Ag - An Attribute Evaluator Generator

J. Grosch

GESELLSCHAFT FÜR MATHEMATIK
UND DATENVERARBEITUNG MBH

FORSCHUNGSSTELLE FÜR
PROGRAMMSTRUKTUREN
AN DER UNIVERSITÄT KARLSRUHE

Project

Compiler Generation

Ag - An Attribute Evaluator Generator

Josef Grosch

Aug. 3, 1992

Report No. 16

Gesellschaft für Mathematik und Datenverarbeitung mbH
Forschungsstelle an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1
D-7500 Karlsruhe

1. Introduction

Ag is an attribute evaluator generator [Gro89, Gro90b]. It processes ordered attribute grammars (OAGs) [Kas80], well-defined attribute grammars (WAGs) as well as higher order attribute grammars (HAGs) [VSK89]. It is oriented towards abstract syntax trees. Therefore the tree structure is fully known. The terminals and nonterminals may have arbitrary many attributes which can have any target language type. This includes tree-valued attributes. *ag* allows attributes local to rules and offers an extension mechanism which provides single inheritance as well as multiple inheritance for attributes and for attribute computations. It also allows the elimination of chain rules. The attribute computations are expressed in the target language and should be written in a functional style. It is possible to call external functions of separately compiled modules. Non-functional statements and side-effects are possible but require careful consideration. The syntax of the specification language is designed to support compact, modular, and readable documents. An attribute grammar can consist of several modules where the context-free grammar is specified only once. There are shorthand notations for copy rules and threaded attributes which allow the user to omit many trivial attribute computations. The generated evaluators are very efficient because they are directly coded using recursive procedures. Attribute storage for OAGs is optimized by implementing attributes as local variables and procedure parameters whenever their lifetime is contained within one visit.

2. Features

The following list tries to give a complete overview of the features of *ag*.

- processes ordered attribute grammars (OAGs)
- processes well-defined attribute grammars (WAGs)
- processes higher order attribute grammars (HAGs)
- allows tree-valued attributes
- allows to use subtrees as attribute values
- allows to create parts of the tree during evaluation time
- allows read access to non-local attributes
- operates on abstract syntax
- cooperates with the generator for abstract syntax trees *ast*
- the tree structure is fully known
- terminals and nonterminals may have attributes
- allows attributes of any type
- differentiates input and output attributes
- allows to eliminate chain rules
- allows attributes local to rules
- offers single and multiple inheritance
- attributes are denoted by unique selector names instead of nonterminals with subscripts
- is largely independent of the target language
- attribute computations are expressed in the target language
- attribute computations are written in a functional style
- attribute computations can call external functions
- non-functional statements and side-effects are possible

- allows to write compact, modular, and readable specifications
- AGs can consist of several modules
- the context-free grammar is specified only once
- checks an AG for completeness of the attribute computations
- checks for unused attributes
- checks an AG for the classes WAG, SNC, DNC, OAG, LAG, and SAG
- generates efficient evaluators
- the evaluators are directly coded using recursive procedures
- the implementation of the trees is efficient
- optimizes attribute storage (for OAG evaluators)
- attributes may be implemented as local variables in procedures and passed as parameters
- generates evaluators in C and Modula-2

3. Specification

The input of *ag* is an attribute grammar. The notation used is an extension of a specification for *ast* [Gro91a]. The *ast* formalism is used to describe the context-free grammar or the tree structure and to declare the attributes and their types. The extension describes the attribute computations. Such an extended specification is processed by both tools *ast* and *ag*. The first one generates a tree module and the second one an evaluator module. Additionally, the specification can be used to derive a scanner and parser which evaluate an S-attribution during parsing. This feature is described in a separate document [Gro91b]. The complete syntax of *ag* specifications is described in Appendix 1 using *ast*'s notation.

3.1. Context-Free Grammar

The context-free grammar on which an attribute grammar is based is described by an *ast* specification. The primary item of such a specification is a *node type* which corresponds to a nonterminal or a terminal as well as to grammar rules. The names of the node types correspond to the names of grammar rules. The grammar symbols on the right-hand side of rules are referred to as *children* in *ast*'s terminology.

3.2. Attribute Declarations

For every node type an arbitrary number of attributes of arbitrary types can be declared again using *ast*'s notation. The extension mechanism (single inheritance) as well as multiple inheritance are also available when using *ag*. The attribute properties *input*, *output*, *synthesized*, *inherited*, *threaded*, *ignore*, and *virtual* are meaningful for *ag* (see next section). The properties *synthesized* and *inherited* are often optional because they are determined automatically. The concept of views also works in combination with *ag*. The global property *ignore* is effective for attribute computations (see section 3.4), too. This allows for the activation of different sets of attribute computations.

3.3. Properties

The description of children and attributes can be refined by the association of so-called properties. These properties are expressed by the keywords listed in Table 1.

Table 1: Properties for Children and Attributes

long form	short form
`INPUT`	`IN`
`OUTPUT`	`OUT`
`SYNTHESIZED`	`SYN`
`INHERITED`	`INH`
`THREAD`	
`IGNORE`	
`VIRTUAL`	

The properties have the following meanings: *Input* attributes (or children) receive a value at node-creation time, whereas non-input attributes may receive their values at later times. *Output* attributes are supposed to hold a value at the end of a node's existence, whereas non-output attributes may become undefined or unnecessary earlier. *Synthesized* and *inherited* describe the kinds of attributes occurring in attribute grammars.

The property *thread* supports so-called threaded attributes: An attribute declaration [a THREAD] is equivalent to the declaration of a pair of attributes with the suffixes In and Out: [aIn INH] [aOut SYN]. In attribute computations, these attributes have to be accessed with their full name including the suffixes. Additionally, special default rules are applied to threaded attributes in order to yield the desired behaviour (see section 3.5.).

The property *ignore* instructs *ag* to disregard or omit an attribute or a child. It is useful in connection with the concept of views [Gro91a] (see section 4.). Attributes with the property *virtual* may depend on regular attributes or vice versa. However, virtual attributes neither take storage nor are the computations specified for them executed. Attributes with this property may be used to influence the evaluation order.

Properties are specified either locally or globally. Local properties are valid for one individual child or attribute. They are listed after the type of this item. Example:

```
Expr = [Value SYN] [Env: tEnv INH] .
```

Global properties are valid for all children and attributes defined in one or several modules. They are valid in addition to the local properties that might be specified. In order to describe global properties, a module may contain several property clauses which are written in the following form:

```
PROPERTY properties [ FOR module_names ]
```

The listed properties become valid for the given modules. If the FOR part is missing, the properties become valid for the module that contains the clause.

Example:

```
PROPERTY OUTPUT
PROPERTY SYN OUT FOR Eval2
```

3.4. Attribute Computations

For every node type, attribute computations (ACs) or actions may be specified. ACs can be placed everywhere within the list of attribute and child declarations and are enclosed in braces '{' '}'. ACs are written in the desired target language, using expressions, statements, or calls of external functions of separately compiled abstract data types. However, ACs have to be functional in order to allow *ag* the derivation of the dependencies among the attributes and the determination of an appropriate evaluation order. Side effects are possible as long as the user knows what he/she is doing. The ACs are copied unchecked to the generated evaluator module. Therefore syntax errors are detected by the compiler.

The ACs may contain *attribute denotations*. At a tree node, the attributes of this node and the attributes of the children are accessible. Attributes of a node or of the left-hand side of a rule are denoted just by their name. Attributes of a child or of the right-hand side of a rule are denoted by the child's selector name, a colon, and the attribute name. If an evaluator for WAGs is generated then it is possible to have read-access to non-local attributes:

```
LhsAttribute = Ident
RhsAttribute = Ident : Ident
RemAttribute = REMOTE Expression => Node_type : Ident
```

In a "remote" access the expression has to evaluate to a pointer to a tree node whose type is a subtype of 'Node_type'. The 'Ident' describes the desired attribute of this tree node.

