# Programming Language Specification and Prototyping Using the MAX System

Arnd Poetzsch–Heffter[1]

Cornell University, 4112 Upson Hall, Ithaca, NY 14853
poetzsch@cs.cornell.edu

**Abstract**

The paper describes the MAX system, a tool for specification and prototyping of language processors. The MAX system is based on a first–order framework generalizing attribute grammar like frameworks. It allows to refer to syntax tree nodes and "distant" attribute occurrences. Attributes may have tree nodes as values, so that global relations between distant tree nodes can be expressed. This enables more compact and readable specifications for a wide class of complex problems. Within the presented framework, context conditions can be globally formulated by first–order predicate formulae.

The paper explains the fundamental semantical concepts of the system, shows its application to a small name analysis problem, and describes the main implementation issues. In particular, we present a powerful attribute evaluation algorithm that can handle attribute dependencies arising during evaluation time. Finally, we report on the experiences made by the system and compare it to other specification frameworks for language–based programming tool generation.

## 1 Introduction

To provide the user with powerful and flexible modularization and programming concepts, modern programming languages like Ada and C++ have very complex visibility, identification, and typing rules including import/export mechanisms, named scopes, renaming, overloading, and rules to solve ambiguity problems with (multiple) inheritance.

As these rules not only decide which programs satisfy the context conditions, but determine the semantics of a program — e.g. in C++, they influence the selection of member functions —, design and standardization efforts and compiler development based on formal specifications are even more important than for smaller languages. The following C++ fragment illustrates some identification subtleties arising from inheritance and nested classes:

```
(1)   struct  A  { int memb; };
(2)   struct  B : A  {                   //   B.2 is derived from A.1
(3)       struct  A  { int memb; };      //   A.3 is nested in B.2
(4)   };
(5)   int  foo(){
(6)       struct  C :  B, B::A  {...};   //   C is derived from B.2, A.3
(7)       struct  B :  A, C     {...};   //   B.7 is derived from A.1, C
(8)       B  obj;                        //   obj is of type B

(9)      obj . B::A::memb     ...        //   which memb is selected?
(10)  }
```

Is the selection in the last line ambiguous? And if not, which member of `obj` is selected? Should we take the base class A of B.7? Or should we look for a base class B of B.7 and then for a base class A of this B? Or should we look for a base class B::A of B.7? As reference manuals are mostly written in an adhoc fashion without systematic support, they often

[1]Supported by DFG grant Po 432/2-1.

leave many questions open; e.g. the C++ reference manual does not give a precise answer to the above problem. We developed a system to support modular language design that is based on a high–level specification language and provides implementations for realistic prototyping purposes.
Many of the conventional software tools supporting the language design and implementation process (cf. section 5) have the following disadvantages when faced with problems like the one above:

- All declaration and scope control information has to be collected in table data structures to pass it around in the syntax tree; as these symbol tables become rather complex data structures for big languages, symbol table based specifications are hard to read.
- Symbol table techniques tie up identification, typing (the type of a variable has to be recorded in the symbol table) and later tasks of language processing (e.g. storage allocation); i.e. they force specifications to give up the natural modularity based on the different aspects of semantic analysis.
- They provide a bad basis to prove language properties, mainly because of the complex symbol tables and because context conditions are usually mixed up with attribute definitions.

The MAX system, described in this paper, tries to overcome these disadvantages. The MAX system is based on a formal, first–order framework properly generalizing attribute grammar like frameworks. In addition to other features, the framework

- provides access to the syntax tree and to distant attribute occurrences in attribute definitions;
- allows attributes to have tree nodes as values;
- enables the formulation of context conditions by first–order predicate formulae;
- provides a simple and purely functional interface between semantic analysis and later tasks of language processing; e.g. it provides an excellent basis for recursively defined interpreters.

The notable aspect of the second feature is that it allows to define additional edges in the syntax tree, which is very useful to represent identification, type, and flow information (cf. figure 1).

**Paper Overview** The paper is organized as follows: In section 2, we explain how a specification according to our framework looks like. Section 3 sketches the semantics of specifications. Section 4 presents the main implementation concepts and experiences. In section 5, we relate the presented work to comparable systems and frameworks, in particular to the Cornell Synthesizer Generator and to higher order attribute grammars. Finally, we present conclusions and discuss topics for future research.

## 2 Specifying Semantic Analysis with MAX

This section explains the main parts of a small MAX specification to illustrate the key concepts of the system. A specification consists of the abstract syntax, attributes, functions, and context conditions. As an example, we use a tiny C++ subset featuring name analysis in the presence of inheritance – a simplified version of the name analysis problem illustrated in section 1 :

```
Program       (  DeclList  )
DeclList      *  Decl
Decl          =  ObjectDecl |  FunctDecl  |  ClassDecl
ObjectDecl    (  ClassId  Ident  )
FunctDecl     (  ClassId  Ident  /* NoParams */   DeclList  Exp  )
ClassDecl     (  Ident  ClassIdList  DeclList  )
ClassIdList   *  ClassId
ClassId       (  Ident  )

Exp           =  Object  |  Selection  |  ...
Object        (  Ident  )
Selection     (  Exp  Ident  )
```

A program of this subset is just a declaration list. Function declarations[2] are simplified by omitting parameters and allowing only an expression as body. A class declaration consists of a list of base class identifiers and a list of local members. A selection consists of an expression and a member identifier.

