Ast - A Generator for Abstract Syntax Trees

J. Grosch

GESELLSCHAFT FÜR MATHEMATIK
UND DATENVERARBEITUNG MBH

FORSCHUNGSSTELLE FÜR
PROGRAMMSTRUKTUREN
AN DER UNIVERSITÄT KARLSRUHE

Project

Compiler Generation

Ast - A Generator for Abstract Syntax Trees

Josef Grosch

Aug. 3, 1992

Report No. 15

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

1. Introduction

Ast is a generator for program modules that define the structure of abstract syntax trees and provide general tree manipulating procedures. The defined trees may be decorated with attributes of arbitrary types. Besides trees, graphs can be handled, too. The structure of the trees is specified by a formalism based on context-free grammars. The generated module includes procedures to construct and destroy trees, to read and write trees from (to) files, and to traverse trees in some commonly used manners. The mentioned readers and writers process ascii as well as binary tree representations. All procedures work for graphs as well.

The advantages of this approach are: record aggregates are provided which allow a concise notation for node creation. It is possible to build trees by writing terms. An extension mechanism avoids chain rules and allows, for example lists with elements of different types. Input/output procedures for records and complete graphs are provided. The output procedures and the interactive graph browser facilitate the debugging phase as they operate on a readable level and know the data structure. The user does not have to care about algorithms for traversing graphs. He/she is freed from the task of writing large amounts of relatively simple code. All of these features significantly increase programmer productivity.

Ast is implemented in Modula-2 as well as in C and generates Modula-2 or C source modules. The following sections define the specification language, explain the generated output, discuss related approaches, and present some examples.

2. Specification

The structure of trees and directed graphs is specified by a formalism based on context-free grammars. However, we primarily use the terminology of trees and types in defining the specification language. Its relationship to context-free grammars is discussed later.

2.1. Node Types

A tree consists of *nodes*. A node may be related to other nodes in a so-called *parent-child* relation. Then the first node is called a *parent* node and the latter nodes are called *child* nodes. Nodes without a parent node are usually called *root* nodes, nodes without children are called *leaf* nodes.

The structure and the properties of nodes are described by *node types*. Every node belongs to a node type. A specification for a tree describes a finite number of node types. A node type specifies the names of the child nodes and the associated node types as well as the names of the attributes and the associated attribute types. A node type is introduced by a name which can be an identifier or a string. The names of all node types have to be pairwise distinct. A node type can be regarded as a *nonterminal*, a *terminal*, or an *abstract* entity. Nonterminals are characterized by the character '=' following its name, terminals by the character ':', and abstract node types by the characters ':='. Undefined node types are implicitly defined to be terminals without attributes. The distinction between nonterminals and terminals is only of interest if concrete syntax is described. In the case of abstract syntax this distinction does not make sense and therefore nonterminal node types and eventually abstract ones suffice. Abstract node types are explained in section 2.6.

Example:

```
If        = .
While     = .
'()'      = .
Ident     : .
":="      : .
SCOPE     := .
```

The example defines the node types *If*, *While*, and *'()'* to be nonterminals, the node types *Ident* and *":="* to be terminals, and *SCOPE* to be an abstract node type.

The following names are reserved for keywords and can not be used for node types:

```
BEGIN        CLOSE     DECLARE   DEMAND    END
EVAL         EXPORT    FOR       FUNCTION  GLOBAL
IGNORE       IMPORT    IN        INH       INHERITED
INPUT        LEFT      LOCAL     MODULE    NONE
OUT          OUTPUT    PARSER    PREC      PROPERTY
REMOTE       REV       REVERSE   RIGHT     RULE
SCANNER      SELECT    STACK     SUBUNIT   SYN
SYNTHESIZED  THREAD    TREE      VIEW      VIRTUAL
VOID
```

2.2. Children

Children are distinguished by *selector* names which have to be unambiguous within one node type. The children are of a certain node type.

Example:

```
If        = Expr: Expr Then: Stats Else: Stats .
While     = Expr: Expr Stats: Stats .
```

The example introduces two node types called *If* and *While*. A node of type If has three children which are selected by the names *Expr*, *Then*, and *Else*. The children have the node types *Expr*, *Stats*, and *Stats*.

If a selector name is equal to the associated name of the node type it can be omitted. Therefore, the above example can be abbreviated as follows:

```
If        = Expr Then: Stats Else: Stats .
While     = Expr Stats .
```

2.3. Attributes

As well as children, every node type can specify an arbitrary number of *attributes* of arbitrary types. Like children, attributes are characterized by a selector name and a certain type. The descriptions of attributes are enclosed in brackets. The attribute types are given by names taken from the target language. Missing attribute types are assumed to be int or INTEGER depending on the target language (C or Modula-2). Children and attributes can be given in any order. The type of an attribute can be a pointer to a node type. In contrast to children, *ast* does not follow such an attribute during a graph traversal. All attributes are considered to be neither tree nor graph structured. Only the user knows about this fact and therefore he/she should take care.

Example:

```
Binary       = Lop: Expr Rop: Expr [Operator: int] .
Unary        = Expr [Operator] .
IntConst     = [Value] .
RealConst    = [Value: float] .
```

For example the node types *IntConst* and *RealConst* describe leaf nodes with an attribute named *Value* of types int or float respectively. *Binary* and *Unary* are node types with an attribute called *Operator* of type int.

2.4. Declare Clause

The DECLARE clause allows the definition of children and attributes for several node types at one time. Following the keyword DECLARE, a set of declarations can be given. The syntax is the same as described above with the exception that several node type names may introduce a declaration. If there already exists a declaration for a specified node type, the children and attributes are added to this declaration. Otherwise a new node type is introduced.

Example:

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

2.5. Extensions

To allow several alternatives for the types of children an *extension* mechanism is used. A node type may be associated with several other node types enclosed in angle brackets. The first node type is called *base* or *super* type and the latter types are called *derived* types or *subtypes*. A derived type can in turn be extended with no limitation of the nesting depth. The extension mechanism induces a subtype relation between node types. This relation is transitive. Where a node of a certain node type is required, either a node of this node type or a node of a subtype thereof is possible.

Example:

```
Stats           = <
   If           = Expr Then: Stats Else: Stats .
   While        = Expr Stats .
> .
```

In the above example *Stats* is a base type describing nodes with neither children nor attributes. It has two derived types called *If* and *While*. Where a node of type Stats is required, nodes of types Stats, If, and While are possible. Where a node of type If is required, nodes of type If are possible, only.

Besides extending the set of possible node types, the extension mechanism has the property of extending the children and attributes of the nodes of the base type. The derived types possess the children and attributes of the base type. They may define additional children and attributes. In other words they *inherit* the structure of the base type. The selector names of all children and attributes in an extension hierarchy have to be distinct. The syntax of extensions has been designed this way in order to allow single inheritance, only. Multiple inheritance is available, too. It is described in the next section.