Example:

```
Expr             = -> [Type] <
   Binary        = Lop: Expr Rop: Expr [Operator] .
> .

Type                            /*  left-hand side (node ) attribute */
Lop:Type                        /* right-hand side (child) attribute */
REMOTE addr => Expr:Type        /* remote attribute access           */
```

A name conflict occurs if the same identifier denotes an attribute as well as an other item. In this case the escape character '\' should precede the non-attribute item as otherwise *ag* would treat it as attribute. In general, the escape character '\' can be used within ACs to pass characters or tokens unchanged to the generated program module which otherwise *ag* would interpret erroneously.

The special value SELF is a pointer referring to the current tree node. It is of interest when higher order attribute grammars (HAGs) are used.

The following kinds of ACs are available (meta characters are '[', ']', and '|'):

```
Assignment = Attribute := Expression ;
Copy       = Attribute :- Attribute ;
AssignCode = Attributes := { Statement_sequence } ;
Check      = Conditions ;
Conditions = Condition | Condition Conditions
Condition  = [ CHECK Expression ] [ => Statement | => { Statement_sequence } ]
After      = Attributes AFTER  Attributes ;
Before     = Attributes BEFORE Attributes ;
```

The simplest form is the assignment of an arbitrary expression to an attribute. A copy rule behaves exactly like an assignment, with the restriction that only an attribute may be assigned instead of an arbitrary expression. The use of copy rules allows better optimizations in the generated attribute evaluator.

Example:

```
a   := 1;                       /* constant       */
a   :- c:b;                     /* copy rule      */
c:b :- a;                       /* copy rule      */
a   := c:a + 1;                 /* infix operator */
a   := f (c:b, c:a, 2);         /* function call  */
a   := f (c:b, g (c:a))) * 3;   /* nested calls   */
```

The *AssignCode* statement allows the computation of attributes using other statements than assignments. In this case the attribute dependencies can not be derived from the statements, automatically. The result attributes being computed have to be specified explicitly on the left-hand side of the symbol ':='. The right-hand side is a block containing arbitrary statements. This feature allows the description of conditional expressions and to compute several attributes at one time. Note, that for the target language C the symbol ':=' is used in assignments and before the block of the *AssignCode* statement. However, the block itself contains pure target language code and therefore the symbol '=' is the correct assignment operator there.

Example in C:

```
   a := { a = 1; };
   a := { sum (b, c, a); };
a, b := { p (a, b, c, d); };
a, b := { p (a, c); q (b, d); };
   a := { if (c:d) a = c:a; else a = c:b; };      /* or */
   a := c:d ? c:a : c:b;
```

Example in Modula-2:

```
   a := { a := 1; };
   a := { sum (b, c, a); };
a, b := { p (a, b, c, d); };
a, b := { p (a, c); q (b, d); };
   a := { IF c:d THEN a := c:a; ELSE a := c:b; END; };
```

The next form of ACs allows to check attribute values for certain conditions. It consists of a semicolon terminated list of checks where every check is composed out of an expression part and a statement part. If an expression evaluates to *true* then the next check in the list is considered. Otherwise, the statement part of this check is executed and the rest of the list is ignored.

Example:

```
CHECK a   # 0 => WriteString ("a is zero");
CHECK c:b > 0 => { Error ("some error text"); INC (ErrorCount); };

CHECK Object       != NULL => Error ("identifier not declared")
CHECK Object->Kind == kVar => Error ("variable required");
```

A missing expression part is equivalent to:

```
CHECK FALSE
```

This allows the execution of arbitrary statements during attribute evaluation. A missing statement part is equivalent to an empty statement.

In some cases it is desirable to add artificial dependencies between attributes. This feature can be used to turn AGs which are not OAG into OAG ones or to explicitly specify attribute evaluation order: The attributes on the left-hand side of AFTER are evaluated after the ones on the right-hand side or in other words they artificially depend on them. BEFORE works the other way round. Alternatively, the pseudo function DEP (x, y) can be used to describe artificial dependencies between attributes. It returns the value of its first argument x. This result depends on both arguments x and y.

Example:

```
a AFTER c:b;
a, c:a BEFORE c:b, d;
a:b := DEP (c, d:e);
```

3.5. Default Computations

If ACs are missing and they are not inherited via the extension mechanism (see next section), *ag* tries to insert copy rules as default ACs in the following ways:

- If an AC is missing for an right-hand side attribute *c:a* which is known to be inherited and not threaded, and if the left-hand side has an attribute with the same name *a*: c:a :- a;

- If an AC is missing for an left-hand side attribute *a* which is known to be synthesized and not threaded, and if there is a child *c* with an attribute named *a*: a :- c:a; If there are several children with attributes named *a* then the right-most child is selected.
- If an AC is missing for a threaded attribute *a* then the attribute is threaded through all children from left to right as shown in the example below. Or more precisely:
- If an AC is missing for a threaded attribute *a* and there is no child having a threaded attribute with the same name *a*: aOut :- aIn;
- If an AC is missing for a threaded attribute *a* of a child *c* and there is no other child having a threaded attribute named *a* to the left of it and there is a threaded attribute named *a* on the left-hand side: c:aIn :- aIn;
- If an AC is missing for a threaded attribute *a* of a child *c* and there is a child *b* having a threaded attribute named *a* to the left of it: c:aIn :- b:aOut;
- If an AC is missing for a threaded attribute *a* of the left-hand side and there is a child *c* having a threaded attribute named *a*: aOut :- c:aOut; If there are several children with threaded attributes named *a* then the right-most child is selected.

Example: automatically inserted default ACs

```
L       = [a INH] L1: L L2: L . /* L1:a :- a; L2:a :- a;             */

L       = [a SYN] L1: L L2: L . /* a :- L2:a;                        */

L       = [a THREAD] <          /* aOut :- aIn;                      */
   L0   = .                     /* aOut :- aIn;                      */
   L1   = L1: L L2: L .         /* L1:aIn :- aIn; L2:aIn :- L1:aOut; */
> .                             /* aOut :- L2:aOut;                  */
```

3.6. Extensions

The extension mechanism of *ast* fits together with the one of *ag*. First, like with *ast*, derived node types possess the attributes of their base type. Second, a derived type inherits the ACs of its base type. It may overwrite these by giving own ACs. *ag* tries to inherit ACs from a base type before the rules for default computations (see section 3.5) are applied.

Example:

```
Expr            = -> [Type]                      { Type := NoType; } <
   Binary       = Lop: Expr Rop: Expr [Operator] .
   Unary        = Expr [Operator]                { Type := Integer; } .
> .
```

The node type Binary inherits the computation of the attribute Type from the node type Expr whereas the node type Unary overwrites it.

3.7. Target Code

For both, the generated tree and evaluator modules, several sections containing *target code* may be specified. Target code is code written in the target language which is copied unchecked and unchanged to certain places in the generated modules. It has to be enclosed in braces '{' '}'. Balanced braces within the target code are allowed. Unbalanced braces have to be escaped by a preceding '\' character. In general the escape character '\' escapes everything within target code. Therefore especially the escape character itself has to be escaped. The keywords TREE and EVAL each introduce a set of sections with the meaning given below.

These two keywords may optionally be followed by an identifier that specifies the name of the generated moduile (compilation unit):

```
TREE [ Name ] ... EVAL [ Name ] ...
```

If several modules contain a name, the first one is chosen. If none of the modules contains a name, the default names *Tree* and *Eval* are used.

The meaning of the target code sections is as follows:

IMPORT:	Declarations to be included in the definition module at a place where IMPORT statements are legal.
EXPORT:	Declarations to be included in the definition module after the declaration of the tree type *tTREE* in the tree module. Behaves like IMPORT in the evaluator module.
GLOBAL:	Declarations to be included in the implementation module at global level.
LOCAL:	Declarations to be included in the procedures of the attribute evaluator.
BEGIN:	Statements to initialize the declared data structures.
CLOSE:	Statements to finalize the declared data structures.

3.8. Modules

The context-free grammar with attribute declarations and attribute computations may be followed by an arbitrary number of modules. The modules allow the combination of parts of the specification that logically belong together. A module consists of additional target code sections and specifications of node types with attribute declarations and attribute computations. The information given in the modules is merged in the following way: The target code sections are concatenated. If a node type has already been declared, its attribute declarations and attribute computations are added to the existing ones. If it has not been declared, it gets declared thus introducing a new node type. See Appendix 2 for an elaborated example. Additionally, the DECLARE section allows the definition of attributes for several node types at one time.

