Attributed Abstract Program Trees

Henk Alblas and Frans J. Faase

Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands

Traditionally, an attribute grammar is presented as a context-free grammar which is augmented with attributes and attribute evaluation rules. This makes attribute grammars a suitable means for the specification of the semantics of programming languages in the context of derivation trees. For the specification of semantic integrity constraints in the context of abstract program trees the concept of attribute grammars has to be re-defined. For this purpose, a language for the specification of context-free tree grammars is defined. This language is extended to an attribute tree grammar specification language.

1. Introduction

In the classical theory [7] attribute grammars form an extension of the context-free grammar framework in the sense that information is associated with programming language constructs by attaching attributes to the grammar symbols representing these constructs. Each attribute has a set of possible values. Attribute values are defined by attribute evaluation rules associated with the productions of the context-free grammar.

The attributes associated with a grammar symbol are divided into two disjoint classes, the synthesized attributes and the inherited attributes. The attribute evaluation rules associated with a production define the synthesized attributes attached to the grammar symbol on the left-hand side and the inherited attributes attached to the grammar symbols on the right-hand side of the production.

A non-ambiguous context-free grammar assigns a single derivation tree to each sentence. The values of the synthesized attributes at a node of a derivation tree and the inherited attributes at its immediate descendents are defined by the attribute evaluation rules associated with the production applied at that node. The value of a synthesized attribute of the parent is computed from the values of the attributes at its children and (possibly) other attributes of the parent itself. The value of an inherited attribute of a child is computed from the values of attributes at its parent and its siblings and (possibly) other attributes of the child itself.

Generally speaking, a synthesized attribute attached to a tree node contains information concerning the subtree at that node. This

attribute therefore contains information from the terminal string derived from the nonterminal symbol labeling that node. Inherited attributes are convenient for expressing the dependence of a programming language construct on the context in which the construct appears.

The traditional way of thinking about attribute grammars is in terms of derivation trees. First, the parser generates a derivation tree for a given program. Next, the attribute evaluator computes the values of the attribute instances attached to the nodes of the derivation tree by executing the attribute evaluation rules associated with these attribute instances.

In this research we consider attribute grammars for abstract program trees in which nonterminal symbols are no longer used and where operators and keywords do not appear as leaves, but rather are associated with the interior node that would be the parent of those leaves in the derivation tree. Another reduction found in abstract program trees is the elimination of chain productions. This allows an abstract program tree to be viewed as a compact and simplified representation of a derivation tree, where each operator is represented by an interior node whose children represent the arguments of that operator and where all redundant information needed for the syntactical analysis has been deleted. In other words, an abstract program tree is a non-redundant representation of the hierarchical structure of a source program. In this paper we want to describe abstract program trees as a separate concept, not as an adaptation of context-free grammars. We also do not discuss the conversion of derivation trees of a context-free grammar to abstract program trees, nor do we discuss how context-free grammars can be transformed into abstract program trees as described in [10].

Essentially the compilation process consists of an analysis phase and a synthesis phase. The result of the analysis phase (i.e., lexical and syntactic analysis) is a tree (possibly in a linearized form), which expresses the structure of the source program. Attributes can be attached to the nodes of the tree to carry semantic information. Semantic analysis (e.g., type checking) can be expressed by attribute evaluation rules and semantic conditions. The aim of the synthesis phase is the inclusion of necessary constraint checks (e.g., array bound checks) and the translation of the control structures and the data structures of the source program into the instructions and the storage locations of the target machine. Our ultimate goal is to specify these translations by a stepwise application of tree transformations, starting from the structures of the source program and ending with the structures of the target machine.

Tree transformations also form a suitable means for the specification of compiler optimizations. These transformations replace complicated and non-efficient tree structures by equivalent but simpler and more efficient tree structures.

For the specification of tree transformations, both for the purpose of translations and for optimizations, the classical attribute grammar framework has to be extended with conditional tree transformation rules [2,6,11]. The predicates on attribute values (carrying context information) may be used to enable the application of these transformations.

So, we are interested in non-redundant program trees, which can be used as a concept to define the information flow of the associated program and which can also be considered as an object to be operated on. In this paper we restrict ourselves to the description of the structure and the attribution of abstract program trees. We will deal with tree transformations in a future paper.