The example specification uses four attributes to describe identification: the attribute `env` (see below); the attribute `decl` yielding for each identifier the corresponding declaration, i.e. `decl` expresses the result of the identification; the attribute `type` yielding for each expression the corresponding class declaration; the attribute `accessible_membs` yielding for each class the accessible members of that class, i.e. all local members and all those members of the base classes that have a unique name. The syntax tree fragment in the following figure shows how `decl` and `type` can be understood as additional edges in the syntax tree. Here is an example of their use: To compute the declaration of the identifier `"m"` in the selection, take the type of the selected expression (the ○–attribute of Object)

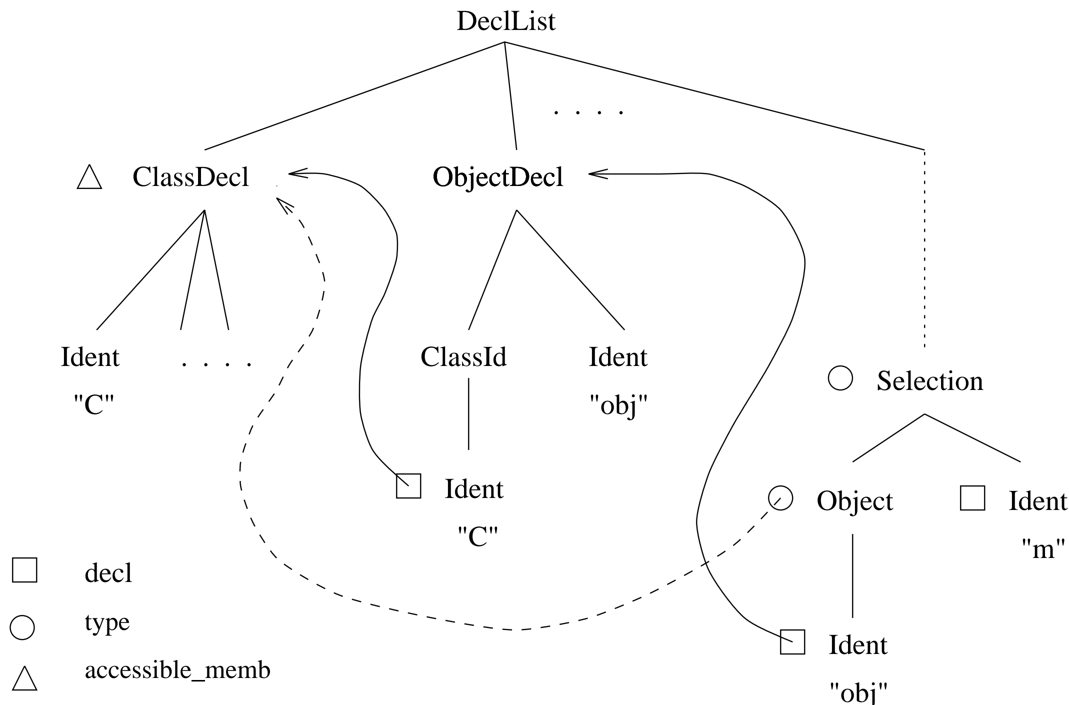


Figure 1: sample syntax tree

[2] In C++ terminology: function definitions.

and lookup the identifier in the attribute `accessible_membs` of that type. The following subsection explains how to specify such remote attribute accesses.

## 2.1 Attributes

An attribute in MAX is a special kind of function having exactly one *node valued* argument. Node sorts are denoted by the sort name as defined in the abstract syntax suffixed by `@`. To show how attribute definitions look like, we shortly discuss the definitions for `env`, `decl`, and `type`. The attribute `env` yields for each declaration the list of the locally valid declarations. In the body of an attribute definition, we use a pattern notation to refer to the context of a tree node. E.g. the definition of `env` reads as follows: The (static) environment of the first declaration `DCL` of a program is the list containing `DCL` as its only element; the environment

```
ATT  env( Decl@ DCL ) DeclNodeList :
  IF  Program@< < DCL, * > >          :  append( DCL, DeclNodeList() )
   |  FunctDecl@<_,_,< DCL, * >,_> FD :  append( DCL, env(FD) )
   |  ClassDecl@<_,_,< DCL, * > > CD  :  append( DCL, rest(env(CD)) )
   |  DeclList@<*, DCL1 ,DCL , *>     :  append( DCL, env(DCL1) )
  ELSE  nil()
```

of the first local declaration `DCL` in a function results from appending `DCL` to the environment of the function; likewise for classes; if the declaration `DCL` has a predecessor `DCL1` in its declaration list, then append the `DCL` to `env(DCL1)`. Notice that such conditional patterns must contain at least one bound identifier (`DCL` in the above example); all other identifiers are bound by the pattern. The definition of the attribute `decl` uses `env` directly when looking up class identifiers, and via the remote attribute access `fct_env` yielding the

```
ATT  decl( Ident@ IDN ) Decl@ :
  IF  Decl@< ClassId@<IDN>, * D :      lookup( term(IDN), rest(env(D)) )
   |  ClassDecl@<_,<*,<IDN>,*>,_> CD:  lookup( term(IDN), rest(env(CD)) )
   |  Object@< IDN >  OBJ :            lookup( term(IDN), fct_env(OBJ) )
   |  Selection@< E, IDN >:            lookup( term(IDN),
                                               accessible_membs(type(E)) )
  ELSE  nil()

ATT  type( Exp@ E ) ClassDecl@ :
  IF  Object@< IDN >     E :    decl( fstson(fstson(decl(IDN))) )
   |  Selection@<_,IDN > E :    decl( fstson(fstson(decl(IDN))) )
  ELSE  nil()
```