Example:

```
DECLARE
   Decls Stats Expr = -> [Level] [Env: tEnv] .
               Expr = -> [Type: tType] .
```

4. Several Attribute Evaluators

In some cases it might be desirable to use two or even more attribute evaluators that operate on one tree structure. These evaluators run one after the other and every one computes a disjoint subset of the attributes. There are two problems to solve:

First, a preceding evaluator may compute attributes which are used by a following evaluator. These attributes should have the property *output* in the first case and *input* in the latter case. Therefore we need means to switch the properties of attributes.

Second, the check for the completeness of an attribute grammar should be restricted to the attributes processed by the current evaluator. Otherwise those attributes that are to be computed by preceding evaluators are reported as not computed.

The solution is to generate all evaluators from one common attribute grammar and to use the concept of views. Each generation step uses a distinguished view that selects those parts of the complete information that are of interest and it adds properties such as *input* and *output*.

Example: Suppose, we want to run two attribute evaluators in sequence. In general, the complete specification might have the following parts:

module	contents	
G	context-free grammar + input attributes	for Eval 1 + Eval 2
I1	intermediate attributes	of Eval 1
O1	output attributes = input attributes	of Eval 1 of Eval 2
I2	intermediate attributes	of Eval 2
O2	output attributes	of Eval 2
C1	attribute computations	of Eval 1
C2	attribute computations	of Eval 2

We distribute the parts on separate modules. The final program consists of three compilation units:

Tree	tree module
Eval1	first evaluator
Eval2	second evaluator

The three compilation units can be generated with the following UNIX commands:

```
echo SELECT G I1 O1 I2 O2                          | cat - spec | ast -di
echo SELECT G I1 O1 C1 PROPERTY OUTPUT FOR O1 | cat - spec | ag  -DI
echo SELECT G O1 C2 O2 PROPERTY INPUT  FOR O1 | cat - spec | ag  -DI
```

The select clause choses distinct subsets of all information that represent complete attribute grammars. The property clause adds the global properties *input* and *output* to the module O1. We assume the following information to be present in the modules:

```
G    TREE Tree
C1   EVAL Eval1
C2   EVAL Eval2
O2   PROPERTY OUTPUT
```

This defines the names of the compilation units and the static global properties.

Note, if several attribute evaluators are used then currently the option for the optimization of attribute storage does not work. Therefore all attributes have to be stored in the tree. The reason is that the tree module has to be generated by a separate run of *ast* in order to yield a tree module with all attributes. However, this independent run of *ast* does not know about the optimizer results of the two *ag* runs.

5. Output

The output of *ag* is an evaluator module consisting of a definition part and an implementation part. The module exports 3 procedures: *EVAL* is the generated evaluator procedure, *BeginEVAL* and *CloseEVAL* are the procedures containing the BEGIN and CLOSE target code sections. The definition parts for C and Modula-2 are given below. The strings EVAL and TREE are replaced by the specified module names which default to *Eval* and *Tree*.

Definition part in C (header file):

```
extern void EVAL (tTREE t);
extern void BeginEVAL ();
extern void CloseEVAL ();
```

Definition part in Modula-2:

```
DEFINITION MODULE EVAL;

IMPORT TREE;

PROCEDURE EVAL (t: TREE.tTREE);
PROCEDURE BeginEVAL;
PROCEDURE CloseEVAL;

END EVAL.
```

6. Higher Order Attribute Grammars

ag is able to process *higher order attribute grammars* [VSK89] which are a reinvention of *generative attribute grammars* [Den84]. These grammars are characterized by the following features:

- Subtrees can be used as operands in expressions. They can appear on the right-hand side of attribute computations or as parameters of function calls.
- Non-input children or attributes of type tTREE may receive a value of type tTREE during attribute evaluation. This value can be an existing subtree or a dynamically created new tree.
- The attributes in dynamically created subtrees are evaluated like all other attributes. Attributes of this subtrees may depend on attributes of the already existing tree parts or vice versa.

The following example taken from [VSK89] computes faculty numbers. Together with the given main program it works as follows: the main program constructs an initial tree out of two nodes of types R and P1. Then the generated evaluator is called. It dynamically extends the tree by computing values for the non-input child F of the nodes P1 thus creating a tree of height n. All nodes have two attributes called n and r. The attribute n is inherited and holds the values to be multiplied. The attributes r is synthesized and computes the result. The node R serves as interface: the value of its n attribute is read in, the value of its r attribute receives the final result which is printed when attribute evaluation is finished. Note, that the node P2 inherits its computation for the attribute r from its base type F. The main program shows how an attribute evaluator can be called several times in one run of a program.

Example in C:

```
RULE

R       = F -> [n IN] [r OUT] { F:n := n; r := F:r; } .

F       =   -> [n] [r]        { r   := 1; } <

   P1   = -> F                { F   := n <= 1 ? mP2 () \: mP1 ();
                                F:n := n - 1;
                                r   := F:r * n; } .

   P2   = .
>.
```

Main program:

```
# include "Tree.h"
# include "Eval.h"

main ()
{
   tTree t; int i;

   do {
      scanf ("%d", & i);
      t = mR (mP1 (), i);
      Eval (t);
      printf ("%d\n", t->R.r);
   } while (t->R.n != 0);
   return 0;
}
```

Example in Modula-2:

```
RULE

R       = F -> [n IN] [r OUT] { F:n := n; r := F:r; } .

F       =   -> [n] [r]        { r   := 1; } <

   P1   = -> F                { F   := {IF n <= 1
                                        THEN F := Tree.mP2 ();
                                        ELSE F := Tree.mP1 ();
                                        END;};
                                F:n := n - 1;
                                r   := F:r * n; } .
   P2   = .
>.
```

Main program:

```
MODULE Main;

FROM StdIO      IMPORT ReadI, WriteI, WriteNl, CloseIO;
FROM Tree       IMPORT tTree, mR, mP1;
FROM Eval       IMPORT Eval;

VAR t   : tTree;

BEGIN
   REPEAT
      t := mR (mP1 (), ReadI ());
      Eval (t);
      WriteI (t^.R.r, 0); WriteNl;
   UNTIL t^.R.n = 0;
   CloseIO;
END Main.
```

7. Example

Appendix 2 contains an attribute grammar that specifies the semantic analysis for the example language *MiniLAX* [GrK88]. A complete specification of the MiniLAX compiler and a more detailed description of the attribute grammar can be found in [Gro90a] and [WGS89]. The attribute grammar is based on the abstract syntax of the language. It is divided into modules where each module describes the computation of one attribute. The first page of the specification describes the abstract syntax and the intrinsic attributes whose values are supplied by the scanner and parser. The attributes for semantic analysis are introduced in the individual modules. A separate module named *Conditions* contains all context checks for MiniLAX. The reporting of error messages is completely expressed in the target language. The source position is treated like any other attribute. This allows the combination of error messages with precise source positions.

8. When Things Go Wrong

This section gives advice and points out debugging facilities when problems with the attribute grammar or the generated evaluator occur. The information in this section is organized according to the time the problem can arise.

8.1. During Specification

An unalterable precondition for the use of *ag* is that the user has a basic understanding of attribute grammars. This knowledge should be available either through some kind of education or from appropriate text-books. Some hints for writing attribute grammars may help to make the generated evaluator work.

- Attribute computations should obey the functional stye, that means all arguments a computation depends upon have to be mentioned and no side-effects on global variables should occur. In other words, global variables should neither be read nor written during an attribute computation. Only the above rule allows the generator to determine a correct order for the evaluation of all attributes.
- If the user decides to compute attributes using global variables, special care has to be taken. The correct execution order should be validated for example by inspecting the visit sequences. If there are mistakes in the execution order, then BEFORE and AFTER clauses can be used to influence this order.
- Attributes with the property VIRTUAL may be used to influence the evaluation order, too. Virtual attributes may depend on regular attributes or vice versa. However, virtual attributes neither take storage nor are the computations specified for them executed.