Example:

```
Stats           = Next: Stats [Pos: tPosition] <
   If           = Expr Then: Stats Else: Stats .
   While        = Expr Stats .
> .
```

Nodes of type *Stats* have one child selected by the name *Next* and one attribute named *Pos*. Nodes of type *While* have three children with the selector names *Next*, *Expr*, and *Stats* and one attribute named *Pos*.

A node of a base type like *Stats* usually does not occur in an abstract syntax tree for a complete program. Still, *ast* defines this node type. It could be used as placeholder for unexpanded nonterminals in incomplete programs which occur in applications like syntax directed editors.

2.6. Multiple Inheritance

The extension mechanism described in the previous section allows for single inheritance of children and attributes. The syntax of the extensions has been designed to reflect the notation of context-free grammars as close as possible. For multiple inheritance a different syntax and the concept of *abstract node types* are used.

Abstract node types are characterized by the definition characters ':=' instead of '=' or ':' which are used for nonterminals or terminals. They are termed abstract because they describe only the structure of nodes or parts thereof but nodes of this types do not exist. Therefore no code is generated by for abstract node types: no constant, no record type, no constructor procedure, etc.. Abstract node types can be used as base types in combination with multiple inheritance.

For multiple inheritance the following syntax is used: The name of a node type may be followed by a left arrow '<-' and a list of names. This construct is available for all three kinds of node types: nonterminals, terminals, and abstract node types. The names after the left arrow have to denote abstract node types. The meaning is that the node type inherits the structure of all listed abstract node types. Multiple inheritance is possible from abstract node types to non abstract ones and among abstract node types. Among non abstract node types only single inheritance is allowed.

Example:

```
DECLS              := [Objects: tObjects THREAD] <
  NODECLS          := .
  DECL             := [Ident: tIdent INPUT] Next:DECLS .
> .

ENV                := [Env: tEnv INH] .

USE   <- ENV       := [Ident: tIdent INPUT] [Object: tObject SYN] .

SCOPE <- ENV       := [Objects: tObjects SYN] [NewEnv: tEnv SYN] .

Root               = Proc .

Decls <- DECLS ENV = <
  NoDecls          = .
  Decl <- DECL     = <
    Var            = .
    Proc <- SCOPE  = Decls Stats .
  > .
> .
Stats <- ENV       = <
  NoStats          = .
  Stat             = <
    Assign         = Name Expr .
    Call <- USE    = .
  > .
> .
Expr <- ENV        = <
  Plus             = Lop: Expr Rop: Expr .
  Const            = [Value] .
  Name <- USE      = .
> .
```

The above example uses multiple inheritance and abstract node types to describe the identification problem of programming languages. The node types written with all upper-case letters represent abstract node types. DECLS specifies lists of declared objects, SCOPE describes scopes or visibility regions, ENV stands for environment and is used to distribute scope information, and USE is intended to be used at the application of identifiers. In the second part of the example, the abstract node types are connected to nonterminal node types. *Decls* is the concrete node type to describe lists of declarations. *Decl* represents one declaration and there are to alternatives, *Var* and *Proc*, variables and procedures. A procedure introduces a scope and therefore it inherits from SCOPE. At the node types *Call* and *Name* identifiers are used and thus they inherit from USE. Finally, the node types *Decls*, *Stats*, and *Expr* are regions where the environment information has to be distributed and they inherit from ENV.

2.7. Modules

The specification of node types can be grouped into an arbitrary number of modules. The modules allow to combine parts of the specification that logically belong together. This feature can be used to structure a specification or to extend an existing one. A module consists of target code sections (see section 2.12.) and specifications of node types with attribute declarations. 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 the given children, attributes, and subtypes are added to the existing declaration. Otherwise a new node type is introduced. This way of modularization offers several possibilities:

- Context-free grammar and attribute declarations (= node types) can be combined in one module.
- The context-free grammar and the attribute declarations can be placed in two separate modules.
- The attribute declarations can be subdivided into several modules according to the tasks of semantic analysis. For example, there would be modules for scope handling, type determination, and context conditions.
- The information can be grouped according to language concepts or nonterminals. For example, there would be modules containing the grammar rules and the attribute declarations for declarations, statements, and expressions.

Example:

```
MODULE my_version

Stats       = [Env: tEnv] <                     /* add attribute   */
   While    = Init: Stats Terminate: Stats .    /* add children    */
   Repeat   = Stats Expr .                      /* add node type   */
> .

END my_version
```

2.8. 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.

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. They have no meaning for *ast*. The property *thread* supports so-called threaded attributes: An attribute declaration [a THREAD] is equivalent

Table 1: Properties for Children and Attributes

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

to the declaration of a pair of attributes with the suffixes In and Out: [aIn] [aOut]. These attributes have to be accessed with their full name including the suffixes. The property *reverse* specifies how lists should be reversed. It is discussed in section 2.11. The property *ignore* instructs *ast* to disregard or omit an attribute or a child. It is useful in connection with the concept of views (see section 2.10.). The property *virtual* is meaningful in attribute grammars. It is used to describe dependencies among attributes. However, no space will be allocated for those attributes and the attribute computations for those attributes will be omitted in the generated attribute evaluator. Within *ast* the properties *input*, *reverse*, and *ignore* are of interest, only.

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:

```
Stats = Next: Stats IN REV [Pos: tPosition IN] [Level 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 INPUT
PROPERTY SYN OUT FOR Mapping
```

Input attributes are included into the parameter list of the node constructor procedures (see section 3). The global property *input* replaces the symbol '->' of former versions of *ast*. For compatibility reasons this symbol still works in a restricted way: The symbol '->' could be included in a list of children and attributes as a shorthand notation to separate input from non-input items. In a list without this symbol all children and attributes are treated as input items. This meaning of the symbol '->' is still in effect as long as *ast* does not encounter a global property clause. After encountering such a clause, local and global properties are in effect only – the symbol '->' is ignored.

Example:

```
Stats           = Next: Stats REV [Pos: tPosition] -> [Level INH] <
   If           = Expr Then: Stats Else: Stats .
   While        = -> Expr IN Stats IN .
> .
```

The node types of the example possess the children and attributes listed in Table 2.