environment of the enclosing function declaration. The interesting part of this attribution is the identification of member identifiers in selections: The identifier is looked up in the `accessible_membs` attribute of the class declaration being the type of the selected expression (cf. figure 1). I.e. it uses the `type` attribute to refer to a distant tree node and the attribute occurrence of `accessible_membs` at that node.

In C++, type equality is declaration equality and not structural equality. Accordingly, the attribute `type` yields for each expression node the corresponding class declaration. E.g. to get the type of an object, take the class identifier in its declaration (the function `fstson` yields the first son of a tree node); as this identifier is as well subject to identification, take the declaration of it. Two aspects of the specification should be noticed: First, attributes can be mutually dependent; e.g. `decl` uses `type` and vice versa. Second, the dependencies between attribute occurrences may depend on attribute values; e.g. in `type`, the attribute `decl` is applied to a node that itself is determined using `decl`.

Compared to a classical attribute grammar, our framework has the following advantages: The static environment/symbol table mechanism is just a flat list (cf. [KW91]); there is no need for entering declaration information like the type of a variable or the accessible members for class declaration. This not only reduces the number of functions and the size of data structures a reader has to understand in order to understand the specification, but seperates as well the different tasks of static analysis that are otherwise tied up by the symbol table. In addition to this, we get a very nice and slim program representation as interface for later tasks of language processing, as space consuming tables are not needed (even the attribute `env` can be dropped). This is similar to what is done in the Ada intermediate language DIANA [GWEB87].

### 2.2 Functions

In contrast to semantic functions in classical AG frameworks, functions in MAX can access the syntax tree and may be mutual recursive with attributes. As small example consider the definition of the function `base_membs_rec` that recursively joins the accessible members of all base classes of a class declaration:

```
FCT  base_membs_rec( ClassId@ CID ) DeclNodeList:
  IF  ClassIdList@< *, <ID>CID >:      accessible_membs( decl(ID) )
   |  ClassIdList@<*,<ID>CID,CID1,*>:  join( accessible_membs(decl(ID)),
                                                  base_membs_rec(CID1) )
  ELSE  nil()
```

Notice that `base_membs_rec` could have also been specified as an attribute, because it has exactly one node valued parameter. As we will see, this would not have changed the semantics, but the implementation.

### 2.3 Context Conditions

The framework allows to formulate context conditions in a natural, and convenient way based on predicate logic. Especially during language design time, such high–level executable specifications of context conditions proved to be very useful. A context condition consists of a quantification part and a formula. The quantification is described using patterns. Here are two typical context conditions for our C++ subset:

```
CND  Ident@ IDN :   ! is_Decl@[ fath(IDN) ]  ->  decl(IDN) # nil()

CND  DeclList@<*,D1,*,D2,*> :  declid(D1) # declid(D2)
```

The first condition can be read as follows: For all identifier occurrences it must be true that if it is an applied occurrence (i.e. the father node is not a declaration), the declaration attribute must be defined (for the meaning of `nil` see section 3). The second condition requires that two declarations in a declaration list must have different identifiers. Error messages can be issued by attaching string expressions to the context conditions.

## 3 Semantics for MAX Specifications

This section explains the semantics of the MAX specification language. First, it defines the domain specified by an abstract syntax and a given syntax tree; then it defines the semantics of attributes, functions, and context conditions.

### 3.1 Occurrence Structures

This subsection introduces occurrence structures. Occurrence structures are used to model programs including all predefined functions and predicates of our framework. An occurrence structure consists of a set of order–sorted terms $T_{\mathcal{AS}}$, the set of occurrences of one given term $t \in T_{\mathcal{AS}}$, and the functions and predicates that express the relationship between terms and occurrences. The main advantages of occurrence structures compared to term algebras are as follows: They provide a global view to syntax trees allowing to formally express relations between distant parts of the tree, and they make available the first–order logical framework. To make this precise, let $\mathcal{AS}$ be an abstract syntax like that given at the beginning of section 2 consisting of

- tuple productions $tp_1, \ldots, tp_p$ where $tp_i$ has the form $\quad ts_i \;\; ( \; ts^i_1 \ldots ts^i_{m_i} \; )$
- list productions $lp_1, \ldots, lp_q$ where $lp_i$ has the form $\quad ls_i \;\; * \;\; ls^i$
- class productions $cp_1, \ldots, cp_r$ where $cp_i$ has the form $\quad cs_i \;\; = \;\; cs^i_1 \,|\ldots|\, cs^i_{n_i}$ .