8.2. During Generation

- Sometimes the extension or inheritance mechanism does not behave as expected. The inherited attribute computations are printed with the option 2 and should be checked if necessary.
- Sometimes the mechanism for default computations does not behave as expected. The copy rules inserted by default can be printed with the option 1 and should be checked if necessary.
- A serious problem occurs if cycles in the attribute dependencies are detected. This fact is reported as error together with the concerned node type and the attribute instances involved. The direct attribute dependencies specified in the attribute grammar and the closure operations applied by the tool result in a partial order between attribute instances. In order to arrive at a total order necessary to construct visit sequences the OAG algorithm adds additional dependencies between attributes. Sometimes these added dependencies introduce a cycle. Often this dependencies come in pairs. The explicit reversion of one dependency in a pair using an AFTER or BEFORE clause can solve this problem. This is possible when the attribute grammar is l-ordered or partitioned, but the tool did a bad guess in trying to produce a total order. However, if the attribute grammar is not l-ordered but DNC or SNC for example, a larger reformulation of the attribute grammar may be necessary.

  The following school example illustrates this phenomenon:

```
Z       = <
   s    = X Y { X:b := Y:d; Y:c := 1; Y:e := X:a; } .
   t    = X Y { X:b := Y:f; Y:c := X:a; Y:e := 2; } .
> .
X       = -> [a] [b] { a := 3; } .
Y       = -> [c] [d] [e] [f] { d := c; f := e; } .
```

It produces the following error messages:

```
  2,  4: Error        cycle in OAG

Cyclic Attributes and Artificially Introduced Dependencies

s       Reachable Nonterminal Explicit HasChildren HasActions

X:a     : X:b (1)
X:b     : (0)
Y:d     : Y:e (1)
Y:e     : (0)

  3,  4: Error        cycle in OAG

Cyclic Attributes and Artificially Introduced Dependencies

t       Reachable Nonterminal Explicit HasChildren HasActions

X:a     : X:b (1)
X:b     : (0)
Y:c     : (0)
Y:f     : Y:c (1)

  1,  1: Information  grammar is DNC
```

At both node types, s and t, there is cycle in the internal relation called OAG. The name of the node type and its properties are printed. It is followed by a list of the attribute instances on the cycle. For every attribute, the dependencies artificially introduced by the OAG algorithm are printed after the character ':'. The number in parentheses counts the artificial dependencies. In this example, there are always pairs of artificial dependencies. If we reverse the dependencies of X:a upon X:b by specifying that X:a should be computed before X:b, the conflict is solved. This can be done with one BEFORE clause as follows:

```
Z       = <
   s    = X Y { X:b := Y:d; Y:c := 1; Y:e := X:a; } .
   t    = X Y { X:b := Y:f; Y:c := X:a; Y:e := 2; } .
> .
X       = -> [a] [b] { a := 3; a BEFORE b; } .
Y       = -> [c] [d] [e] [f] { d := c; f := e; } .
```

- Sometimes the attribute dependencies occurring in cycles are not obvious. The *dialog system* of *ag* allows the interactive inquiry of detailed information about the attribute grammar and the problem. With options like V or P the visit sequences or the direct attribute dependencies for all node types can be printed. The output may be quite voluminous for realistic applications. The dialog system allows to obtain this information only for those node types which are of interest. The dialog system is started when *ag* is rerun with the same input and with the additional option J. The dialog system displays a menu and asks for commands which consist mostly of one letter. All commands are terminated by the RETURN key. Important commands are:

t <node type> or <node type>
: Selects a certain node type as current node type.

m
: Prints a summary of all input belonging to the current node type. This includes all attribute declarations and all attribute computations including inherited ones and copy rules added automatically as default computations. In modular attribute grammars this information can

be spread over several modules. Only this command provides a complete view of all information collected for a node type.

v Prints the visit sequence of a node type. For example the node type *Index* of the MiniLAX example in Appendix 2 has the following visit sequence:

```
Index   Reachable Nonterminal Explicit HasChildren HasAttributes HasActions

visit   parent 1. time to compute
        Env
compute Expr:Env
visit   Expr 1. time to compute
        Expr:Type
check   condition 19
compute Adr:Env
visit   Adr 1. time to compute
        Adr:Type
compute type
compute IsLegal
check   condition 18
compute Type
visit   parent 2. time to compute
        Level
        CodeSizeIn
        Co
compute Adr:Co
compute Expr:Level
compute Expr:Co
compute Adr:Level
compute Adr:CodeSizeIn
visit   Adr 2. time to compute
        Adr:CodeSizeOut
compute Expr:CodeSizeIn
visit   Expr 2. time to compute
        Expr:CodeSizeOut
compute CodeSizeOut
visit   parent
```

The first line gives the name of the node type and its properties. In this case, Index is an explicitly declared nonterminal which can be reached from the start symbol. It is a rule having children, attributes, and attribute computations (actions). The first two lines of the visit sequence might erroneously be interpreted as to do nothing and to return to the parent node, immediately. However, the correct interpretation is, that the Index node is visited the first time from the parent node and the inherited attribute *Env* has been computed in the context of the parent node. Therefore the first operation carried out at an Index node is the computation of the attribute *Expr:Env*. The rest of the visit sequence should be self explanatory.

f Finds and prints a path of dependencies between two attribute instances. These attribute instances can be selected with the commands a and b. The printed path can be considered as one path in the many possible trees. The path is printed as a sequence of lines. Every line contains the name of a node type and two attribute instances. It represents a direct dependency. The nodes of two neighbouring lines can be neighbours in the tree. The concatenation of the direct dependencies represents the desired (usually non-direct) dependency information.

Example: Suppose for MiniLAX (Appendix 2) we want know how in the rule *Proc* the attribute *CodeSizeOut* depends upon the attribute *Env*. The following is a slightly beautified listing showing how to answer this question.

```
ag -J minilax.ag

... the dialog system displays its menu here ...

? t Proc
? a CodeSizeOut
? b Stats:Env
? ?
node type: Proc, a: CodeSizeOut = 8, b: Stats:Env = 43

? f
Proc    CodeSizeOut      Stats:Env

Proc    CodeSizeOut      Next:CodeSizeOut
Decls   CodeSizeOut      CodeSizeIn
Proc    Next:CodeSizeIn  Decls:CodeSizeOut
NoDecl  CodeSizeOut      CodeSizeIn
Proc    Decls:CodeSizeIn Stats:CodeSizeOut
Stat    CodeSizeOut      Next:CodeSizeOut
Assign  CodeSizeOut      Next:CodeSizeOut
Call    CodeSizeOut      Next:CodeSizeOut
Stats   CodeSizeOut      CodeSizeIn
Call    Next:CodeSizeIn  Actuals:CodeSizeOut
Actual  CodeSizeOut      Next:CodeSizeOut
Actuals CodeSizeOut      CodeSizeIn
Actual  Next:CodeSizeIn  Expr:Co
Actual  Expr:Co          Formals
Call    Actuals:Formals  Object
Call    Object           Env
Assign  Next:Env         Env
Stat    Next:Env         Env
? x
```

C and c

In case of problems with cycles because of attribute dependencies introduced artificially, those dependencies can be reported. The command C displays all dependencies of a node type added artificially. The command c restricts this information to the attribute instances on the cycle. The following dialog illustrates the analysis of the school example from above:

```
ag -J dnc2

... the dialog system displays its menu here ...

? t s
? c
s       Reachable Nonterminal Explicit HasChildren HasActions

X:a     : X:b (1)
X:b     : (0)
Y:d     : Y:e (1)
Y:e     : (0)

? C
s       Reachable Nonterminal Explicit HasChildren HasActions

X       : (0)
Y       : (0)
X:a     : X:b (1)
X:b     : (0)
Y:c     : (0)
Y:d     : Y:e (1)
Y:e     : (0)
Y:f     : Y:c (1)
```

```
? x
```

8.3. During Compilation

Syntax errors in the attribute computations are not detected by the *ag* tool but by the following compilation. In case of Modula-2 as implementation language, the errors are reported with respect to the line number of the generated evaluator module instead of the specification. However, the generated evaluator contains comments referring to the line numbers in the specification. In case of C as implementation language, the errors are reported with respect to the line number of the specification. In general, the generated source code is relatively readable. It consists of case statements where the case labels represent node types. The names of the attributes are prefixed by access pathes. They allow the identification of individual attribute computations which are more or less unchanged.

