce new methods by which the AGs may be characterized.

Usually an AG is characterized by that it belongs to a subclass of AGs. The subclasses are often defined by putting some restrictions on the type of dependencies that may be between the attributes. Some examples are:

- the L-attribute grammars of [LRS74], [Boc76], and [May78]
- the multipass AGs of [Boc76] and [Sch76]
- the alternating pass AGs of [JaW75], [Sch76], [Rhi77], and [Poz79]
- the ordered AGs of [Kas78] and [Sch76]
- the adequate (or absolutely well-defined) AGs of [Kew76] and [Sch76]
- the benign AGs of [May78]
- the one-visit (or reordered) AGs of [EnF79] and [May78]
- the well-defined AGs of [Knu71], [Sch76], [May78] and [CoH79]

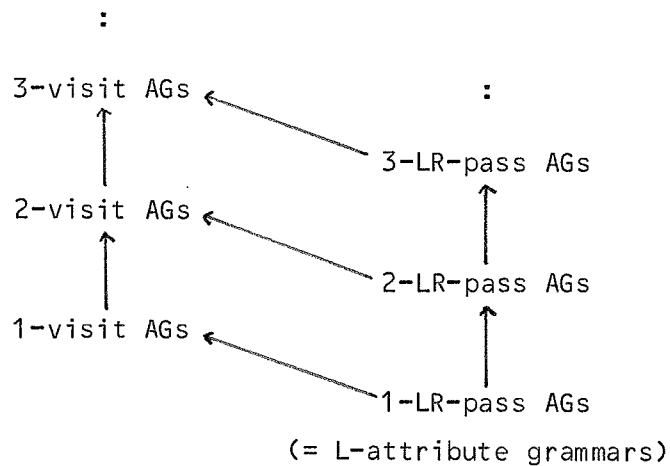
These subclasses give rise to the following hierarchy of AGs. (We use a general version of the multipass and alternating pass AGs as e.g. proposed in chapter 5 for the multipass case. If we instead use the original definitions of [Boc76] and [JaW75] then the multipass and alternating pass AGs will be subclasses of the ordered AGs.) I expect the unconnected subclasses to be incomparable whereas the arrows represent proper inclusions.



These subclasses are defined on the basis of properties of the grammars and not the translations specified by the grammars. For instance the transformation of a well-defined AG into a partly uniform AG in chapter 3 shows that the well-defined, the benign and the adequate AGs are able to specify the same translations. An obvious topic for further research will be to compare the classes of translations specified by the subclasses of AGs above.

If we know that an AG is k-visit we will have more information about how 'complicated' the translation specified by the AG is than if we only know that it is e.g. benign. Thus I think that the k-visit and the k-LR-pass properties will be useful properties of the AGs when considering the translations specified by AGs. However also here much research remains.

The k-visit and k-LR-pass AGs constitute hierarchies as shown on the figure below. The same inclusions hold for the classes of translations specified by the classes of AGs.



The  $k$ -visit and the  $k$ -LR-pass properties have (at least) one weakness: they are defined on the basis of a specific definition of a visit and this definition is in turn inspired by a specific structure of an evaluation algorithm. I think that it is very difficult to remove this weakness entirely although something may be done. Nevertheless the definitions may be useful.

Some of the results presented in chapter 4 and 5 can presumably be improved. Let me mention three of them:

We have that there exist a translation that can be specified by a  $k$ -visit AG but which cannot be specified by a  $(k-1)$ -visit AG over the same semantic domain and with the same underlying grammar. I think that the condition 'with the same underlying grammar' can be removed. Similarly the same condition may be removed from the result about 1-visit and  $k$ -LR-pass AGs in chapter 5.

Another problem that has to be solved is how (deterministically) to determine the minimal  $k$  such that an AG is  $k$ -visit. We have only considered how to find bounds within which the minimal  $k$  will be.

The third of the results that ought to be improved is the test for the  $k$ -LR-pass property. Given an AG and a  $k$  we have seen how to test whether the AG is  $k$ -LR-pass. But we may be interested in testing whether the AG is  $k$ -LR-pass for any  $k$ .

Furthermore in all the previous chapters the complexity of the various tests and constructions have to be analysed.

## 6.2 COMPOSITION AND DECOMPOSITION OF AGs

The results in chapter 5 may be considered as the very beginning of a theory for composition and decomposition of AGs.