This paper is organized as follows. Section 2 provides an introduction to the basic concepts of the classical attribute grammar framework, based on context-free grammars. A language for the specification of abstract program trees is given in Section 3. In Section 4 this language is extended to an attribute grammar specification language. Concluding remarks are made in Section 5.

2. Classical Attribute Grammars

In the classical theory [7] an attribute grammar AG is based on a context-free grammar G which is augmented with attributes and attribute evaluation rules.

The underlying grammar G is a 4-tuple (V_N, V_T, P, S). The finite sets V_N of nonterminal and V_T of terminal symbols form the vocabulary $V = V_N \cup V_T$. P is the set of productions and $S \in V_N$ is the start symbol, which does not appear in the right part of any production. The grammar G is reduced in the sense that each nonterminal symbol is reachable from the start symbol and can generate a string which contains no nonterminal symbols.

Each symbol $X \in V$ has a finite set $A(X)$ of attributes, partitioned into two disjoint subsets $I(X)$ and $S(X)$ of inherited and synthesized attributes, respectively.

Let P consist of r productions, numbered from 1 to r and let the p-th production be

$$X_{p0} \rightarrow X_{p1} X_{p2} \cdots X_{pn_p}$$

where $n_p \geqslant 0$, $X_{p0} \in V_N$ and $X_{pk} \in V$ for $1 \leqslant k \leqslant n_p$.

Production p is said to have the attribute occurrence (a,p,k) if $a \in A(X_{pk})$. The set of attribute occurrences of production p will be denoted by $AO(p)$. This set can be partitioned into two disjoint sets of defined occurrences and used occurrences denoted by $DO(p)$ and $UO(p)$, respectively.

These subsets are defined as

$$DO(p) = \{(s,p,0) \mid s \in S(X_{p0})\} \cup \{(i,p,k) \mid i \in I(X_{pk}) \wedge 1 \leqslant k \leqslant n_p\}$$

$$UO(p) = AO(p) - DO(p) = \{(i,p,0) \mid i \in I(X_{p0})\} \cup \{(s,p,k) \mid s \in S(X_{pk}) \wedge 1 \leqslant k \leqslant n_p\}$$

Associated with each production p is a set of attribute evaluation rules which specify how to compute the values of the attribute occurrences in $DO(p)$. The evaluation rule defining attribute occurrence (a,p,k) has the form

$$(a,p,k) := f((a_1,p,k_1),(a_2,p,k_2), \cdots ,(a_m,p,k_m))$$

where $(a,p,k) \in DO(p)$, f is a total function and $(a_j,p,k_j) \in AO(p)$ for $1 \leqslant j \leqslant m$.

An attribute grammar is said to be in normal form if the extra condition $(a_j,p,k_j) \in UO(p)$ holds for $1 \leqslant j \leqslant m$. It is easy to transform every attribute evaluation rule (by a sequence of transformations) such that only attribute occurrences in $UO(p)$ appear as arguments of f.

For each sentence of G a derivation tree exists. The nodes of the tree are labeled with symbols from V. At each inner node a production $p: X_{p0} \rightarrow X_{p1} X_{p2} \cdots X_{pn_p}$ is applied, such that the node is labeled with X_{p0} and its sons with $X_{p1}, X_{p2}, \cdots , X_{pn_p}$.

Given a derivation tree, instances of attributes are attached to the nodes in the following way: if node N is labeled with grammar symbol X, then for each attribute $a \in A(X)$ an instance of a is attached to node N. An attribute instance a of node N will be denoted by a of N.

Let N_0 be a node, p the production at N_0 and $N_1, N_2, \cdots , N_{n_p}$ its sons from left to right, respectively. An attribute evaluation instruction

$$a \text{ of } N_k := f(a_1 \text{ of } N_{k_1}, a_2 \text{ of } N_{k_2}, \cdots , a_m \text{ of } N_{k_m})$$

is associated with attribute instance a of N_k if the attribute evaluation rule

$$(a,p,k) := f((a_1,p,k_1),(a_2,p,k_2), \cdots ,(a_m,p,k_m))$$

is associated with production p.

The task of an attribute evaluator is to compute the values of all attribute instances attached to the derivation tree by executing their associated evaluation instructions. In general the order of evaluation is unimportant, with the only restriction that an attribute evaluation instruction cannot be executed before the values of its arguments are

available. Initially the values of all attribute instances attached to the derivation tree are undefined, with the exception of the inherited attribute instances attached to the root (containing information concerning the environment of the program) and the synthesized attribute instances attached to the leaves (determined by the parser).