The symbols denoting $ts_i$, $ls_j$, $cs_k$ where $i, j, k$ range over the appropriate range of natural numbers are called the sorts defined by $\mathcal{AS}$. Predefined are the sorts *Ident, Int, Bool, Char, String*. With each sort $s$, we associate a set of terms $T_s$. This is done as usual, except that the list constructors may have an arbitrary number of arguments. Thus, we get the following definition: Let $T_{Ident}$, $T_{Int}$, $T_{Bool}$, $T_{Char}$, $T_{String}$ be the sets associated with the predefined sorts, then the sets associated with $ts_i$, $ls_j$, $cs_k$ are defined to be the smallest sets fulfilling the following axioms:

- If $t_j \in T_{ts^i_j}$ for $j \in [1..m_i]$, then $ts_i(t_1, \ldots, t_{m_i}) \in T_{ts_i}$ .
- If $t_j \in T_{ls^i}$ for $j \in [0..k]$, where $k \in \mathrm{Nat}$, then $ls_i(t_1, \ldots, t_k) \in T_{ls_i}$ .
- If $t_j \in T_{cs^i_j}$ for a $j \in [1..n_i]$, then $t_j \in T_{cs_i}$ ; i.e. the sort defined by a class production is just the union of the sorts on the right–hand side.

The union of all sets associated with the sorts of $\mathcal{AS}$ is called the set of (order–sorted) terms of $\mathcal{AS}$, denoted by $T_{\mathcal{AS}}$. Now, let $t \in T_{\mathcal{AS}}$; we define the *set of occurrences Occ(t)* to be the multi–set of subterms of $t$ that contains for each occurrence of subterm $s$ in $t$ exactly one instance; we often call *Occ(t)* the nodes of $t$.

As an occurrence structure is a special kind of first–order structure, we first give the basic definitions for signatures and first–order structures: A *signature* (of a first–order structure) consists of two families of finite sets of symbols, the predicate symbols $(PRED_s)_{s \in \mathbb{N}}$ and the function symbols $(FUNC_s)_{s \in \mathbb{N}}$. A *first–order structure* S with signature $\Sigma$ is given by a set $U$ called the universe of S and two families of mappings $(\varphi_s)_{s \in \mathbb{N}}$ and $(\pi_s)_{s \in \mathbb{N}}$,

$$\varphi_s : FUNC_s \rightarrow \mathcal{F}(U^s, U) \quad \text{and} \quad \pi_s : PRED_s \rightarrow \mathcal{P}(U^s) \;\; ,$$

where $\mathcal{F}(U^s, U)$ denotes the functions from $U^s$ to $U$ and $\mathcal{P}(U^s)$ denotes the power–set of $U^s$. $\Sigma$–formulae and their interpretation in a $\Sigma$–structure are defined as usual. Details can be found e.g. in [End72].

The *occurrence structure* $\mathrm{OS}_{\mathcal{AS},t}$ given by an abstract syntax $\mathcal{AS}$ and a term $t \in T_{\mathcal{AS}}$ is a first–order structure $(\Sigma, U, (\varphi_s)_{s \in \mathbb{N}}, (\pi_s)_{s \in \mathbb{N}})$, where

- $\Sigma$ contains[3] the function symbols for the predefined functions $nil_{(0)}$, $root_{(0)}$, $fath_{(1)}$, $rbroth_{(1)}$, $fstson_{(1)}$, $append_{(2)}$, $first_{(1)}$, $rest_{(1)}$, $term_{(1)}$ and as those for the constructor functions $ts_{i(m_i)}$, $i \in [1..p]$, and the empty list constructors $ls_{i(0)}$, $i \in [1..q]$ (the

[3] The list of symbols is not complete w.r.t. MAX, but suffices to show the main features.

arity is denoted by the subscript) as well as the predicate symbols $=_{[2]}$ and $is^s_{[1]}$ for each sort $s$ or $s \equiv Node$.

- the universe $U$ is the disjoint union of $T_{\mathcal{AS}}$, $Occ(t)$, and the extra element *nil*; *nil* is used to make functions total that are usually defined only for a subset of $U$.
- $(\varphi_s)_{s \in \mathbb{N}}$ interpretes the functions as follows: $nil_{(0)}$ yields the extra element *nil*. $root_{(0)}$ yields the root node of $t$. $fath_{(1)}$, $rbroth_{(1)}$, $fstson_{(1)}$ yield the father, right brother, and first son node, if the argument is a node and its relative exists, otherwise they yield *nil*. $append_{(2)}$, $first_{(1)}$, $rest_{(1)}$ denote the ordinary list functions, made total by *nil*. $term_{(1)}$ yields for each node, i.e. for each subterm occurrence of $t$, the corresponding subterm; *nil* otherwise. Constructor functions are interpreted as usual and yield *nil*, if the arguments are not correctly sorted. Empty list constructors just yield the corresponding terms.
- $\pi_2$ interpretes $=_{[2]}$ as the equality on $U$ and $is^s_{[1]}$ as the sort check; e.g. $is^{Node}[\,n\,]$ is true, iff $n$ is a node.

On top of occurrence structures, we can define further sorts. A typical example is "`DeclNodeList * Decl@`", the sort of lists of declaration nodes used in the C++ example to represent environments. For an axiomatic definition of occurrence structures and enrichments by further data types see [PH91a].