Table 2: Example of Properties

node type	selector name	associated type	kind	properties
Stats	Next	Stats	child	IN REV
	Pos	tPosition	attribute	IN
	Level	int	attribute	INH
If	Next	Stats	child	IN REV
	Pos	tPosition	attribute	IN
	Level	int	attribute	INH
	Expr	Expr	child	IN
	Then	Stats	child	IN
	Else	Stats	child	IN
While	Next	Stats	child	IN REV
	Pos	tPosition	attribute	IN
	Level	int	attribute	INH
	Expr	Expr	child	IN
	Stats	Stats	child	IN

2.9. Subunits

Usually, an *ast* specification is translated into one program module. This module receives the name that immediately follows the keyword TREE. If several modules contain a name, the first one is chosen. If none of the modules contains a name, the default name *Tree* is used. It is possible to generate several modules out of an *ast* specification. Then there is exactly one module called *main unit* that describes the tree structure and an arbitrary number of modules called *subunits*. This is of interest if the generated source code becomes too large for one compilation unit. Then either some or all desired procedures could be placed into separate subunits. In the extreme, there might be a subunit for every procedure. It is possible to have two or more versions of one procedure (e. g. WriteTREE) where every one uses a different view (see section 2.10.).

The names of the main unit and the subunits are described in the header of an *ast* specification:

```
[ TREE [ Name ] ] [ SUBUNIT Name ] [ VIEW Name ]
```

The name after the keyword VIEW is necessary if abstract syntax trees are to be processed by program modules generated with other tools such as e. g. *puma* [Gro91] that need to know the definition of the tree structure. *Ast* has an option that requests to write a binary version of the tree definition to a file whose name is derived from the name given after the keyword VIEW by appending the suffix ".TS" (Default: "Tree.TS").

Every unit has to be generated by a separate run of *ast*. If a subunit name is present, then a subunit is generated – otherwise a main unit is generated. The options select the procedures to be included in the units. It is probably wise not to include the subunit name in an *ast* specification. If this name is added "on the fly" with UNIX commands then different subunits can be generated from one specification without the need to change it.

Example:

```
                                       ast -dim spec.ast
echo SUBUNIT read  | cat - spec.ast | ast -dir
echo SUBUNIT write | cat - spec.ast | ast -diw
```
or
```
echo TREE MyTree                | cat - spec.ast | ast -dim
echo TREE MyTree SUBUNIT read   | cat - spec.ast | ast -dir
echo TREE MyTree SUBUNIT write  | cat - spec.ast | ast -diw
```

2.10. Views

An *ast* specification can be roughly seen as a collection of node types and associated children and attributes. A so-called *view* selects a subset of this collection and it may attach further properties to some parts of this collection.

The concept of views is necessary for instance if two programs communicate a common data structure via a file. Every program might need additional data which should neither appear in the other program nor in the file. In order to make this work both programs must agree upon the coding of the node types in the shared part of the data structure. This is accomplished by using one common *ast* specification that contains the description of the complete data structure. Every program uses a specific view and selects only those parts of the common specification that are of interest. See Figure 1 for an example.

Another need for views arises if several attribute evaluators operate one after the other on one tree. The output attributes of a preceding evaluator become the input attributes of a succeeding one. Here it is necessary to be able to change the properties of attributes. In one view the attributes are regarded as output and in the other one they are regarded as input. The usage of views for the specification of several attribute evaluators is described in [Gro89].

Furthermore, views are necessary if abstract syntax trees are to be processed by program modules generated with other tools such as e. g. *puma* [Gro91] that need to know the definition of the tree structure. In general, there might be several tree processing modules and every one uses a different view. In this case, the views have to be communicated to the other tool in a file. *ast* has an option that requests to write a binary version of the tree definition to a file whose name is derived from the name given after the keyword VIEW by appending the suffix ".TS" (Default: "Tree.TS", see section on "Subunits").

The concept of views is based on the global properties:

```
PROPERTY properties FOR module_names
```

allows the dynamic addition of properties.

```
PROPERTY IGNORE FOR module_names
```

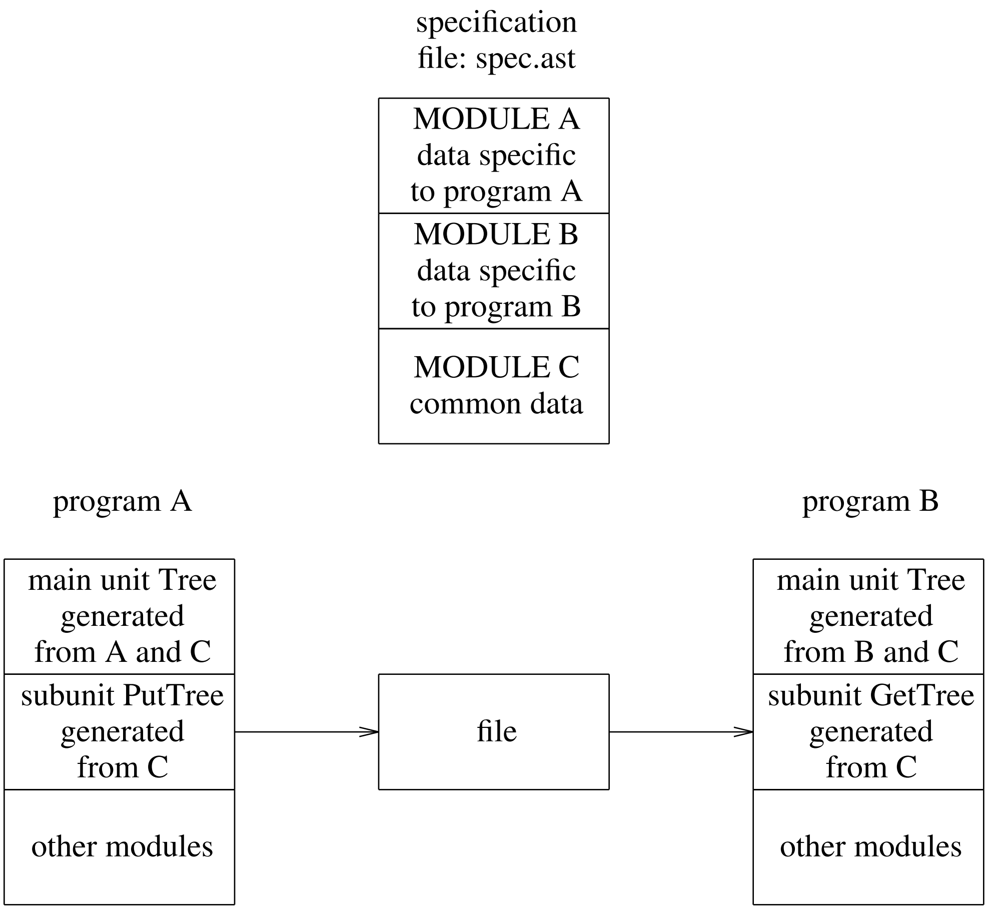
allows the suppression of all definitions given in the listed modules. Additionally there is the so-called select clause:

```
SELECT module_names
```

This is equivalent to:

```
PROPERTY IGNORE FOR module_names_not_given
```

It is wise to assign names to all modules of an *ast* specification, because otherwise they can not be selected with the select clause. Furthermore, the property or select clauses that express views should probably not be included in the file containing the *ast* specification. The reason is that this form is not flexible, because it is relatively hard to change. It is better to add the one line that is necessary for views "on the fly" using UNIX commands like echo and cat (see Figure 1).