At each step we choose an attribute instance whose value can be computed. The evaluation process continues until all attribute instances in the derivation tree are defined or until none of the remaining attribute instances can be evaluated.

An attribute grammar is circular if a derivation tree exists for which it is not possible to evaluate all attribute instances.

Several methods have been developed to evaluate the semantic attributes within the derivation tree of a program. An overview is given in [4].

3. Abstract Program Trees

In this section we present a mechanism to define abstract program trees. One could think of defining an abstract program tree by a transformation applied to a derivation tree. This is, however, not a suitable approach as we want the definition mechanism of abstract program trees to be a framework that can easily be extended to an attribute grammar-like definition. Moreover, we view an abstract program tree as an intermediate structure subject to further transformations. The concept of a derivation tree is therefore not a good starting point. Instead, we shall use a completely different model, namely graphs. Starting with graphs we will make a number of restrictions which will finally bring us to a grammar for the specification of abstract program trees.

3.1. Starting with Graphs

First of all we restrict our graphs to be directed and we assume every graph includes one node from which all other nodes can be reached when following the arcs. We will call this distinguished node the root node, because for trees it can be considered as the node on which the whole tree stands (or from which it hangs).

Secondly, we will color the nodes and the arcs (using graph terminology) and put some restrictions on the coloring. In our terminology we would refer to typing and labeling. The idea is to associate with every node a type and to define node types by a type rule that determines the number of outgoing arcs, their labeling (or distinguishing colors) and the permitted node types pointed to by these arcs.

We will now have a closer look at the type system that we are going to use. For each node type a (possibly empty) set of pairs could be given determining the number of arcs. Each pair consists of a

unique label and a non-empty set of allowed node types.
This leads to

$$
\begin{aligned}
& RULE(node\_type) \\
= \quad & \{ (label_1, \{node\_type_{1,1}, \cdots, node\_type_{1,m_1}\}) \\
& , \cdots \\
& , (label_n, \{node\_type_{n,1}, \cdots, node\_type_{n,m_n}\}) \\
& \}
\end{aligned}
$$

where $n \geqslant 0$, $m_i > 0$ for $1 \leqslant i \leqslant n$, and $label_i \neq label_j$ for $1 \leqslant i < j \leqslant n$.

Now a family of graphs can be defined by a 3-tuple (N,L,R), where N is the set of node types, L is the set of labels and R is the set of type rules with a single type rule for every node type. Each member of this family is a typed graph in which all the nodes have the right number of arcs with the right labels, and the right node types at the end of the arcs.

3.2. Towards an Abstract Program Tree Grammar

Having constructed a language for the definition of graphs one could think of an extension of the classical attribute grammar framework from trees to graphs. However, such an extension introduces complications which go far beyond the scope of this paper. For this reason in this paper we restrict ourselves to trees only.

The above-mentioned type system applied to trees defines for every node type a fixed number of sons. The type of the father node puts some restrictions on the types of the son nodes. Every tree node can be reached from the root node through a unique path. The root node is considered as the highest node from which all other nodes hang.

We require a strict ordering of the sons of a node, which is the same for all the nodes of the same type. Such an ordering is needed for the definition of certain tree traversal strategies (e.g., especially for the pass-oriented strategies [1]).

In Section 3.1 the node types allowed at a certain arc have been written as a set. In practical applications, these sets are often similar. We therefore introduce an abbreviation mechanism called classes for these sets. A class will be defined as a subset of the node types. Having defined classes, trees can be defined by so-called tree rules, which are similar to the node type definitions in the previous section, but differ in that the labels have a fixed order, and the sets are replaced by a class or a node type. The labels will be called partnames, because they indicate a part of the tree under a node.

Tree rules are of the form

$$
\begin{aligned}
& a\_node\_type \\
\Longrightarrow \quad & partname_1 : element_1 \\
& , \cdots \\
& , \quad partname_n : element_n \ .
\end{aligned}
$$

where $n \geqslant 0$, $element_i \in node\_types \cup classes$ for $1 \leqslant i \leqslant n$ and $partname_i \neq partname_j$ for $1 \leqslant i < j \leqslant n$.

In our formalism we shall also expand the class definitions. In most applications it is possible to divide the members of a certain class into groups that have the same properties, i.e., they have tree rules that are similar if we look at the allowed node types at their parts. This observation leads to two extensions of the previous definition.