In a few cases, *ag* can not distinguish conditional expressions (in C) or case labels (in C and Modula-2) from attribute denotations.

Example:

```
a := b ? c : d;

a := { switch (b) { case k : a = 1; }; };

a := { CASE b OF k : a := 1; END; };
```

In these ACs, the sequences c:d and k:a would erroneously be interpreted as attribute denotations. This mistake reappears later as syntax error in the generated evaluator. The problem is caused by the ambiguous use of the character ':'. Escaping the colon with the character '\' solves the problem.

Example:

```
a := b ? c \: d;

a := { switch (b) { case k \: a = 1; }; };

a := { CASE b OF k \: a := 1; END; };
```

8.4. During Execution

When things go wrong, an attribute evaluator may either yield wrong results or crash with a runtime error. In both cases, the trace facility of *ag* can offer significant help to locate the problem. When the evaluator is regenerated with the additional option T, it prints a trace during execution on standard output. Every action of the evaluator prints a line in one of the following formats:

```
<node type> e <attribute instance> = <value>
<node type> c <number>             = T or F
<node type> v <child name> <n>
<node type> v parent
```

Every action starts with the type of the current node.

The letter e (evaluate) indicates the computation of the mentioned attribute instance. The resulting attribute value is printed using the type specific write macro also used by the ASCII writer of *ast*.

The letter c (check) indicates the execution of a CHECK statement which are internally distinguished by numbers. The value of the boolean expression is printed (T = TRUE, F = FALSE).

The letter v (visit) indicates either the n-th visit to a child node or to the parent node.

The trace output is usually voluminous. It should be requested for small inputs, only. It also increases the size of the evaluator by approximately a factor of two. In order to reduce the output volume or the size of the evaluator module, the trace can be restricted to e (and c) or v actions or to e actions without printing of the values. This is possible with the options X, Y, and Z (see section 9). If the option U is used, the trace is restricted to a subset of all node types. The names of the desired node types are read from a file named *TraceTab*. This is a text file containing a name of a node type in every line. The trace property is extended or inherited along the extension hierarchy. Therefore it suffices to enumerate base types, only.

If the example in section 5 is processed with the command 'ag -cdimwDI0T hag' the trace produced for the input 2 is as follows:

```
R                e F:n              = 2
R                v F 1
P1               e F                =  P1
  F                =  NoTree
P1               e F:n              = 1
P1               v F 1
P1               e F                =  P2
P1               e F:n              = 0
P1               v F 1
P2               e r                = 1
P2               v parent
P1               e r                = 1
P1               v parent
P1               e r                = 2
P1               v parent
R                e r                = 2
R                v parent
```

9. Future Work

The following features should be added to *ag*:

- Optimization of attribute storage should be improved turning attributes into global variables.
- So-called *demand* attributes could be added, which are not stored in the tree but evaluated every time they are needed.
- So-called *circular* attribute grammars should be supported.
- A mechanism for turning non-OAGs into OAG ones like the *arrange orderly* feature of the GAG system [KHZ82] should be added.
- Integration of automatically determined evaluation sequences with user specified ones would be desirable.

10. Usage

NAME

ag – generator for attribute evaluators

SYNOPSIS

ag [-options] [-l dir] [file]

DESCRIPTION

Ag generates a program module to evaluate an attribute computation specified by an attribute grammar. A typical application is the semantic analysis phase in a compiler. The input *file* contains an attribute grammar which describes the structure of all possible trees, the attributes, and the attribute computations. *Ag* checks whether the attribute grammar is *ordered* (OAG) and generates an evaluator consisting out of recursive procedures. If *file* is omitted the specification is read from standard input.

OPTIONS

A	generate all, same as -DI (default)
D	generate definition module
I	generate implementation module
L	generate a (lazy) evaluator for WAG (default OAG)
W	suppress warnings
B	allow missing attribute computations in extended node types
V	print visit sequences
M	print summary of all node types (rules) from source
P	print dependency relations DP
S	print dependency relations SNC
N	print dependency relations DNC
O	print dependency relations OAG
G	print attribute instances sorted by declaration
E	print attribute instances sorted by evaluation order
C	print dependencies introduced for total order (completion)
T	generate evaluator with trace output (all actions, T = XZ)
U	trace only node types specified in file TraceTab
X	trace attribute evaluation actions with values
Y	trace attribute evaluation actions without values
Z	trace visit actions
J	start dialog system
Q	browse internal data structure
0	optimize attribute storage
1	print inserted copy rules
2	print inherited attribute computation rules
3	print attribute storage assignment
6	generate # line directives
7	touch output files only if necessary
8	report storage consumption
9	generate code to measure stack size
c	generate C code (default is Modula-2)
H	print help information for evaluator module
ldir	*dir* is the directory where ag finds its table files

FILES

if output is in C:

<module>.h	specification of the generated evaluator module
<module>.c	body of the generated evaluator module
yy<module>.w	macro definitions

if output is in Modula-2:

<module>.md	definition module of the generated evaluator module
<module>.mi	implementation module of the generated evaluator module

SEE ALSO

J. Grosch: "Ast - A Generator for Abstract Syntax Trees", GMD Forschungsstelle an der Universitaet Karlsruhe, Compiler Generation Report No. 15

J. Grosch: "Ag - An Attribute Evaluator Generator", GMD Forschungsstelle an der Universitaet Karlsruhe, Compiler Generation Report No. 16

J. Grosch: "Object-Oriented Attribute Grammars", in: A. E. Harmanci, E. Gelenbe (Eds.): Proceedings of the Fifth International Symposium on Computer and Information Sciences (ISCIS V), Cappadocia, Nevsehir, Turkey, 807-816, Oct .1990

References

[Den84] P. Dencker, Generative attributierte Grammatiken, Dissertation, Universität Karlsruhe, 1984.

[GrK88] J. Grosch and E. Klein, Übersetzerbau-Praktikum, Compiler Generation Report No. 9, GMD Forschungsstelle an der Universität Karlsruhe, June 1988.

[Gro89] J. Grosch, Ag - An Attribute Evaluator Generator, Compiler Generation Report No. 16, GMD Forschungsstelle an der Universität Karlsruhe, Aug. 1989.

[Gro90a] J. Grosch, Specification of a Minilax Interpreter, Compiler Generation Report No. 22, GMD Forschungsstelle an der Universität Karlsruhe, Mar. 1990.

[Gro90b] J. Grosch, Object-Oriented Attribute Grammars, in *Proceedings of the Fifth International Symposium on Computer and Information Sciences (ISCIS V)*, A. E. Harmanci and E. Gelenbe (ed.), Cappadocia, Nevsehir, Turkey, Oct. 1990, 807-816.

[Gro91a] J. Grosch, Ast - A Generator for Abstract Syntax Trees, Compiler Generation Report No. 15, GMD Forschungsstelle an der Universität Karlsruhe, Sep. 1991.

[Gro91b] J. Grosch, Preprocessors, Compiler Generation Report No. 24, GMD Forschungsstelle an der Universität Karlsruhe, Feb. 1991.

[Kas80] U. Kastens, Ordered Attribute Grammars, *Acta Inf. 13*, 3 (1980), 229-256.

[KHZ82] U. Kastens, B. Hutt and E. Zimmermann, *GAG: A Practical Compiler Generator*, Springer Verlag, Heidelberg, 1982.

[VSK89] H. H. Vogt, S. D. Swierstra and M. F. Kuiper, Higher Order Attribute Grammars, *SIGPLAN Notices 24*, 7 (July 1989), 131-145.

[WGS89] W. M. Waite, J. Grosch and F. W. Schröer, Three Compiler Specifications, GMD-Studie Nr. 166, GMD Forschungsstelle an der Universität Karlsruhe, Aug. 1989.