UNIX commands to generate the compilation units:

```
echo                  SELECT A C | cat - spec.ast | ast -dim
echo SUBUNIT PutTree SELECT   C | cat - spec.ast | ast -dip
echo                  SELECT B C | cat - spec.ast | ast -dim
echo SUBUNIT GetTree SELECT   C | cat - spec.ast | ast -dig
```

Fig. 1: Programs Sharing a Part of a Data Structure

2.11. Reversal of Lists

Recursive node types like *Stats* in the abstract grammar of the example below describe lists of subtrees. There are at least two cases where it is convenient to be able to easily reverse the order of the subtrees in a list. The facility provided by *ast* is a generalization of an idea presented in [Par88].

2.11.1. LR Parsers

Using LR parsers, one might be forced to parse a list using a left-recursive concrete grammar rule because of the limited stack size. The concrete grammar rules of the following examples are written in the input language of the parser generator *Lalr* [GrV88, Gro88] which is similar to the one of YACC [Joh75]. The node constructor procedures within the semantic actions are the ones provided by *ast* (see section 3).

Example:

concrete grammar (with tree building actions):

```
Stats:                          {$$ = mStats0 ();       } .
Stats: Stats Stat ';'           {$$ = mStats1 ($2, $1);} .
Stat : WHILE Expr DO Stats END {$$ = mWhile  ($2, $4);} .
```

abstract grammar:

```
Stats           = <
   Stats0       = .
   Stats1       = Stat Stats .
> .
```

A parser using the above concrete grammar would construct statement lists where the list elements are in the wrong order, because the last statement in the source would be the first one in the list. The WHILE rule represents a location where statement lists are used.

To easily solve this problem *ast* can generate a procedure to reverse lists. The specification has to describe how this should be done. At most one child of every node type may be given the property *reverse*. The child's type has to be the node type itself or an associated base type. The generated list reversal procedure ReverseTREE then reverses a list with respect to this indicated child. The procedure ReverseTREE has to be called exactly once for a list to be reversed. This is the case at the location where a complete list is included as subtree (e. g. in a WHILE statement).

Example:

concrete grammar (with tree building actions):

```
Stats:                          {$$ = mStats0 ();       } .
Stats: Stats Stat ';'           {$$ = mStats1 ($2, $1);} .
Stat : WHILE Expr DO Stats END {$$ = mWhile  ($2, ReverseTREE ($4));} .
```

abstract grammar:

```
Stats           = <
   Stats0       = .
   Stats1       = Stat Stats REVERSE .
> .
```

It is possible to represent lists differently in an abstract syntax. A more sophisticated solution is given in the next example. The procedure ReverseTREE handles this variant, too.

Example:

concrete grammar (with tree building actions):

```
Stats:        {$$ = mEmpty ();} .
Stats: Stats IF Expr THEN Stats ELSE Stats END ';'
              {$$ = mIf ($3, ReverseTREE ($5), ReverseTREE ($7), $1);} .
Stats: Stats WHILE Expr DO Stats END ';'
              {$$ = mWhile ($3, ReverseTREE ($5), $1);} .
```

abstract grammar:

```
Stats           = Next: Stats REVERSE <
   Empty        = .
   If           = Expr Then: Stats Else: Stats .
   While        = Expr Stats .
> .
```

2.11.2. LL Parsers

Using LL parsers a similar problem as in the LR case can arise if extended BNF is used. Lists are parsed with an iteration which is turned into a loop statement as follows: (The identifiers Stats0, Stats1, Stat0, and Stat1 in the concrete grammar rules denote the symbolic access to the L-attribution mechanism provided by *Ell* [GrV88]. These identifiers should not be mixed up with the similar ones used as node names in the abstract syntax.)

Example:

concrete grammar (with tree building actions):

```
Stats: {*Stats0=mStats0 ();} ( Stat ';' {*Stats0=mStats1 (Stat1, Stats0);} ) * .
Stat : WHILE Expr DO Stats END {*Stat0=mWhile (Expr1, ReverseTREE (Stats1));} .
```

abstract grammar:

```
Stats           = <
   Stats0       = .
   Stats1       = Stat Stats REVERSE .
> .
```

The list elements (statements) are inserted in the wrong order within the first concrete grammar rule. The order is corrected by a call of the procedure ReverseTREE in the second concrete grammar rule.

2.12. Target Code

An *ast* specification may include several sections containing so-called *target code*. This sections follow the keywords TREE or SUBUNIT. Target code is code written in the target language which is copied unchecked and unchanged to certain places in the generated module. 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.

Example in C:

```
TREE SyntaxTree
IMPORT {# include "Idents.h" }
EXPORT {  typedef tSyntaxTree MyTree; }
GLOBAL {# include "Idents.h"
          typedef struct { unsigned Line, Column; } tPosition;
BEGIN  { ... }
CLOSE  { ... }
```

Example in Modula-2:

```
TREE SyntaxTree
IMPORT { FROM Idents IMPORT tIdent; }
EXPORT { TYPE MyTree = tSyntaxTree; }
GLOBAL { FROM Idents IMPORT tIdent;
         TYPE tPosition = RECORD Line, Column: CARDINAL; END; }
BEGIN  { ... }
CLOSE  { ... }
```

The meaning of the 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*.

GLOBAL:	declarations to be included in the implementation module at global level.
LOCAL:	same as GLOBAL within *ast*.
BEGIN:	statements to initialize the declared data structures.
CLOSE:	statements to finalize the declared data structures.

2.13. Common Node Fields

Sometimes it is desirable to include certain record fields into all node types. This can be done directly in the target language by defining the macro *TREE_NodeHead* in the IMPORT or EXPORT target code sections. These fields become members of the variant *yyHead* and they can be accessed as shown in the following examples:

Example in C:

```
# define Tree_NodeHead int MyField1; MyType MyField2;
t->yyHead.MyField1 = ... ;
```

Example in Modula-2:

```
# define Tree_NodeHead MyField1: INTEGER; MyField2: MyType;
t^.yyHead.MyField1 := ... ;
```

2.14. Type Specific Operations

Procedures generated by *ast* apply several operations to attributes: initialization, finalization, ascii read and write, binary read and write, and copy (see Table 3). *Initialization* is performed whenever a node is created. It can range from assigning an initial value to the allocation of dynamic storage or the construction of complex data structures. *Finalization* is performed immediately before a node is deleted and may e. g. release dynamically allocated space. The *read* and *write* operations enable the readers and writers to handle the complete nodes including all attributes, even those of user-defined types. The operation *copy* is needed to duplicate values of attributes of user-defined types. By default, *ast* just copies the bytes of an attribute to duplicate it. Therefore, pointer semantics is assumed for attributes of a pointer type. If value semantics is needed, the user has to take care about this operation. The operation *equal* checks whether two attributes are equal. It is used as atomic operation for the procedure that tests the equality of trees.