We first introduce a hierarchical class definition by allowing classes to have other classes as their members. A class definition is of the form

$$a\_class = \{element_1, \cdots, element_n\}$$

where $n > 0$ and $element_i \in node\_types \cup classes$ for $1 \leqslant i \leqslant n$.
The introduction of hierarchical class definitions requires some restrictions. We exclude recursive class definitions, as for example

$$class\_A = \{class\_B, class\_A, node\_type\_N\}$$
$$class\_B = \{class\_A, node\_type\_M\}$$

We now introduce the concept of the closure over the class definitions. We shall use the function $class(C)$ to denote the set of members of class C. Likewise we shall use the function $clos\_class(C)$, for the closure over the member-of-class relation that will return all the members of $class(C)$ together with the $clos\_class$ of all the classes in $class(C)$.

Although recursive class definitions are not allowed, there remains a kind of ambiguity, as illustrated by the example

$$class\_A = \{class\_B, class\_C\}$$
$$class\_B = \{node\_type\_N, node\_type\_M\}$$
$$class\_C = \{node\_type\_N, node\_type\_K\}$$

In this example $node\_type\_N$ is a member of $clos\_class(class\_A)$, but in the case of an instance of $class\_A$ in an abstract program tree, where $node\_type\_N$ is selected, it is not clear whether $node\_type\_N$ is a member of $class\_B$ or $class\_C$. The question whether we will allow these ambiguous class definitions, is further dealt with in Section 4.1.

Secondly, we make the extension that tree rules may also be associated with a class. This means that in addition to a $node\_type$, we will also allow a class as the left-hand side of a tree rule. The tree rule associated with the class C holds also for all the elements in $clos\_class(C)$.

In the previous section we decided to associate exactly one tree rule with each node type. With respect to classes this rule requires that once a tree rule has been written for a class, no tree rule may be written for a member (or a member of a member, etc) of the class.

Also, if an element is a member of more than one class, it is not possible that more than one of these classes has an associated tree rule.

3.3. Formal Definition of Abstract Program Trees

In this section the concept of an Abstract Program Tree Grammar (APTG) will be defined formally. An APTG can be defined as a 5-tuple (V_N, V_C, TR, CD, R). The finite set V_N of node types and V_C of classes form the set of elements $V = V_N \cup V_C$. TR is the set of tree rules and CD is the set of class definitions. R is the root element which may be either a class or a node type.

The members of TR will be of the form

$$V_0 \Longrightarrow P_1 : V_1, \cdots, P_n : V_n \ .$$

where $n \geqslant 0$, $V_i \in V$ for $0 \leqslant i \leqslant n$, P_i is a partname and $P_i \neq P_j$ for $1 \leqslant i < j \leqslant n$.

The members of CD will be of the form

$$C = \{V_1, \cdots, V_n\}$$

where $C \in V_C$, $n > 0$ and $V_i \in V$ for $1 \leqslant i \leqslant n$. For each $C \in V_C$ there is exactly one member in CD which has C as its left-hand side. All the members of CD together should not include recursive class definitions. TR and CD combined should not define more than one tree rule for each node type of the grammar.

We will define the function $class\,(e:V)$ in the context of CD as

$class\,(e:V)$
= **if** $e \in V_N$
then $\emptyset$
else $\{V_1, \cdots, V_n\}$
where '$e = \{V_1, \cdots, V_n\}$' $\in CD$
fi

We will now define the set of abstract program trees defined by an APTG. We do not talk about derivations here, thus this cannot be viewed as an extension from string to tree grammars. Before we can do this we have to choose a representation for trees. We use the representation where every subtree is represented by the node type of its root, followed by the sequence of representations of its subtrees (in the same order as they are hanging in the tree), enclosed in the brackets "<" and ">". The brackets may be omitted if a node type has no sons. Each node type name with its associated brackets represents one instance of a node, with that node type, and the links (or arcs) to its subtrees. Because we have only one tree rule associated with every node type, this representation will be complete. The following example depicts the tree representing the arithmetic expression 3*(4+5)

mul_op < *num* , *plus_op* < *num* , *num* > >

In this representation the values of the numbers are not represented.