Appendix 1: Syntax of the Specification Language

```
RULE

/* parser grammar */

Specification    = <
                 = ScannerName ParserCodes TreeCodes EvalCodes PrecPart
                      PropPart DeclPart RulePart Modules .
                 = 'MODULE' Name ScannerName ParserCodes TreeCodes EvalCodes PrecPart
                      PropPart DeclPart RulePart 'END' Name Modules .
> .
ScannerName      = <
                 = .
                 = 'SCANNER' .
                 = 'SCANNER' Name .
> .
ParserCodes      = <
                 = .
                 = 'PARSER'      Codes .
                 = 'PARSER' Name Codes .
> .
TreeCodes        = <
                 =             SubUnit .
                 = 'TREE'      SubUnit Codes .
                 = 'TREE' Name SubUnit Codes .
> .
EvalCodes        = <
                 = .
                 = 'EVAL'      Codes .
                 = 'EVAL' Name Codes .
> .
Codes            = <
                 = .
                 = Codes 'IMPORT' tTargetCode .
                 = Codes 'EXPORT' tTargetCode .
                 = Codes 'GLOBAL' tTargetCode .
                 = Codes 'LOCAL'  tTargetCode .
                 = Codes 'BEGIN'  tTargetCode .
                 = Codes 'CLOSE'  tTargetCode .
> .
SubUnit          = <
                 = .
                 = SubUnit 'SUBUNIT' Name .
                 = SubUnit 'VIEW'    Name .
> .
PrecPart         = <
                 = .
                 = 'PREC' Precs .
> .
Precs            = <
                 = .
                 = Precs 'LEFT'  Names .
                 = Precs 'RIGHT' Names .
                 = Precs 'NONE'  Names .
> .
PropPart         = Props .

Props            = <
                 =
                 = Props 'PROPERTY' Properties
                 = Props 'PROPERTY' Properties 'FOR' Names
                 = Props 'SELECT' Names
> .
DeclPart         = <
                 = .
                 = 'DECLARE' Decls .
> .
Decls            = <
                 = .
   MoreNonterms  = Decls Names '=' AttrDecls '.' .
   MoreTerminals = Decls Names ':' AttrDecls '.' .
> .
Names            = <
                 = .
                 = Names Name .
```

```
                  = Names ',' .
> .
RulePart          = <
                  = .
                  = 'RULE' Types .
> .
Types             = <
                  = .
   Nonterminal0   = Types             BaseTypes '='      AttrDecls Prec Extensions '.' .
   Nonterminal1   = Types Name        BaseTypes '='      AttrDecls Prec Extensions '.' .
   Terminal1      = Types Name        BaseTypes ':' Code AttrDecls Prec Extensions '.' .
   Terminal2      = Types Name tIdent BaseTypes ':' Code AttrDecls Prec Extensions '.' .
   Abstract       = Types Name        BaseTypes ':='     AttrDecls Prec Extensions '.' .
> .
BaseTypes         = <
                  = .
                  = '<-' Names .
> .
Code              = <
                  = .
                  = tInteger .
> .
Prec              = <
                  = .
                  = 'PREC' Name .
> .
Extensions        = <
                  = .
                  = '<' Types '>' .
> .
AttrDecls         = <
                  = .
   ChildSelct     = AttrDecls     Name ':' Name Properties .
   ChildNoSelct   = AttrDecls              Name Properties .
   AttrTyped      = AttrDecls '[' Name ':' Name Properties ']' .
   AttrInteger    = AttrDecls '[' Name          Properties ']' .
   Block          = AttrDecls '{' Actions '}' .
> .
Properties        = <
                  = .
                  = Properties 'INPUT' .
                  = Properties 'OUTPUT' .
                  = Properties 'SYNTHESIZED' .
                  = Properties 'INHERITED' .
                  = Properties 'THREAD' .
                  = Properties 'IGNORE' .
                  = Properties 'VIRTUAL' .
                  = Properties 'REVERSE' .
> .
Actions           = <
                  = .
   Assign         = Actions Attributes ':=' tExpression ';' .
   Copy           = Actions Attribute  ':-' Attribute   ';' .
   AssignCode     = Actions Attributes ':=' '{' tStatement_Sequence '}' ';' .
   After          = Actions Attributes 'AFTER'  Attributes ';' .
   Before         = Actions Attributes 'BEFORE' Attributes ';' .
   Condition      = Actions Checks ';' .
> .
Attributes        = <
                  = .
   LhsAttribute   = Attributes tIdent .
   RhsAttribute   = Attributes tIdent ':' tIdent .
   RemAttribute   = Attributes 'REMOTE' tExpression '=>' tIdent ':' tIdent .
> .
Modules           = <
                  = .
                  = Modules 'MODULE' Name ParserCodes TreeCodes EvalCodes
                       PropPart DeclPart RulePart 'END' Name .
> .
Checks            = <
                  = Check .
                  = Check Checks .
> .
Check             = <
                  = 'CHECK' tExpression Statement .
                  = 'CHECK' tExpression           .
```

```
                  =                     Statement .
> .
Statement         = <
                  = '=>' tStatement .
                  = '=>' '{' tStatement_Sequence '}' .
> .
Name              = <
                  = tIdent .
                  = tString .
> .

/* lexical grammar */

tIdent            : <
                  = Letter .
                  = tIdent Letter .
                  = tIdent Digit .
                  = tIdent '_' .
> .
tInteger          : <
                  = Digit .
                  = tInteger Digit .
> .
tString           : <
                  = "'" Characters "'" .
                  = '"' Characters '"' .
> .
tTargetCode       : '{' Characters '}' .

Comment           : '/*' Characters '*/' .

Characters        : <
                  = .
                  = Characters Character .
> .

tExpression       : .    /* target language expression          */

tStatement        : .    /* target language statement           */

tStatement_Sequence: .   /* target language statement sequence  */
```