### 3.2 Semantics of Attributes and User–Defined Functions

For the following let $\mathrm{OS}_{\mathcal{AS},t}$ be an occurrence structure and *AFL* be a finite list of possibly mutual recursive attribute and function definitions. In order to define a fixpoint semantics for *AFL* w.r.t. $\mathrm{OS}_{\mathcal{AS},t}$, we extend $\mathrm{OS}_{\mathcal{AS},t}$ by adding a bottom element $\perp$ to $U$, thereby introducing a flat domain structure on $U$. All functions and predicates are extended such that they are strict w.r.t. $\perp$. Notice that we cannot use *nil* for that purpose, because we defined the equality to be non–strict w.r.t. *nil*. The *nil* element is intended to denote "observable" failure situations like failing lookups in order to check their outcome later on.

To express correct sorting of parameters and function results (cf. section 2), we embed each body of an attribute and function in *AFL* into a conditional expression that first checks whether the arguments are correctly sorted, and if so and the result is correctly sorted, returns the result; otherwise it returns *nil*. As patterns in conditional expressions have to contain at least one bound identifier, we can translate them into boolean expressions checking whether the pattern matches the context of the bound node; if the pattern contains free identifiers, these identifiers are bound by let–expressions in the corresponding then–branch.

After these transformation steps, *AFL* is an ordinary system of recursive function definitions over a flat domain. Thus, we can define the semantics of *AFL* to be the least fixpoint of this system (see e.g. [Man74] chapter 5 for fixpoint theory of recursive functions).

### 3.3 Context Conditions

From a semantical point of view, the context conditions are just a convenient notation for first–order predicate formulae. E.g. the meaning of the first context condition in subsection 2.3 is expressed by the formula

$$\forall\, IDN :\ is^{Ident}[term(IDN)] \ \rightarrow\ (\,\neg is^{Decl}[term(fath(IDN))] \ \rightarrow\ \neg(\,decl(IDN) = nil()\,)\,)$$

A program $P$ is context correct, if all context conditions are valid in the occurrence structure of $P$ extended by attributes and functions. To be precise, "valid" means that for each assignment of non–bottom values to the variables bound in the quantification, the body of the formula is true; in particular, it must not be bottom.

# 4 Implementation Aspects and Experiences

## 4.1 Implementation

The overall structure of the implementation is as follows: In a first step, the system transforms the term representation of the syntax tree coming from the parser into an internal representation. Then attribute evaluation and context checking is performed. Finally, control is given back to the user program for further tasks of language processing. For that, s/he has access to the attributed syntax tree through the functional interface given by the specification, i.e. the tree walk functions, defined functions, and attributes. The first part of this subsection describes the internal program representation and context checking; the second part attribute evaluation; and the third part optimizations.

**Program Representation and Checking** In order to achieve efficient implementations for context checking and attribute evaluation, we cannot use a straightforward pointer implementation to represent programs, as we need for both tasks a fast access to all nodes of a given sort. In our implementation, we code each node by a positive integer, implementing the predefined treewalk functions (`fath`,..) by arrays. The coding of the nodes is performed according to the following rules where *maxnode* denotes the number of nodes in the considered syntax tree:

- the coding is a bijection from the set of nodes onto the interval [1,*maxnode*]
- all nodes of a sort *NS* defined by a tuple or list production are continuously coded, i.e. their codes form a continuous interval $[min_{NS}, max_{NS}]$
- all nodes of sort *NS* defined by a class production, should be contained in a continuous interval, if possible

To avoid technical overhead, we assume for the following that the third rule can be fulfilled for each class production. Thus, the nodes of each sort are contained in a continuous integer interval. This coding has very nice properties for the implementation of our framework: We can implement all attributes by compact arrays using the argument node as index; this enables trivial attribute allocation, and allows direct access to all attribute occurrences of a node. In addition to this, pattern matching for context conditions can be implemented very fast: Let *NS* be the outermost sort of the pattern; we only need a for–loop running from $min_{NS}$ to $max_{NS}$ checking for each node whether the top productions of the corresponding subtree match the pattern. For each of these matches, the variables in the pattern are bound to the corresponding nodes, and the body of the context condition is evaluated. If the check fails, the corresponding error message is issued.

The internal representation of a program is constructed in two passes over the term representation which may come from a parser, the MAX system itself or some other tools. The *first pass* counts for each sort defined by a tuple or list production how many nodes of this sort exist in the program. With the results of this pass, it is easy to compute the bounds of the intervals for the sorts. A counter for each sort of a tupel or list production is initialized to the lower bound of the corresponding intervall. During the *second pass*, the arrays for the predefined treewalk functions are computed by incrementing these counters whenever a subterm of this sort is visited.

**Attribute Evaluation** In contrast to attribute grammars, our framework allows attribute dependencies that arise during attribute evaluation, i.e., we can not even determine all attribute dependencies knowing the syntax tree. For example in the forth case of attribute

decl (see section 2.1), we access an attribute occurrence of accessible_membs at a node that is determined by the attribute value of type. We call such attribute dependencies *dynamic*. This subsection presents a straightforward implementation technique for attribute evaluation with dynamic dependencies; optimizations are discussed in the following paragraph.