We first define a function *Tree* that returns all subtrees for a certain root element as

$$
\begin{aligned}
&Tree(V_N, V_C, TR, CD, s \in V) \\
= \quad &\{ N < t_1, \cdots, t_n > \\
&\mid N \in V_N \cap clos\_class\_and(s) \\
&\wedge\ 'V_0 \Longrightarrow P_1{:}V_1, \cdots, P_n{:}V_n' \in TR \\
&\wedge\ N \in clos\_class\_and(V_0) \\
&\wedge\ \forall 0 \leqslant i \leqslant n \\
&\qquad (t_i \in Tree(V_N, V_C, TR, CD, V_i)) \\
&\} \\
&\textbf{where}\ clos\_class\_and(s \in V) \\
&\qquad = \{s\} \cup \bigcup_{e \in class(s)} clos\_class\_and(e)
\end{aligned}
$$

Using this definition we can define the function *Tree* that for a given APTG G yields the set of all trees that are defined by it as

$$Tree(G{:}APTG) = Tree(V_N(G), V_C(G), TR(G), CD(G), R(G)) .$$

Above we have essentially presented a language to describe APT's, and defined the set of APT's that are defined by a given APTG. This language shows certain similarities with the Interface Description Language IDL [8,9]. A more restricted formalism for the description of abstract program trees is presented in [3].

4. Attributed Abstract Program Trees

For the specification of the information flow in abstract program trees we will augment our abstract program tree grammars with attributes and attribute evaluation rules in a similar way as classical attribute grammars have evolved from context-free grammars. However, for abstract program tree grammars this extension turns out to be far more complex.

4.1. How to Add Attributes

Abstract program trees are assumed to be decorated with attributes in the same way as attributes are attached to the nodes of a derivation tree in a classical attribute grammar implementation. As for classical attribute grammars, the attribute instances attached to an abstract program tree can be partitioned into inherited and synthesized attributes, according to the way they carry and receive their information to and from other attribute instances. The simplest way of defining the attributes of abstract program trees is to associate with each node type $E \in V_N$ a finite set $A(E)$ of attributes, partitioned into two disjoint subsets $I(E)$ and $S(E)$ of inherited and synthesized attributes, respectively.

Given a tree rule, the problem occurs of how to identify the attribute occurrences of this tree rule, as both the left-hand side and the right-hand side may contain classes which do not have attributes. In the rest of this paper we shall identify the elements of a tree rule by their partnames, and introduce # as the partname for the left-hand side element.

We distinguish two ways to identify an attribute occurrence of a tree rule $V_0 \Longrightarrow P_1{:}V_1, \cdots, P_n{:}V_n$. The first way is to write it as *attr* of $P_i.N$, where $0 \leqslant i \leqslant n$, $P_0 = \#$, $attr \in A(N)$ and $N \in V_N \cup clos\_class\_and(V_i)$. The second way is to write it as a sequence of elements expressing the hierarchical structure of classes containing classes, in the following way: *attr* of $P_i.E_1.\cdots.E_m$, where $attr \in A(E_m)$, $E_m \in V_N$, $m \geqslant 1$, $E_1 = V_i$, $E_j \in V_C$ and $E_{j+1} \in class(E_j)$ for $1 \leqslant j < m$.

At first sight the second method seems to be unnecessarily complicated or appears to use redundant information. This is true if we require the inclusion of node types in classes to be non-ambiguous; cf. 3.2. If classes are allowed to be ambiguous we can ask the question whether attributes at node types, ambiguously included in a class, have to be considered as different. In the latter case, we need to distinguish such attributes, and the only way to do this is to use the second method. We shall use the first method in the rest of this paper, and leave open the question of how to handle ambiguous classes.

The set of attribute occurrences associated with the tree rule of element E will be denoted by $AO(E)$. This set can be partitioned into two disjoint sets of defined occurrences and used occurrences, denoted by $DO(E)$ and $UO(E)$, respectively.

These subsets are defined as

$$\begin{aligned} DO(E) &= \{\, s \text{ of } P_0.N \mid s \in S(N) \,\} \\ &\quad \cup \{\, i \text{ of } P_i.N \mid i \in I(N) \wedge 1 \leqslant k \leqslant n \,\} \\ UO(E) &= AO(E) - DO(E) \\ &= \{\, i \text{ of } P_0.N \mid i \in I(N) \,\} \\ &\quad \cup \{\, s \text{ of } P_i.N \mid s \in S(N) \wedge 1 \leqslant k \leqslant n \,\}. \end{aligned}$$

We have defined above how to address attribute instances in the tree by attribute occurrences, and we have defined the sets of attribute occurrences associated with a tree rule. The set of attribute instances attached to a concrete node in a tree and its sons, is generally only a subset of the attribute occurrences associated with the tree rule applied at this node. This is because attributes are attached to node types, and for a given tree rule, classes can be involved in both sides of the rule.