The operations are type specific in the sense that every type has its own set of operations. All attributes having the same type (target type name) are treated in the same way. Chosing

Table 3: Type Specific Operations

operation	macro name	default macro C	default macro Modula-2
initialization	beginTYPE(a)		
finalization	closeTYPE(a)		
ascii read	readTYPE(a)	yyReadHex (& a, sizeof (a));	yyReadHex (a);
ascii write	writeTYPE(a)	yyWriteHex (& a, sizeof (a));	yyWriteHex (a);
binary read	getTYPE(a)	yyGet (& a, sizeof (a));	yyGet (a);
binary write	putTYPE(a)	yyPut (& a, sizeof (a));	yyPut (a);
copy	copyTYPE(a, b)		
equal	equalTYPE(a, b)	memcmp (& a, & b, sizeof (a)) == 0	yyIsEqual (a, b)

different type names for one type introduces subtypes and allows to treat attributes of different subtypes differently. Type operations for the predefined types of a target language are predefined within *ast* (see Appendices 6 and 7). For user-defined types, *ast* assumes default operations (see Table 3). The procedures yyReadHex and yyWriteHex read and write the bytes of an attribute as hexadecimal values. The procedures yyGet and yyPut read and write the bytes of an attribute unchanged (without conversion). The operations are defined by a macro mechanism. The procedure yyIsEqual checks the bytes of two attributes for equality. TYPE is replaced by the concrete type name. *a* is a formal macro parameter referring to the attribute. The predefined procedures mentioned in Table 3 use the global variable *yyf* of type FILE * or IO.tFile [Gro87] describing the file used by the readers and writers.

It is possible to redefine the operations by including new macro definitions in the GLOBAL section. The following example demonstrates the syntax for doing this. It shows how records of the type tPosition might be handled and how subtypes can be used to initialize attributes of the same type differently.

Example in C:

```
IMPORT {
# include "Sets.h"
typedef struct { unsigned Line, Column; } tPosition;
typedef tSet Set100;
typedef tSet Set1000;
}

GLOBAL {
# define begintPosition(a) a.Line = 0; a.Column = 0;
# define readtPosition(a)  fscanf (yyf, "%d%d", & a.Line, & a.Column);
# define writetPosition(a) fprintf (yyf, "%d %d", a.Line, a.Column);

# define beginSet100(a)    MakeSet    (& a, 100);
# define closeSet100(a)    ReleaseSet (& a);
# define readSet100(a)     ReadSet    (yyf, & a);
# define writeSet100(a)    WriteSet   (yyf, & a);

# define beginSet1000(a)   MakeSet    (& a, 1000);
# define closeSet1000(a)   ReleaseSet (& a);
# define readSet1000(a)    ReadSet    (yyf, & a);
# define writeSet1000(a)   WriteSet   (yyf, & a);
}
```

Example in Modula-2:

```
IMPORT {
   FROM Sets      IMPORT tSet;
   TYPE tPosition = RECORD Line, Column: CARDINAL; END;
   TYPE Set100    = tSet;
   TYPE Set1000   = tSet;
}

GLOBAL {
   FROM IO        IMPORT ReadI, WriteI, WriteC;
   FROM Sets      IMPORT MakeSet, ReleaseSet, ReadSet, WriteSet;

# define begintPosition(a) a.Line := 0; a.Column := 0;
# define readtPosition(a)  a.Line := ReadI (yyf); a.Column := ReadI (yyf);
# define writetPosition(a) WriteI (yyf, a.Line, 0); WriteC (yyf, ' '); \
                           WriteI (yyf, a.Column, 0);

# define beginSet100(a)    MakeSet    (a, 100);
# define closeSet100(a)    ReleaseSet (a);
# define readSet100(a)     ReadSet    (yyf, a);
# define writeSet100(a)    WriteSet   (yyf, a);

# define beginSet1000(a)   MakeSet    (a, 1000);
# define closeSet1000(a)   ReleaseSet (a);
# define readSet1000(a)    ReadSet    (yyf, a);
# define writeSet1000(a)   WriteSet   (yyf, a);
}
```

2.15. Storage Management

The storage management for the nodes to be created is completely automatic. Usually, the user does not have to care about it. The predefined storage management works as follows: Every generated tree module contains its own heap manager which is designed in favour of speed. The constructor procedures use an in-line code sequence to obtain storage (see below). The heap manager does not maintain free lists. It only allows to free the complete heap of one module using the procedure *ReleaseTREEModule*. The procedure *ReleaseTREE* does not free the node space, it only finalizes the attributes of the node.

To change the predefined behaviour, two macro definitions may be included in the GLOBAL section. For C these macros are initialized as follows:

```
# define yyALLOC(ptr, size) \
  if ((ptr = (tTREE) TREE_PoolFreePtr) >= (tTREE) TREE_PoolMaxPtr) \
     ptr = TREE_Alloc (); \
  TREE_PoolFreePtr += size;
# define yyFREE(ptr, size)
```

For Modula-2 these macros are initialized as follows:

```
# define yyALLOC(ptr, size) ptr := yyPoolFreePtr; \
  IF SYSTEM.ADDRESS (ptr) >= yyPoolMaxPtr THEN ptr := yyAlloc (); END; \
  INC (yyPoolFreePtr, size);
# define yyFREE(ptr, size)
```

The following lines switch the heap manager to a global storage allocator with free lists:

```
# define yyALLOC(ptr, size) ptr = Alloc (size);
# define yyFREE(ptr, size)  Free (size, ptr);
```

Now *ReleaseTREE* will work as expected whereas *ReleaseTREEModule* does not work any more.

3. About the Generated Program Module

A specification as described in the previous section is translated by *ast* into a program module consisting of a definition and an implementation part. Only the definition part or header file respectively is sketched here – Appendices 4 and 5 contain the general schemes. The definition part contains primarily type declarations to describe the structure of the trees and the headings of the generated procedures.