The first step of attribute evaluation allocates for each attribute attr an array attr_array to store the attribute values. If *NS* is the parameter sort of the attribute, the index range of the corresponding array is $[min_{NS}, max_{NS}]$. Initialize these arrays to a special value undef. The attribute values are computed by recursive function procedures. To show how these procedures look like, let attr be an attribute with parameter sort parSort, result sort resSort, parameter name n, and body expression BODY. Then, we get the following procedure for attr (given in pseudo ANSI–C); to break up circular attribute dependencies, we use the special value seen:

```
resSort  attr  ( parSort n ){
   if(  attr_array[ n ] == undef  ){
        attr_array[ n ] =  seen ;
        attr_array[ n ] =  BODY ;
   } else if(  attr_array[ n ] == seen  ){
        exit(  circular attribute dependencies  );
   }
   return   attr_array[ n ] ;
}
```

Using this procedure at each call site in the translated specification, an attribute is only computed, if it is needed. As we made the observation that usually all attribute values are needed, we do not make use of this property, but compute all attribute values by simple for–loops over the index ranges of the attribute arrays. This guarantees that possible circular attribute dependencies are detected, and allows us to implement attribute calls in program parts using the attributed trees by simple array accesses.

The presented algorithm is essentially an adaption of the one proposed in [Jou84a]. Notice that it is different from that of Jourdan [Jou84b] and Katayama [Kat84]. The recursion here is not controled by the tree structure, but directly by the attribute dependencies.

**Optimization** If we compare the time needed to manage the control flow per evaluated attribute value, two reasons cause the MAX attribute evaluator to perform slower than a statically generated treewalk evaluator. The first reason is that a function call is needed for each attribute evaluation. The other reason stems from the evaluation of the conditional expressions in the body of an attribute definition. Whereas the treewalk evaluator "knows" his context in the tree, the first task in evaluating an attribute occurrence in MAX is to determine its context. This is usually done by conditional expressions (cf. the examples in section 2). By a careful implementation of conditional pattern matching, the resulting overhead can be drastically reduced. We obtained a reduction by the factor of four for complex attributions, if compared to a simple translation into if–then–elseif..–else statements. In our test suite, we needed in the average 1.8 switches or branches for the computation of one attribute value.

In order to discuss storage optimization, we have to review one of the basic design principles underlying MAX. In contrast to systems like GAG [KHZ82] or LINGUIST–86 [Far82], we were interested in a system that provides an attributed syntax tree for use in succeeding tasks of language processing, in particular for use by other tools. Therefore, we could not profit of techniques like those discussed in [FY91], but designed the framework in such a way that the number and complexity of attributes could be reduced. Only for attributes

that are declared "temporary", we deallocate the corresponding attribute arrays when all depending permanent attributes are computed. The dependencies between the attributes (not between the attribute *occurrences*) can be simply determined by analysing the recursive dependencies of the attribute definition; i.e. the storage optimization is performed on entire attributes and not on attribute occurrences.

Of course, we have to pay a certain price for the simple, but very powerful attribute evaluation technique: In the worst case, e.g. if we start attribute evaluation with an attribute occurrence that depends on every other attribute occurrence, we need stack space proportional to the depth of the attribute (occurrence) dependency graph. The interesting point here is that an evaluation strategy approximating the dependency graph suffices to avoid this problem. The difference between strategy and real dependencies is automatically handled by the stack mechanism. We say that an evaluation strategy approximates the dependency graph for a tree $T$ by a measure $M(T)$, iff the following holds whenever the strategy tries to evaluate an attribute occurrence *attr*: The number of unevaluated predecessors of *attr* in the attribute (occurrence) dependency graph is bound by $M(T)$. A typical example for such a measure is the depth of the tree times the maximum number of attributes of one node. In general, it can be hard to find and prove measures for given strategies, just as it is difficult to measure the needed stack space for recursive programs. In practice, we obtained very good results using the following technique.

In a first step, the attributes (not the attribute occurrences) are grouped in a sequence of *partitions* so that an attribute only depends on attributes in its own partition $P$ or in partitions being before $P$ in the sequence. These partitions are evaluated in turn, so that only the attributes of one partition have to be considered for the stack problem. Even in realistic applications, these partitions contain no more than 5–10 attributes. As evaluation strategy, we use a left to right tree traversal. This heuristic strategy reflects the textual order in program texts. Further improvements are sketched in the last section.

### 4.2 Experiences

Until now, MAX was used for two realistic size tasks and for a number of small up to mid–size application. The first realistic task was the bootstrap of the system itself: Currently, 74% of MAX is generated from a MAX specification; the rest consists of a yacc parser and a C–code generator. We like to shortly report on two experiences made during bootstrapping. The first is the importance of the high–level context condition facility for language design. Whereas context conditions are often treated as a disliked appendix to language design, in our framework they are an integral part designed together with abstract syntax and attribution; this leads to clearer languages and better formal specifications that can be nicely mixed up with informal descriptions for language documentation. Secondly, we learned about the practical importance of the simple functional interface between the system and C. As attributes and functions are translated into C–function procedures, it is very simple to switch from hand–written to generated code, or what is more important, to stepwise refine generated code by hand–written code to improve space and runtime aspects.