Appendix 2: Attribute Grammar for MiniLAX

```
MODULE AbstractSyntax /* ------------------------------------------ */

TREE IMPORT  {
FROM Idents     IMPORT tIdent;
FROM Positions  IMPORT tPosition;
}
GLOBAL  {
FROM Idents     IMPORT tIdent;
FROM Positions  IMPORT tPosition;
}
EVAL Semantics

PROPERTY INPUT

RULE

MiniLAX         = Proc .
Decls           = <
   NoDecl       = .
   Decl         = Next: Decls REV [Ident: tIdent] [Pos: tPosition] <
      Var       = Type .
      Proc      = Formals Decls Stats .
   >.
>.
Formals         = <
   NoFormal     = .
   Formal       = Next: Formals REV [Ident: tIdent] [Pos: tPosition] Type .
>.
Type            = <
   Integer      = .
   Real         = .
   Boolean      = .
   Array        = Type OUT            [Lwb] [Upb] [Pos: tPosition] .
   Ref          = Type OUT .
   NoType       = .
   ErrorType    = .
>.
Stats           = <
   NoStat       = .
   Stat         = Next: Stats REV <
      Assign    = Adr Expr            [Pos: tPosition] .
      Call      = Actuals             [Ident: tIdent] [Pos: tPosition] .
      If        = Expr Then: Stats Else: Stats .
      While     = Expr Stats .
      Read      = Adr .
      Write     = Expr .
   >.
>.
Actuals         = <
   NoActual     =                     [Pos: tPosition OUT] .
   Actual       = Next: Actuals REV Expr .
>.
Expr            =                     [Pos: tPosition] <
   Binary       = Lop: Expr Rop: Expr [Operator: SHORTCARD] .
   Unary        = Expr                [Operator: SHORTCARD] .
   IntConst     =                     [Value            OUT] .
   RealConst    =                     [Value: REAL      OUT] .
   BoolConst    =                     [Value: BOOLEAN   OUT] .
   Adr          = <
      Index     = Adr Expr .
      Ident     =                     [Ident: tIdent] .
   >.
>.
Coercions       = <
   NoCoercion   = .
   Coercion     = Next: Coercions OUT <
      Content   = .             /* fetch contents of location    */
      IntToReal = .             /* convert integer value to real */
   >.
>.

END AbstractSyntax
```

```
MODULE Output /* ------------------------------------------------- */

PROPERTY OUTPUT

DECLARE
   Formals Decls        = [Decls: tObjects THREAD] .
   Call Ident           = [Object: tObjects] [level: SHORTINT] .
   If While             = [Label1] [Label2] .
   Read Write Binary    = [TypeCode: SHORTCARD] .
   Expr                 = Type Co: Coercions .
   Index                = type: Type .

END Output

MODULE Decls /* -------------------------------------------------- */

EVAL GLOBAL { FROM Definitions IMPORT mNoObject, mProc, mVar, mProc2, mVar2, Identify; }

DECLARE Formal Decl     = [Object: tvoid OUT] .

RULE

MiniLAX = { Proc:       DeclsIn := nNoObject                         ; } .
Decl    = { Next:       DeclsIn := nNoObject                         ;
                        DeclsOut:= Next:        DeclsOut             ;
                        Object  := {}                                ; } .
Proc    = { Next:       DeclsIn := mProc (DeclsIn, Ident, Formals)   ;
                        Object  := {mProc2 (Next:DeclsIn, Level, CodeSizeIn,
                                   Formals:DataSizeOut, Decls:DataSizeOut);};
            Formals:    DeclsIn := nNoObject                         ; } .
Var     = { Next:       DeclsIn := mVar (DeclsIn, Ident, Type)       ;
                        Object  := {mVar2 (Next:DeclsIn, Level, DataSizeIn);}; } .
Formal  = { Next:       DeclsIn := mVar (DeclsIn, Ident, Type)       ;
                        Object  := {mVar2 (Next:DeclsIn, Level, DataSizeIn);}; } .
Call    = {             Object  := Identify (Ident, Env)             ; } .
Ident   = {             Object  := Identify (Ident, Env)             ; } .

END Decls

MODULE Formals /* ------------------------------------------------- */

EVAL GLOBAL     {
FROM Definitions IMPORT tObjects, GetFormals;
FROM Tree       IMPORT Formal;
FROM Types      IMPORT CheckParams;
}

DECLARE Actuals = [Formals: MyTree] .

RULE

Call    = { Actuals:    Formals := GetFormals (Object)               ;
            => { CheckParams (Actuals, Actuals:Formals); }           ; } .
Actual  = { Next:       Formals := {IF Formals^.Kind = Formal
                                    THEN Next:Formals := Formals^.Formal.\Next
                                    ELSE Next:Formals := Formals;
                                    END;}                            ; } .

END Formals

MODULE Env /* ---------------------------------------------------- */

EVAL GLOBAL     { FROM Definitions      IMPORT tEnv, NoEnv, mEnv; }

DECLARE Decls Stats Actuals Expr = [Env: tEnv INH] .

RULE

MiniLAX = { Proc:       Env     := NoEnv                             ; } .
Proc    = { Stats:      Env     := mEnv (Decls:DeclsOut, Env)        ;
            Decls:      Env     := Stats:       Env                  ; } .

END Env

MODULE Type /* --------------------------------------------------- */
```

```
EVAL GLOBAL     {
FROM Definitions IMPORT GetType;
FROM Types       IMPORT GetElementType, Reduce, ResultType;
FROM Tree        IMPORT tTree, mBoolean, mInteger, mReal, mRef, mNoType;
}

RULE

Expr     = {             Type    := nNoType                             ; } .
Binary   = {             Type    := ResultType (Lop:Type, Rop:Type, Operator); } .
Unary    = {             Type    := ResultType (Expr:Type, nNoType, Operator); } .
IntConst  = {            Type    := nInteger                            ; } .
RealConst = {            Type    := nReal                               ; } .
BoolConst = {            Type    := nBoolean                            ; } .
Adr      = {             Type    := nNoType                             ; } .
Index    = {             Type    := mRef (GetElementType (type))        ;
                         type    := Reduce (Adr:Type)                   ; } .
Ident    = {             Type    := GetType (Object)                    ; } .

END Type

MODULE TypeCode /* -------------------------------------------------- */

EVAL GLOBAL     { FROM ICodeInter IMPORT IntType, RealType, BoolType; }

DECLARE Read Write Binary = [type: tTree] .

Read     = {             type     := Reduce (Adr:Type)                 ;
                         TypeCode := ICodeType [type^.Kind]            ; } .
Write    = {             type     := Reduce (Expr:Type)                ;
                         TypeCode := ICodeType [type^.Kind]            ; } .
Binary   = {             type     := Reduce (Rop:Type)                 ;
                         TypeCode := ICodeType [type^.Kind]            ; } .

END TypeCode

MODULE Co /* -------------------------------------------------- */

EVAL GLOBAL     { FROM Types    IMPORT Reduce1, ReduceToRef, Coerce; }

RULE

Assign   = { Adr :       Co := Coerce (Adr :Type, ReduceToRef (Adr:Type));
             Expr:       Co := Coerce (Expr:Type, Reduce (Adr:Type))     ; } .
If       = { Expr:       Co := Coerce (Expr:Type, Reduce (Expr:Type))    ; } .
While    = { Expr:       Co := Coerce (Expr:Type, Reduce (Expr:Type))    ; } .
Read     = { Adr :       Co := Coerce (Adr :Type, ReduceToRef (Adr:Type)); } .
Write    = { Expr:       Co := Coerce (Expr:Type, Reduce (Expr:Type))    ; } .
Actual   = { Expr:       Co := {
                IF Formals^.Kind = NoFormal
                THEN Expr:Co := NIL;
                ELSE Expr:Co := Coerce (Expr:Type, Reduce1 (Formals^.Formal.Type));
                END; }                                                   ; } .
Binary   = { Lop :       Co := Coerce (Lop :Type, Reduce (Lop:Type))     ;
             Rop :       Co := Coerce (Rop :Type, Reduce (Rop:Type))     ; } .
Unary    = { Expr:       Co := Coerce (Expr:Type, Reduce (Expr:Type))    ; } .
Index    = { Adr :       Co := Coerce (Adr :Type, ReduceToRef (Adr:Type));
             Expr:       Co := Coerce (Expr:Type, Reduce (Expr:Type))    ; } .

END Co

MODULE DataSize /* -------------------------------------------------- */

EVAL GLOBAL     { FROM Types    IMPORT TypeSize; }

DECLARE Decls Formals = [DataSize THREAD] .

RULE

MiniLAX = { Proc:        DataSizeIn      := 0                            ; } .
Decl     = {             DataSizeOut     := Next:        DataSizeOut     ; } .
Proc     = { Formals:    DataSizeIn      := 3                            ; } .