Every node type is turned into a constant declaration and a struct or record declaration. That is quite simple, because node types and record declarations are almost the same concepts except for the extension mechanism and some shorthand notations. All these records become members of a union type or a variant record used to describe tree nodes in general. This variant record has a tag field called *Kind* which stores the code of the node type. A pointer to the variant record is a type representing trees. Like all generated names, this pointer type is derived from the name of the specification. Table 4 briefly explains the exported objects (replace TREE by the name of the generated module (see section 2.12.) and <node type> by all the names of node types). Whereas in Modula-2 the names of the constants to code the node types and the names of the record variants are identical to the names of the node types this is not the case in C. In C only the names of the union members are identical, the constant names are prefixed with the letter 'k' standing for *Kind*.

The parameters of the procedures *m<node*type> have to be given in the order of the *input* attributes in the specification. Attributes of the base type (recursively) precede the ones of the derived type. The procedures *TraverseTREETD* and *TraverseTREEBU* visit all nodes of a tree or a graph respectively. At every node a procedure given as parameter is executed. An assignment of a tree or graph to a variable of type *tTREE* can be done in two ways: The usual assignment operators '=' or ':=' yield pointer semantics. The procedure *CopyTREE* yields value semantics by duplicating a given graph.

The procedure *QueryTREE* allows to browse a tree and to inspect one node at a time. A node including the values of its attributes is printed on *standard output*. Then the user is prompted to provide one of the following commands from *standard input*:

parent	display parent node
quit	quit procedure QueryTREE
<selector>	display specified child (first match, abbreviation possible)
<selector><space>	display specified child (exact match, no abbreviation)

All commands can be abbreviated to an unambiguous prefix. Usually, a first match strategy is used to determine a child from its (abbreviated) selector name. With this search strategy, children whose name is a prefix of others may not be accessible. If an unabbreviated selector name is supplied together with a following space character an exact match strategy is used, which allows to access every child. The empty command behaves like *parent*.

The construction of the pointer and the union type above does not enforce the tree typing rules through the types of the target language. In fact, it is possible to construct trees that violate the specification. The user is responsible to adhere to the type rules. Most of the generated procedures do not care about the type rules. Moreover, type violations are possible and such erroneous trees are handled correctly by all procedures. The procedure *CheckTREE* can be used to check if a tree is properly typed. In case of typing errors the involved parent and child nodes are printed on *standard error*.

The binary graph writer procedure *GetTREE* produces a binary file containing the graph in linearized form. The nodes are written according to a depth first traversal. Edges are either represented by concatenation of nodes or by symbolic (integer) labels. The following kinds of records specified by C types are written to the file:

Table 4: Generated Objects and Procedures

object/procedure	description
k<node type>	named constant to encode a node type in C
<node type>	named constant to encode a node type in Modula-2
<node type>	name of union member or record variant for a node type
tTREE	pointer type, refers to variant record type describing all node types
TREERoot	variable of type tTREE, can serve as root
	(additional variables can be declared)
TREE_NodeName	array mapping node types to names (strings) in C
TREE_NodeSize	array mapping node types to the size of the nodes in C
yyNodeSize	array mapping node types to the size of the nodes in Modula-2
MakeTREE	node constructor procedure without attribute initialization
IsType	test a node for a certain type
n<node type>	node constructor procedures with attribute initialization
	according to the type specific operations
m<node type>	node constructor procedures with attribute initialization
	from a parameter list for *input* attributes
ReleaseTREE	node or graph finalization procedure,
	all attributes are finalized, all node space is deallocated
ReleaseTREEModule	deallocation of all graphs managed by a module
WriteTREENode	ASCII node writer procedure
ReadTREE	ASCII graph reader procedure
WriteTREE	ASCII graph writer procedure
GetTREE	binary graph reader procedure
PutTREE	binary graph writer procedure
ReverseTREE	procedure to reverse lists
TraverseTREETD	top down graph traversal procedure (reverse depth first)
TraverseTREEBU	bottom up graph traversal procedure (depth first search)
CopyTREE	graph copy procedure
IsEqualTREE	equality test procedure for trees
CheckTREE	graph syntax checker procedure
QueryTREE	graph browser procedure
BeginTREE	procedure to initialize user-defined data structures
CloseTREE	procedure to finalize user-defined data structures

```
# define yyNil          0374
# define yyNoLabel      0375
# define yyLabelDef     0376
# define yyLabelUse     0377

typedef unsigned char  TREE_tKind;   /* less than 252 node types */
typedef unsigned short TREE_tKind;   /* more than 251 node types */
typedef unsigned short TREE_tLabel;

struct { char yyNil     ;                                 } NoTree;
struct { char yyLabelUse; TREE_tLabel <label>;            } LabelUse;
struct { char yyLabelDef; TREE_tLabel <label>;
                          TREE_tKind Kind; <attributes> } LabelDef;
struct { char yyNoLabel ; TREE_tKind Kind; <attributes> } NoLabel;
struct { char Kind      ; <attributes>                    } Kind;
```

Record fields whose name starts with *yy* have a constant value as defined. <label> is an integer representing a certain address. <attributes> are written with the type specific *put* macros which either copy the bytes of an attribute unchanged or do whatever the user has specified. If the value of the tag field Kind is less than 252 then the format *Kind* is used, otherwise the format *NoLabel* is used to write unlabeled nodes.

4. Using the Generated Program Module

This section explains how to use the objects of the generated program module. Trees or graphs are created by successively creating their nodes. The easiest way is to call the constructor procedures m<node_type>. These combine node creation, storage allocation, and attribute assignment. They provide a mechanism similar to record aggregates. Nested calls of constructor procedures allow programming with (ground) terms as in Prolog or LISP. In general, a node can be created by a call of one of the procedures

MakeTREE, n<node type>, or m<node type>.

The type of a node can be retrieved by examination of the predefined tag field called *Kind*. Alternatively the function *IsType* can be used to test whether a node has a certain type or a subtype thereof. Children and attributes can be accessed using two record selections. The first one states the node type and the second one gives the selector name of the desired item.

Example in C:

abstract syntax:

```
Expr         = [Pos: tPosition] <
   Binary    = Lop: Expr Rop: Expr [Operator: int] .
   Unary     = Expr [Operator] .
   IntConst  = [Value] .
   RealConst = [Value: float] .
> .
```

tree construction by a term:

```
# define Plus 1
tTREE t;
tPosition Pos;

t = mBinary (Pos, mIntConst (Pos, 2), mIntConst (Pos, 3), Plus);
```

tree construction during parsing:

```
Expr: Expr '+' Expr {$$.Tree = mBinary ($2.Pos, $1.Tree, $3.Tree, Plus);} .
Expr:      '-' Expr {$$.Tree = mUnary  ($1.Pos, $2.Tree, Minus);        } .
Expr: IntConst      {$$.Tree = mIntConst ($1.Pos, $1.IntValue);         } .
Expr: RealConst     {$$.Tree = mRealConst ($1.Pos, $1.RealValue);       } .
```

tree construction using a statement sequence:

```
t = MakeTREE (Binary);
t->Binary.Pos.Line            = 0;
t->Binary.Pos.Column          = 0;
t->Binary.Lop                 = MakeTREE (IntConst);
t->Binary.Lop->IntConst.Pos   = Pos;
t->Binary.Lop->IntConst.Value = 2;
t->Binary.Rop                 = MakeTREE (IntConst);
t->Binary.Rop->IntConst.Pos   = Pos;
t->Binary.Rop->IntConst.Value = 3;
t->Binary.Operator            = Plus;
```

access of tag field, children, and attributes:

```
switch (t->Kind) {
case kExpr  : ... t->Expr.Pos              ...
case kBinary: ... t->Binary.Operator       ...
              ... t->Binary.Lop            ...
case kUnary : ... t->Unary.Expr->Expr.Pos ...
};
```