4.2. The Attribute Evaluation Rules

In the same way as for attribute grammars we associate with each tree rule a set of attribute evaluation rules which specify how to compute the values of the attribute occurrences in $DO(E)$, where E is the element in the left part of the tree rule. Only the evaluation rules for those attributes that are attached to a certain node in a tree are applied. But for the right-hand side of the rule some problems may arise. We cannot know whether an attribute occurrence is available at a certain position in the tree. Take for example the tree rule

plus ⟹ *left* : *expression*,
right : *expression*.

together with the class rule

expression = {*constant*, *plus*}

and an attribute *value* ∈ *A* (*constant*). At a node typed *plus* we cannot with certainty refer to *value* of *left.constant* because its identity may not be of type *constant* but of type *plus*.

To solve this problem we need a mechanism to find out which node type is applied at a position in the tree, where a tree rule has a class. We do this by introducing case-expressions on the partnames. Referring to the above example, we could for example write

case *left* **of**
constant : *value* **of** *left.constant* ;
plus : 0
esac

to express that we want the value of attribute *value* if the actual node at *left* is of type *constant*, and otherwise the value 0. In the case of hierarchical class definitions we need nested case-expressions. We shall now define the expressions which form the right-hand side of the attribute evaluation rules. We distinguish three different constructs. Firstly an attribute occurrence, secondly a general function format in which the arguments are again expressions, and thirdly the case-expression. We could describe the syntax using

expr ::= *attr* **of** *partname*
| $f(expr_1, \cdots, expr_n)$
| **case** *partname* **of**
E_1 : $expr_1$;
$\cdots$
E_m : $expr_m$
esac

We need a number of semantic conditions. To every subexpression a context can be assigned that specifies the binding of elements to partnames. In the context of the whole expression the binding is specified

by the tree rule. For an attribute occurrence *attr* of *partname*, *attr* of *partname.N* has to be a member of $UO(E)$, where N is the node type that is bound to partname *partname*, and E the element in the left part of the tree rule. The context of the arguments of a function will be the same as the context of the whole function-expression. For a case expression the following restrictions should be imposed. If we assume C to be the element bound to partname *partname*, then $C \in V_C$, $class(C) = \{E_1, \cdots, E_m\}$ and $E_i \neq E_j$ for $1 \leqslant i < j \leqslant m$. The context of each subexpression $expr_i$ has to be such that E_i will be bound to *partname*, and the rest of the context will be the same as the context of the whole case-expression.

We can now define the attribute evaluation rules to be of the form

attr of *partname.N* = *expr*

where *attr* of *partname.N* $\in DO(E)$ and *expr* is a correct expression as described above.

We could also introduce for reasons of orthogonality a case mechanism for the left-hand side of the evaluation rules. With these case constructions it is possible to combine several rules together that have the same partname for the left-hand side. Furthermore, if we make the extension that we can combine cases with the same expression into one case, then this can lead to a reduction on the size of the rules. However, these extensions are merely syntactic sugar.

4.3. Attributes Attached to Classes

In the foregoing discussion we attached attributes to the node types of the grammar only. In practical applications it often occurs that the same attributes are associated with all the members of a class. This leads to case-expressions which have the same expression for all the cases. To solve this problem, we allow an attribute to be associated with a class if it is associated with all the members of that class. Of course, this rule applies recursively over the hierarchy of the class definitions. This has implications to the definition of $AO(E)$, $UO(E)$ and $DO(E)$. We can now write *attr* of *partname.E'*, where $E' \in V$ instead of $E' \in V_N$. We can also weaken the semantic restriction on the attribute occurrences in the expression on the right-hand side of an evaluation rule. For the attribute occurrence *attr* of *partname*, *attr* of *partname.E'* has to be a member of $UO(E)$ where the element bound to the partname *partname* is a member of *clos_class_and* (E').