Consider two AGs  $G_1$  and  $G_2$  with underlying grammars  $G_1' = (V_{n1}, V_{t1}, R_1, Z_1)$  and  $G_2' = (V_{n2}, V_{t2}, R_2, Z_2)$  satisfying

- there is a homomorphism  $h: V_1 \rightarrow V_2$
- if  $p: X ::= X_1 X_2 \dots X_n$  is in  $R_1$  then  $p': h(X) ::= h(X_1) h(X_2) \dots h(X_n)$  is in  $R_2$ .

Let us denote this property of the AGs with structural equivalence.

We may then define the composition of  $G_1$  and  $G_2$  denoted  $G = G_1 + G_2$  as an AG structural equivalent to  $G_1$  and  $G_2$ . The set of attributes associated with a symbol in  $G$  will be the union of the attributes for the corresponding symbol in  $G_1$  and  $G_2$ . The semantic rules of a production in  $G$  is the 'union' of those for the corresponding productions in  $G_1$  and  $G_2$ .

The composition can easily be extended to any number of AGs:  $G = G_1 + G_2 + \dots + G_k$  where the  $G_j$ 's are structural equivalent for  $1 \leq j \leq k$ .

Composition of AGs becomes especially interesting when we put an ordering on the AGs. Let  $G_1, G_2, \dots, G_k$  be a set of structural equivalent AGs. We will allow the semantic rules of  $G_j$  to have attribute variables occurring in  $G_i$  as parameters for  $1 \leq i \leq j$  and  $1 \leq j \leq k$ . Intuitively the translation specified by  $G = G_1 + G_2 + \dots + G_k$  can be obtained as follows:

Let  $t_0$  be an initial semantic tree and use an evaluator for  $G_1$  to construct a semantic tree  $t_1$ . In stead of using an initial semantic tree as 'input tree' when evaluating the attributes of  $G_2$  we will use  $t_1$ . The resulting semantic tree  $t_2$  will contain the values of the attributes for the symbols in both  $G_1$  and  $G_2$ . By letting  $t_{(j-1)}$  be the 'input tree' to an evaluator for  $G_j$  and  $t_j$  the resulting semantic tree for  $1 \leq j \leq k$  we will have that  $t_k$  is the semantic tree constructed by applying an evaluator for the AG  $G$  to the initial semantic tree  $t_0$ .

Formally this ordering may be expressed by the introduction of a third kind of attributes for the symbols in the AG: the intrinsic attributes defined by [Sch76]. The value of an intrinsic attribute is defined by an external semantic rule and thus it may be treated as a constant by the AG.

The results of chapter 5 may be considered as testing whether an AG  $G$  can be decomposed into  $k$  L-attribute grammars for a given  $k$  and if it is possible then to determine the AGs  $G_1, G_2, \dots, G_k$  such that  $G = G_1 + G_2 + \dots + G_k$ . This can easily be seen by changing a state  $(p^+, A)$  in the evaluator to a production:

Let  $p^+ = (p, Q, Q_1, Q_2, \dots, Q_n)$  and  $p: X ::= X_1 X_2 \dots X_n$ . We will construct a production  $p': X' ::= X_1' X_2' \dots X_n'$  where  $X' = (X, Q, A \cap A(X))$  and  $X_j' = (X_j, Q_j, A \cap A(X_j))$  for  $1 \leq j \leq n$ . The underlying grammar of  $G_1$  will contain the productions obtained from the initial states. The underlying grammar for  $G_j$  will contain the productions obtained from states that the nodes of a tree may be in after the  $(j-1)$ 'th pass over the tree. The homomorphism  $h$  in the definition of structural equivalence of AGs can be obtained from the GOTO table.

Of course one may be interested in the decomposition of AGs into other than L-attribute grammars and here much (almost all) research remains. The same is the case if one consider composition of AGs. Let for instance  $G_1, G_2, \dots, G_k$  be 1-visit AGs. One can then ask: will  $G = G_1 + G_2 + \dots + G_k$  be a benign AG or is it possible to choose the 1-visit AGs such that  $G$  is not benign?

### 6.3 FINAL REMARKS

In the introduction I mentioned two important aspects of compiler writing systems based on attribute grammars:

- How can we determine an order for evaluating the attributes associated with a derivation tree
- What kind of information can be stored in the attributes and what kind of operations are allowed in the semantic rules.

And I asked the question: how important are the various restrictions on the AGs for the translations that can be specified.

Although much research remains we can give some preliminary answers.

The results about the  $k$ -visit and the  $k$ -LR-pass hierarchies show that the domains for the attributes and the operations on the attributes are very important for the translations that can be specified.

We have seen two methods that can be used to construct evaluators for AGs. They represent different strategies for choosing an order in which the

attributes of a derivation tree can be evaluated.