Example in Modula-2:

abstract syntax:

```
Expr         = [Pos: tPosition] <
   Binary    = Lop: Expr Rop: Expr [Operator: INTEGER] .
   Unary     = Expr [Operator] .
   IntConst  = [Value] .
   RealConst = [Value: REAL] .
> .
```

tree construction by a term:

```
CONST Plus = 1;
VAR t: tTREE; Pos: tPosition;

t := mBinary (Pos, mIntConst (Pos, 2), mIntConst (Pos, 3), Plus);
```

tree construction during parsing:

```
Expr: Expr '+' Expr {$$.Tree := mBinary ($2.Pos, $1.Tree, $3.Tree, Plus);} .
Expr:      '-' Expr {$$.Tree := mUnary  ($1.Pos, $2.Tree, Minus);        } .
Expr: IntConst      {$$.Tree := mIntConst ($1.Pos, $1.IntValue);         } .
Expr: RealConst     {$$.Tree := mRealConst ($1.Pos, $1.RealValue);       } .
```

tree construction using a statement sequence:

```
t := MakeTREE (Binary);
t^.Binary.Pos.Line             := 0;
t^.Binary.Pos.Column           := 0;
t^.Binary.Lop                  := MakeTREE (IntConst);
t^.Binary.Lop^.IntConst.Pos    := Pos;
t^.Binary.Lop^.IntConst.Value  := 2;
t^.Binary.Rop                  := MakeTREE (IntConst);
t^.Binary.Rop^.IntConst.Pos    := Pos;
t^.Binary.Rop^.IntConst.Value  := 3;
t^.Binary.Operator             := Plus;
```

access of tag field, children, and attributes:

```
CASE t^.Kind OF
| Expr  : ... t^.Expr.Pos              ...
| Binary: ... t^.Binary.Operator       ...
          ... t^.Binary.Lop            ...
| Unary : ... t^.Unary.Expr^.Expr.Pos ...
END;
```

5. Related Research

5.1. Variant Records

Ast specifications and variant record types like in Pascal or Modula-2 are very similar. Every node type in an *ast* specification corresponds to a single variant. In the generated code every node type is translated into a record type. All record types become members of a variant record type representing the type for the trees.

The differences are the following: *Ast* specifications are shorter than directly hand-written variant record types. *Ast* specifications are based on the formalism of context-free grammars (see section 5.3.). The generator *ast* automatically provides operations on record types which would be simple but voluminous to program by hand. The node constructor procedures allow to write record aggregates and provide dynamic storage management. The reader and writer procedures supply input and output for record types and even for complete linked data structures such as trees and graphs.

5.2. Type Extensions

Type extensions have been introduced with the language Oberon [Wir88a, Wir88b, Wir88c]. The extension mechanism of *ast* is basically the same as in Oberon. The notions extension, base type, and derived type are equivalent. *Type tests* and *type guards* can be easily programmed by inspecting the tag field of a node. *Ast* does not provide assignment of subtypes to base types in the sense of value semantics or a projection, respectively. The tool can be seen as a preprocessor providing type extensions for Modula-2 and C.

The second example in section 2.5. illuminates the relationship between *ast* and Oberon. The node type Stats is a base type with two fields, a child and an attribute. It is extended e. g. by the node type While with two more fields which are children.

5.3. Context-Free Grammars

As already mentioned, *ast* specifications are based on context-free grammars. *Ast* specifications extend context-free grammars by selector names for right-hand side symbols, attributes, the extension mechanism, and modules. If these features are omitted, we basically arrive at context-free grammars. This holds from the syntactic as well as from the semantic point of view. The names of the node types represent both terminal or nonterminal symbols and rule names. Node types correspond to grammar rules. The notions of derivation and derivation tree can be used similarly in both cases. The selector names can be seen as syntactic sugar and the attributes as some kind of terminal symbols. The extension mechanism is equivalent to a short-hand notation for factoring out common rule parts in combination with implicit chain rules.

Example:

ast specification:

```
Stats           = Next: Stats <
   If           = Expr Then: Stats Else: Stats .
   While        = Expr Stats .
> .
```

corresponding context-free grammar:

```
Stats           = Stats .
Stats           = Stats If .
Stats           = Stats While .
If              = Expr Stats Stats .
While           = Expr Stats .
```

In the example above, Stats corresponds to a nonterminal. There are two rules or right-hand sides for Stats which are named If and While. The latter would be regarded as nonterminals, too, if a child of types If or While would be specified.

5.4. Attribute Grammars

Attribute grammars [Knu68, Knu71] and *ast* specifications are based on context-free grammars and associate attributes with terminal and nonterminals symbols. Additionally *ast* allows attributes which are local to rules. As the structure of the tree itself is known and transparent, subtrees can be accessed or created dynamically and used as attribute values. The access of the right-hand side symbols uses the selector names for symbolic access instead of the grammar symbol with an additional subscript if needed. There is no need to map chain rules to tree nodes because of the extension mechanism offered by *ast.* Attribute evaluation is outside the scope of *ast*. This can be done either with the attribute evaluator generator *ag* [Gro89] which understands *ast* specifications extended by attribute computation rules and processes the trees generated by *ast* or by hand-written programs that use an *ast* generated module. In the latter case attribute computations do not have to obey the single assignment restriction for attributes. They can assign a value to an attribute zero, one, or several times.