The second application — an analyser for a PASCAL subset — was chosen to compare MAX with conventional AG systems. We compared it to the CMUG system, a slim successor of MUG2 [GGMW82]. Some remarks are necessary to interprete the figures of the following table. The first column gives the specification length in lines. Even though we did not count the copy rules, the CMUG specification is considerable longer, because it has to contain descriptions of additional data structures like the symbol table and all the semantic functions working on it, because the code for context conditions and error handling in CMUG is more than three times as long as in MAX, and because the attribution is more complicated.

| | # lines | # attributes | # attr. occurrences | analysis time |
|---|---|---|---|---|
| MAX | 610 | 9 | 14825 | 1.72s |
| CMUG | 1470 | 17 | 72713 | 0.65s |

To compare the number of attributes, we counted in CMUG only different "semantical concepts", usually expressed by the same attribute name: E.g. the block nesting level is one attribute, even though it is attached to different nonterminals; but, attributes that occur as inherited and synthesized to one nonterminal are counted twice. The last two colums give the number of attribute occurrences and the analysis time for a typical[4] thousand line program containing all features of the PASCAL subset. With attribute optimizations like those discussed in [FY91] which are not performed in CMUG, it could be expected that the number of non–optimized attribute occurrences (i.e. occurrences allocated with nodes) is almost the same as in MAX. The interesting point here is that the advantages of the more flexible specification concepts of MAX are reflected by the optimization gain in classical AG systems. The showed time for program analysis contains scanning and parsing and is measured on a HP9700.

The system is as well very useful for other applications connected to programming language specification. E.g. it is rather simple to implement class browsers for object–oriented languages on the basis of MAX specifications. Another interesting application was the specification of the operational semantics for a small functional language: As the MAX system allows to seperate identification from interpretation, the runtime environments for functional languages with static binding are mappings from tree nodes (not from identifiers) to values/closures.

## 5 Comparison with Related Work

The MAX framework aims to close the gap in static semantics between frameworks that are primarily theory oriented (e.g. [Mos92]) and compiler generators competing with production quality compilers. In particular, it is related to recent works aiming at raising the specification level of attribute grammars, e.g. [KELP88], [KW91],[KW92]. Whereas those works essentially build new abstraction levels and modularity on top of attribute grammars, we generalized the basic framework. In the following, we compare MAX to three other approaches to static semantics specification based on attribute grammars. The comparison focusses on the specification framework.

**Cornell Synthesizer Generator** The CSG [RT89] generates language–based editors. It has its own applicative specification language SSL based on attribute grammars. The AG framework mainly was choosen because of the incremental evaluation properties. In order to ease specification and to save storage, SSL allows upward remote attribute access and references to attribute occurrences. Upward remote attribute access is just a special case of MAX's ability to inspect distant attribute occurrences. References to attribute occurrences in SSL can only be used in a very restricted way and do not have a clear semantics. In MAX, such references are obsolete, as distant attribute occurrences can be refered to through their node.

**Higher–Order Attribute Grammars** Like the MAX framework, higher–order attribute grammars [VSK89] are a proper generalization of classical attribute grammars. They allow

[4] In fact, we tested a suite of programs. But, as all programs showed almost the same behaviour, we decided to give only the figures for one program. Notice that this is in contrast to mesuring code quality of compiled programs.

to compute parts of the syntax tree during attribute evaluation and to use parts of the syntax tree as attribute values. Whereas the latter can be done in MAX too (by using the function `term`), the former cannot be done, if the computation of the "open" syntax tree parts is recursive as in macro expansion. If it is nonrecursive, such a behaviour can be simulated by defining several attribution phases like in attribute coupled grammars [GG84], i.e. using a term computed during attribution phase $i$ as input for attribution phase $i+1$ [5]. On the other hand, the central advantages of the MAX framework have to be treated in the HOAG similar to classical AG's; in particular, references to distant tree nodes can't be expressed.

**Door Attribute Grammars** In [Hed91], the amalgamation of attribute grammars with object–oriented techniques is described. In that approach, the attribute evaluation enriches the syntax tree by an object structure representing semantical information. This works is similar to the presented one in that the use of non-local attribute references is encouraged and supported. But the focus of the two works is different: Hedin's work focusses on the generation of incremental evaluators; as objects may be created during evaluation time, a special mechanism (the so–called "doors") is introduced to retain a kind of declarativity. The focus of our work was to provide a powerful system based on a simple, formal semantics that can be used to reason about program properties.

# 6 Conclusions and Future Research

## 6.1 Conclusions

We presented the MAX System, a tool for semantic analysis and similar tasks (like e.g. data flow analysis) that provides modular attribute specifications and declarative formulations of context conditions. As node valued attributes can be understood as additional edges in the syntax tree, the framework enables the specification of analysed programs as syntax graphs. Thus, table data structures at the interface between front– and back–ends can be avoided.

We described rather efficient implementation techniques for all parts of the system. The implementation provides an efficient functional access to analysed programs from C so that general language specifications could be used as basis for other language–based tools or as starting point for refining specified language front–ends.

## 6.2 Future Work

**Analysis Techniques** The powerful framework allows and encourages to violate the production locality property of attribute grammars. This makes the circularity test — in general — undecidable, renders optimization more difficult, and needs new techniques for incremental evaluation. We investigate analysis techniques from abstract interpretation to regain as much information as possible (cf. [Ros92]). A typical information of that kind would be e.g. "all occurrences of an attribute only use attribute occurrences left or upward in the syntax tree". With such or similar information, one can prove non–circularity, improve evaluation strategy, and characterize subclasses of the presented framework that allow for efficient incremental evaluation.