If one care about the domains of and the operations on the attributes then we see that different strategies may result in different classes of translations.

## REFERENCES

-----

- [AHU74] A.V.Aho, J.Hopcroft, and J.D.Ullman:  
"The design and analysis of computer algorithms"  
Addison-Wesley Publishing Company (1974)
- [AhU72] A.V.Aho and J.D.Ullman:  
"The theory of parsing, translation and compiling"  
Vol.1. Englewood Cliffs, Prentice Hall (1972)
- [ALb79] H.Alblas:  
"The limitations of attribute evaluation in passes"  
Manuscript, Twente University of Technology (1979)
- [Boc76] G.V.Bochmann:  
"Semantic evaluation from left to right"  
Comm ACM 19 (1976)
- [CoH79] R.Cohen and 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 (1979)
- [EnF79] J.Engelfriet and G.Filè:  
"The formal power of one-visit attribute grammars"  
Technische Hogeschool Twente, Enschede (1979)
- [GRW77] H.Ganzinger, K.Ripken, and R.Wilhelm:  
"Automatic generation of optimizing multipass compilers"  
Information Processing 77, Proceedings of IFIP Congress 77, North-  
Holland Publ. Co. (1977)
- [JaW76] M.Jazayeri and K.G.Walter:  
"The alternating semantic evaluator"  
Proceedings of the ACM 1975 annual conference (1975)



- [JOR75] M.Jazayeri, W.F.Ogden, and W.C.Rounds:  
"The intrinsically exponential complexity of the circularity  
problem for attribute grammars"  
CACM 18, 12 (1975)
- [JMR78] P.Jespersen, M.Madsen, and H.Riis:  
"A compiler writings system: The extended attribute translation  
system (EATS)"  
DAIMI, University of Aarhus (1978)
- [Jon79] N.D.Jones:  
Private communication  
DAIMI, University of Aarhus (1979)
- [Kas78] U.Kastens:  
"Ordered attribute grammars"  
Bericht nr 7/78, Institute für informatik, Universität Karlsruhe  
(1978)
- [Kew76] K.Kennedy and S.K.Warren:  
"Automatic generation of efficient evaluators for attribute gram-  
mars"  
Conference record of the third ACM symposium on principles of  
programming languages (1976)
- [Knu68] D.E.Knuth:  
"Semantics of context free languages"  
Mathematical systems theory 2, 2 (1968)
- [Knu71] D.E.Knuth:  
"Semantics of context free languages: correction"  
Mathematical systems theory 5, 1 (1971)
- [LRS74] P.M.Lewis, D.J.Rosenkrantz, and R.E.Stearns:  
"Attributed translations"  
Journal of computer and system science 9 (1974)

- [Lor77] B.Lorho:  
"Semantic attribute processing in the system DELTA"  
Methods of algorithmic language implementation. Lecture Notes of  
Computer Science 47, Springer Verlag (1977)
- [MaJ79] M.Madsen and N.D.Jones:  
"Letting the attributes influence the parsing"  
DAIMI, University of Aarhus (1979)
- [Mad75] O.L.Madsen:  
"On the use of attribute grammars in a practical translator writing  
system"  
DAIMI, University of Aarhus (1975)
- [Man74] Z.Manna:  
"Mathematical theory of computation"  
McGraw-Hill Inc (1974)
- [May78] B.H.Mayoh:  
"Attribute grammars and mathematical semantics"  
DAIMI PB-90, University of Aarhus (1978)
- [Poz79] D.P.Pozefsky:  
"Building efficient pass-oriented attribute grammar evaluators"  
University of North Carolina at Chapel Hill (1979)
- [Räi77] K-J.Räihä:  
"On attribute grammars and their use in a compiler writing system"  
Report A-1977-4, University of Helsinki (1977)
- [Sch76] W.A.Schulz:  
"Semantic analysis and target language synthesis in a translator"  
University of Colorado, Boulder, Colorado (1976)
- [Sto77] J.E.Stoy:  
"Denotational Semantics - The Scott-Strachey Approach to Program-  
ming Language Theory"  
MIT Press (1977)

[Tie79] M.Tienari:

"A note on the definition of attribute grammars"  
Report C-1979-29, University of Helsinki (1979)

[Wat74] D.A.Watt:

"Analysis oriented two-level grammars"  
Ph.D.Thesis, University of Glasgow (1974)

[Wil78] R.Wilhelm:

"Presentation of the compiler writing system MUG2, examples:  
Global flow analysis and optimization"  
Le point sur la compilation, IRIA (1978)