Var      = { Next:       DataSizeIn      :=              DataSizeIn + TypeSize (Reduce1 (Type)); } .
Formal   = { Next:       DataSizeIn      :=              DataSizeIn + 1  ; } .

END DataSize
```

```
MODULE CodeSize /* ------------------------------------------------- */

DECLARE Decls Stats Actuals Expr = [CodeSize THREAD] .
        Expr Coercions           = [CoercionSize SYN] .

RUL

MiniLAX = { Proc: CodeSizeIn  := 0                  ; } .
Decl    = {       CodeSizeOut := Next: CodeSizeOut   ; } .
Proc    = { Stats:CodeSizeIn  :=       CodeSizeIn +1 ;                  /* ENT */
            Decls:CodeSizeIn  := Stats:CodeSizeOut+1 ;                  /* RET */
            Next: CodeSizeIn  := Decls:CodeSizeOut   ; } .
Stat    = {       CodeSizeOut := Next: CodeSizeOut   ; } .
Assign  = { Adr:  CodeSizeIn  :=       CodeSizeIn    ;
            Expr: CodeSizeIn  := Adr:  CodeSizeOut+Adr:CoercionSize;
            Next: CodeSizeIn  := Expr: CodeSizeOut+Expr:CoercionSize+1; /* STI */ } .
Call    = { Actuals:CodeSizeIn:=       CodeSizeIn+1  ;                  /* MST */
            Next: CodeSizeIn  := Actuals:CodeSizeOut+1;                 /* JSR */ } .
If      = { Expr: CodeSizeIn  :=       CodeSizeIn    ;
            Then: CodeSizeIn  := Expr: CodeSizeOut+Expr:CoercionSize+1; /* FJP */
            Else: CodeSizeIn  := Then: CodeSizeOut+1 ;                  /* JMP */
            Next: CodeSizeIn  := Else: CodeSizeOut   ; } .
While   = { Stats:CodeSizeIn  :=       CodeSizeIn +1 ;                  /* JMP */
            Expr: CodeSizeIn  := Stats:CodeSizeOut   ;
            Next: CodeSizeIn  := Expr: CodeSizeOut+Expr:CoercionSize+2;
                                                                   /* INV, FJP */ } .
Read    = { Adr:  CodeSizeIn  :=       CodeSizeIn    ;
            Next: CodeSizeIn  := Adr:  CodeSizeOut+Adr:CoercionSize+2;
                                                                   /* REA, STI */ } .
Write   = { Expr: CodeSizeIn  :=       CodeSizeIn    ;
            Next: CodeSizeIn  := Expr: CodeSizeOut+Expr:CoercionSize+1; /* WRI */ } .
Actual  = { Expr: CodeSizeIn  :=       CodeSizeIn    ;
            Next: CodeSizeIn  := Expr: CodeSizeOut+Expr:CoercionSize;
                  CodeSizeOut := Next: CodeSizeOut   ; } .
Binary  = { Rop:  CodeSizeIn  := Lop:  CodeSizeOut+Lop:CoercionSize;
                  CodeSizeOut := Rop:  CodeSizeOut+Rop:CoercionSize+1;
                                                     /* INV, MUL, ADD or LES */ } .
Unary   = {       CodeSizeOut := Expr: CodeSizeOut+Expr:CoercionSize+1; /* NOT */ } .
IntConst  = {     CodeSizeOut :=       CodeSizeIn+1  ;                  /* LDC */ } .
RealConst = {     CodeSizeOut :=       CodeSizeIn+1  ;                  /* LDC */ } .
BoolConst = {     CodeSizeOut :=       CodeSizeIn+1  ;                  /* LDC */ } .
Index     = { Expr:CodeSizeIn := Adr:  CodeSizeOut+Adr:CoercionSize;
                  CodeSizeOut := Expr: CodeSizeOut+Expr:CoercionSize+4;
                                                       /* CHK, LDC, SUB, IXA */ } .
Ident     = {     CodeSizeOut :=       CodeSizeIn+1  ;                  /* LDA */ } .

Expr      = {     CoercionSize:= Co:   CoercionSize  ; } .
Coercions = {     CoercionSize:= 0                   ; } .
Content   = {     CoercionSize:= Next: CoercionSize+1;                  /* LDI */ } .
IntToReal = {     CoercionSize:= Next: CoercionSize+1;                  /* FLT */ } .

END CodeSize

MODULE Level /* ------------------------------------------------- */

DECLARE Decls Formals Stats Actuals Expr = [Level: SHORTINT INH] .

RULE

MiniLAX = { Proc:       Level   := 0                                    ; } .
Proc    = { Formals:    Level   :=               Level + 1              ;
            Decls:      Level   := Formals:      Level                  ;
            Stats:      Level   := Formals:      Level                  ; } .
Call    = {             level   :=               Level                  ; } .
Ident   = {             level   :=               Level                  ; } .

END Level

MODULE Label /* ------------------------------------------------- */

RULE

If      = {             Label1  := Else:         CodeSizeIn             ;
                        Label2  := Else:         CodeSizeOut            ; } .
While   = {             Label1  := Stats:        CodeSizeIn             ;
                        Label2  := Expr:         CodeSizeIn             ; } .
```

```
END Label

MODULE Conditions /* ------------------------------------------------- */

EVAL GLOBAL     {
FROM Definitions IMPORT IsDeclared, IsObjectKind, NoObject, Proc, Var;
FROM Tree       IMPORT Integer, Boolean, Array, ErrorType, NoFormal, IsType, Error;
FROM Types      IMPORT IsAssignmentCompatible, IsSimpleType;
}

RULE

Decl    = { CHECK NOT IsDeclared (Ident, DeclsIn)
               ==> Error ("identifier already declared"         , Pos)          ; } .
Formal  = { CHECK NOT IsDeclared (Ident, DeclsIn)
               ==> Error ("identifier already declared"         , Pos)          ;
            CHECK IsSimpleType (Reduce1 (Type))
               ==> Error ("value parameter must have simple type", Pos)         ; } .
Array   = { CHECK Lwb <= Upb
               ==> Error ("lower bound exceeds upper bound"     , Pos)          ; } .
Assign  = { CHECK IsAssignmentCompatible (Adr:Type, Expr:Type)
               ==> Error ("types not assignment compatible"     , Pos)          ; } .
Call    = { CHECK Object^.Kind # NoObject
               ==> Error ("identifier not declared"             , Pos)          ;
            CHECK IsObjectKind (Object, Proc)
               ==> Error ("only procedures can be called"       , Pos)          ; } .
If      = { CHECK IsType (Reduce (Expr:Type), Boolean)
               ==> Error ("boolean expression required"         , Expr:Pos)     ; } .
While   = { CHECK IsType (Reduce (Expr:Type), Boolean)
               ==> Error ("boolean expression required"         , Expr:Pos)     ; } .
Read    = { CHECK IsSimpleType (Reduce (Adr:Type))
               ==> Error ("simple type operand required"        , Adr:Pos)      ; } .
Write   = { CHECK IsSimpleType (Reduce (Expr:Type))
               ==> Error ("simple type operand required"        , Expr:Pos)     ; } .
Binary  = { CHECK Type^.Kind # ErrorType
               ==> Error ("operand types incompatible"          , Pos)          ; } .
Unary   = { CHECK Type^.Kind # ErrorType
               ==> Error ("operand types incompatible"          , Pos)          ; } .
Index   = { CHECK IsType (Reduce (Adr:Type), Array)
               ==> Error ("only arrays can be indexed"          , Adr:Pos)      ;
            CHECK IsType (Reduce (Expr:Type), Integer)
               ==> Error ("integer expression required"         , Expr:Pos)     ; } .
Ident   = { CHECK Object^.Kind # NoObject
               ==> Error ("identifier not declared"             , Pos)          ;
            CHECK IsObjectKind (Object, Var)
               ==> Error ("variable required"                   , Pos)          ; } .

END Conditions

MODULE TypeDecls /* ------------------------------------------------- */

TREE IMPORT     {
FROM SYSTEM     IMPORT ADDRESS;
FROM Definitions IMPORT tObjects, tEnv;
IMPORT Errors, Scanner;

PROCEDURE Error (Text: ARRAY OF CHAR; Position: Scanner.tPosition);

TYPE tvoid      = RECORD END;

CONST
   Plus         = 1;
   Times        = 2;
   Less         = 3;
   Not          = 4;
}

EXPORT          { TYPE MyTree = tTree; }

GLOBAL          {
FROM Strings    IMPORT tString, ArrayToString;
IMPORT Errors, Scanner;

PROCEDURE Error (Text: ARRAY OF CHAR; Position: tPosition);
   BEGIN
      Errors.Message (Text, Errors.Error, Position);
```

```
   END Error;
}

EVAL GLOBAL     {
TYPE MyTree     = Tree.tTree;

VAR nNoObject   : tObjects;
VAR nInteger, nReal, nBoolean, nNoType  : tTree;
VAR ICodeType   : ARRAY [Integer .. Boolean] OF [IntType .. BoolType];
}

BEGIN   {
   nNoObject    := mNoObject    ();
   nInteger     := mInteger     ();
   nReal        := mReal        ();
   nBoolean     := mBoolean     ();
   nNoType      := mNoType      ();

   ICodeType [Tree.Integer      ] := IntType    ;
   ICodeType [Tree.Real         ] := RealType   ;
   ICodeType [Tree.Boolean      ] := BoolType   ;
}

END TypeDecls
```

Contents