**Extensions** The presented specification language should be understood as a kernel for further very useful extensions. The most interesting extensions for us are the following:

[5] The MAX system can attribute any term no matter whether it comes from the parser or from another phase.

- The high–level visibility rule method described in [PH91b] and [PH92].
- Fixpoint definitions of attributes in the sense of Farrow ([Far86]) to specify data flow analysis; this is very promising, because we can use attributes to overlay the syntax tree with appropriate data flow graphs.
- Unification techniques to define attributes (cf. [Sne91]); this can be very useful to implement polymorphic type resolution and similar tasks; in contrast to usual specifications working with type assumption environments, our framework allows to specify the type relation between defined and used identifier occurrences directly by the link constructed during identification.

**Parallelism** Another interesting question for further research would be whether the greater modularity gained by the new attribution model could be exploited in parallel implementations. As the proposed framework considers a front–end as a step by step enrichment of the syntax tree, we would get a pipeline execution model having pipeline stages for each attribute partition.


### Acknowledgements

I would like to thank the anonymous referees for their detailed comments and my former colleague Achim Liebl with whom I developed a predecessor of the MAX system and from whom I have learned a lot about programming.


## References


[End72] H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[Far82] R. Farrow. LINGUIST–86: Yet another translator writing system based on attribute grammars. In *Proc. of the SIGPLAN '82 Symposium on Compiler Construction*, 1982.

[Far86] R. Farrow. Automatic generation of fixed–point–finding evaluatorsfor circular, but well–defined, attribute grammars. In *Proc. of the SIGPLAN '86 Symposium on Compiler Construction*, 1986.

[FY91] R. Farrow and D. Yellin. A comparison of storage optimizations in automatically–generated attribute evaluators. *Acta Informatica 23*, pages 85–98, 1991.

[GG84] H. Ganzinger and R. Giegerich. Attribute coupled grammars. In *Proc. of the SIGPLAN '84 Symposium on Compiler Construction*, pages 85–98, 1984.

[GGMW82] H. Ganzinger, R. Giegerich, U. Möncke, and R. Wilhelm. A truly generative semantics-directed compiler generator. In *Proc. of the SIGPLAN '82 Symposium on Compiler Construction*, 1982.

[GWEB87] G. Goos, W. A. Wulf, A. Evans, Jr., and K. J. Butler, editors. *DIANA An Intermediate Language for Ada*, volume 161 of *LNCS*. Springer–Verlag, 1987.

[Hed91] G. Hedin. Incremental static semantics analysis for object–oriented languages using door attribute grammars. In H. Alblas and B. Melichar, editors, *International Sommer School on Attribute Grammars, Applcations, and Systems*, pages 374–379, 1991. LNCS 545.

[Jou84a] M. Jourdan. An optimal-time recursive evaluator for attribute grammars. In M. Paul and B. Robinet, editors, *International Symposium on Programming*, pages 167–178, 1984. LNCS 167.

[Jou84b] M. Jourdan. Strongly non–circular attribute grammars and their recursive evaluation. In *Proc. of the SIGPLAN '84 Symposium on Compiler Construction*, pages 81–93, 1984.

[Kat84] T. Katayama. Translation of attribute grammars into procedures. *ACM Transactions on Programming Languages and Systems*, 6, 1984.

[KELP88] K. Koskimies, T Elomaa, T. Lehtonen, and J. Paakki. Tools/HLP84 report and user manual. Technical Report A–1988–2, Department of Computer Science, University of Helsinki, 1988.

[KHZ82] U. Kastens, B. Hutt, and E. Zimmermann. Gag: A practical compiler generator. *Lecture Notes in Computer Science 141*, 1982.

[KW91] U. Kastens and W. M. Waite. An abstract data type for name analysis. *Acta Informatica 28*, 1991.

[KW92] U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. Technical Report CU–CS–613–92, University of Colorado at Boulder, 1992.

[Man74] Z. Manna. *Mathematical Theory of Computation.* McGraw–Hill, 1974.

[Mos92] P. Mosses. *Action Semantics.* Cambridge University Press, 1992. "Tracts in Theoretical Computer Science".

[PH91a] A. Poetzsch-Heffter. *Context–Dependent Syntax of Programming Languages: A New Specification Method and its Application.* PhD thesis, Technische Universität München, 1991. (in german).

[PH91b] A. Poetzsch-Heffter. Logic–based specification of visibility rules. In J. Maluszynski and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, 1991. LNCS 528.

[PH92] A. Poetzsch-Heffter. Implementing high–level identification specfications. In P. Pfahler U. Kastens, editor, *Compiler Construction*, 1992. LNCS 641.

[Ros92] M. Rosendahl. Strictness analysis for attribute grammars. In M. Bruynoogh and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, 1992. LNCS 631.

[RT89] T. Reps and T. Teitelbaum. *The Syntheziser Generator Reference Manual.* Springer–Verlag, 1989. (3rd edition).

[Sne91] G. Snelting. The calculus of context relations. *Acta Informatica 28*, pages 411–445, 1991.

[VSK89] H. Vogt, S. Swierstra, and M. Kuiper. Higher order attribute grammars. In *ACM Conference on Progamming Language Design and Implementation*, 1989.