4.4. How to Evaluate Attributes

Just as with traditional attribute grammars we have to define how the attributes are evaluated for a given tree. For each tree $t \in Tree(G)$, instances of attributes are attached to the nodes in the following way:

if a node n is of node type N, then for each attribute $a \in A(E)$ where $N \in clos\_class(E)$, an instance of a is attached to node n. An attribute instance a of node n will be denoted by a **of** n.

Let n_0 be a node with node type N_0 and let $n_1, n_2, \cdots, n_m$ be its sons with node types $N_1, N_2, \cdots, N_m$ respectively. An attribute evaluation instruction

$$a \text{ of } n_k := f(a_1 \text{ of } n_{k_1}, a_2 \text{ of } n_{k_2}, \cdots, a_m \text{ of } n_{k_l})$$

is associated with the instances of a **of** n_k, which is extracted from the evaluation rule of a **of** $P_k.N_k$, where a_i **of** n_{k_i} represent the instances that are used. Because we now know which node types are applied at the different partnames with the tree rule of node type N_0, we can replace every **case** P_i **of** $\cdots E$: *expr* $\cdots$ **esac** in the evaluation rule of a **of** $P_k.N_k$ by *expr* where $N_i \in clos\_class(E)$. Thus the function f representing the applied evaluation rule, depends only on the attribute values attached to the nodes.

The task of an attribute evaluator is to compute the values of all attribute instances attached to the nodes of a tree by executing their associated evaluation instructions, in the same manner as for traditional attribute grammars.

5. Conclusions

In this paper we have demonstrated how to describe abstract program trees with a tree grammar and we have shown that it is possible to add attributes to the definition of that grammar.

A more detailed description of this approach can be found in [5]. A similar solution is presented in [9]. The attribution of abstract program trees is also discussed in [6] and [11].

In our research project on compiler-compilers we have implemented an attribute evaluator generator that given an APTG generates a PASCAL program to evaluate the attributes in any APT of that grammar. In this work we encountered a number of problems in producing efficient code. These implementation problems involved making non-trivial choices in the storage of the attributes associated with the classes in a tree. This work will be reported in a future paper.

Our current research includes the description of transformations of abstract program trees for the purpose of program optimization, and the generation of programs that can perform these transformations while keeping attribute values consistent. It should also be possible to generate transformations from one grammar to another, for example in the code generation phase of a compiler.

Acknowledgement. We are grateful to Albert Nymeyer who helped in the preparation of this paper.

References

1. H. Alblas: A characterization of attribute evaluation in passes, *Acta Inform.* **16** (1981) 427-464.
2. H. Alblas: Incremental simple multi-pass attribute evaluation, *Proc. NGI-SION Symposium* **4** (1986) 319-342.
3. F.L. DeRemer & R. Jullig: Tree-affix dendrogrammars for languages and compilers, *in:* Semantics-directed compiler generation, Lect. Notes Comp. Sci. **94** (1980) 300-319, Springer-Verlag, Berlin - Heidelberg - New York.
4. J. Engelfriet: Attribute Grammars: Attribute evaluation methods, *in:* B. Lorho (Ed.): *Methods and Tools for Compiler Construction*, Cambridge University Press, (1984) 103-138.
5. F.J. Faase: Een attribuut evaluator generator, Masters thesis, Dept. of Computer Science, University of Twente, Enschede, The Netherlands, (1986).
6. I. Glasner, U. Möncke & R. Wilhelm: OPTRAN, a language for the specification of program transformations, *Informatik-Fachberichte* **34,** (1980) 125-142, Springer-Verlag, Berlin - Heidelberg - New York.
7. D.E. Knuth: Semantics of context-free languages, *Math. Systems Theory* **2** (1968) 127-145, Correction in: *Math. System Theory* **5** (1971) 95-96.
8. J.R. Nestor, W.A. Wulf & D.A. Lamb: IDL-Interface Description Language, Technical Report, Dept. of Computer Science, Carnegie Mellon University (1981).
9. J.R. Nestor, B. Mishra, W.L. Scherlis & W.A. Wulf: Extensions to attribute grammars, Technical Report TL 83-36, Tartan Laboratories Inc., Pittsburgh (1983).
10. H. van Thienen: Automatic Generation of Abstract Grammars, Memorandum INF-87-19, Dept. of Computer Science, University of Twente, Enschede, The Netherlands, (1987).
11. R. Wilhelm: Computation and use of data flow information in optimizing compilers, *Acta Inform.* **12** (1979) 209-225.