5.5. Interface Description Language (IDL)

The approach of *ast* is similar to the one of IDL [Lam87, NNG89]. Both specify attributed trees as well as graphs. Node types without extensions are called nodes in IDL and node types with extensions (base types) are called classes. *Ast* has simplified this to the single notion of a node type. Attributes are treated similarly in both systems. Children and attributes are both regarded as attributes, as structural and non-structural ones, with only little difference in between. Both systems allow multiple inheritance of attributes, *ast* has a separate syntax for single inheritance and uses the notion extension instead [Wir88c]. IDL knows the predefined types INTEGER, RATIONAL, BOOLEAN, STRING, SEQ OF, and SET OF. It offers special operations for the types SEQ OF and SET OF. *Ast* really has no built in types at all, it uses the ones of the target language and has a table containing the type specific operations e. g. for reading and writing. Both *ast* and IDL allow attributes of user-defined types. In *ast*, the type specific operations for predefined or user-defined types are easily expressed by macros using the target language directly. IDL offers an assertion language whereas *ast* does not. IDL provides a mechanism to modify existing specifications. The module feature of *ast* can be used to extend existing specifications. From *ast*, readers and writers are requested with simple command line options instead of complicated syntactic constructs. *Ast* does not support representation specifications, because representations are much more easily expressed using the types of the target language directly. Summarizing, we consider *ast* to have a simpler specification method and to generate more powerful features like aggregates, reversal of lists, and graph browsers.

5.6. Attribute Coupled Grammars

Attribute coupled grammars (ACG's) [Gie88] or algebraic specifications [HHK88] have only very little in common with *ast* specifications. They all view node types or rules as signatures of functions. The name of the node type plays the role of the function name and describes the result type. The types of the children and attributes correspond to the type of the function arguments. The constructor procedures generated by *ast* reflect this view best.

5.7. Object-Oriented Languages

The extension mechanism of *ast* is exactly the same as single inheritance in object-oriented languages like e. g. Simula [DMN70] or Smalltalk [Gol84]. The hierarchy introduced by the extension mechanism corresponds directly to the class hierarchy of object-oriented languages. The notions base type and super class both represent the same concept. Messages and virtual procedures are out of the scope of *ast.* Virtual procedures might be simulated with procedure-valued attributes. Table 5 summarizes the corresponding notions of trees (*ast*), type extensions, and object-oriented programming.

Table 5: Comparison of notions from the areas of trees, types, and object-oriented programming

trees	types	object-oriented programming
node type	record type	class
-	base type	superclass
-	derived type	subclass
attribute, child	record field	instance variable
tree node	record variable	object, instance
-	extension	inheritance

5.8. Tree Grammars

Conventional tree grammars are characterized by the fact that all right-hand sides start with a terminal symbol. They are used for the description of string languages that represent trees in prefix form. *Ast* specifications describe trees which are represented by (absolute) pointers from parent to child nodes. If we shift the names of node types of *ast* specifications to the beginning of the right-hand side and interpret them as terminals we arrive at conventional tree grammars. That is exactly what is done by the *ast* tree/graph writers. They write a tree in prefix form and prepend every node with the name of its node type. That is necessary to be able to perform the read operations.

6. Hints on Specifying Abstract Syntax

- Keep the abstract syntax as short and simple as possible.
- Try to normalize by representing only the most general form.
- Normalize to the general form e. g. by adding default values.
- Normalize several concrete representations to one abstract construct.
- Map concrete to abstract syntax by disregarding the concrete syntax rules and by concentrating on the semantic structure of the abstract syntax.
- Map several concrete nonterminals to one abstract node type (e. g. Expr, Term, and Factor $\rightarrow$ Expr)
- Allow all lists to be empty regardless of the concrete syntax. Otherwise you have to process the list element at two places in exactly the same way. This causes programming overhead and violates the law of singularity: "One thing only once!"
- Operators can be represented by different node types (e. g. Plus, Minus, Mult, ...) or by one node type with an attribute describing the operator (e. g. Binary).
- Lists can be represented by separate nodes for the list and the elements (e. g. Stats and Stat) or by nodes for the elements where every node has a child that refers to the next list element (see last example in section 2.11.1.).

7. Examples

The Appendices 1 to 3 contain examples of *ast* specifications.

Appendix 1 contains the concrete syntax of *ast's* specification language. The node types enclosed in quotes or starting with the character 't' constitute the terminals for *ast's* parser. The extensions and the node types used for the latter describe the lexical grammar.

Appendix 2 contains the concrete syntax of a small example programming language called *MiniLAX* [GrK88]. The attributes specified are the ones a parser would evaluate during parsing. The Appendices 1 and 2 show how concrete grammars can be described with *ast*.

Appendix 3 contains an abstract syntax for MiniLAX. The attributes specified are input or intrinsic ones whose values would be provided by the scanner and parser. The definition follows the hints of the previous section. Terminal symbols without attributes are omitted. All binary and unary operators are expressed by two nodes having one attribute to represent the operator. To simplify things as much as possible all lists are allowed to be empty and procedure declarations as well as calls always have a parameter list. The specification tries to keep the tree as small as possible. The inheritance mechanism allows to avoid all chain rules. There are no nodes for sequences of declarations, statements, etc.. Instead every node for a declaration or a statement has a field named *Next* describing the successor entity. Except for expressions no separate nodes are used for identifiers. The information is included as attribute in the node types *Proc, Var, Formal*, and *Call*. The source position is stored only at the nodes where it might be needed during semantic analysis. The above measures not only reduce the amount of storage but they also reduce run time because less information has to be produced and processed.

8. Experiences

Ast can be used not only for abstract syntax trees in compilers but for every tree or graph like data structure. In general the data structure can serve as interface between phases within a program or between separate programs. In the latter case it would be communicated via a file using the generated reader and writer procedures.

Generated tree respectively graph modules have successfully been used in compilers e. g. for MiniLAX [WGS89] and UNITY [Bie89] as well as for a Modula to C translator [Mar90]. The modules for the internal data structure of *ast* itself and the attribute evaluator generator *ag* [Gro89] have also been generated by *ast*. Moreover, the symbol table module of the Modula to C translator has been generated.

The advantage of this approach is that it saves considerably hand-coding of trivial declarations and operations. Table 6 lists the sizes (numbers of lines) of some specifications and the generated modules. Sums in the specification column are composed of the sizes for the definition of node types and for user-supplied target code. Sums in the tree module column are composed of the sizes for the definition part and for the implementation part. The reason for the large sizes of the tree modules comes from the numerous node constructor procedures and from the graph browser in the case of *ag*. These procedures proved to be very helpful for debugging purposes as they provide readable output of complex data structures. The constructor procedures allow to write record aggregates. Therefore, node creation and assignment of values to the components can be written very compact. It is even possible to write (ground) terms as in Prolog or LISP by nested calls of the constructor procedures.

Table 6: Examples of Ast Applications

application	specification	tree module
MiniLAX	56	202 + 835 = 1037
Modula-2	240	583 + 3083 = 3666
UNITY	210	207 + 962 = 1169
Ag	78 + 347 = 425	317 + 1317 = 1634
Definition table	82 + 900 = 982	399 + 1431 = 1830

9. Usage

NAME

ast - generator for abstract structure/syntax trees

SYNOPSIS

ast [-options] [-ldir] [file]

DESCRIPTION

Ast generates a program module to handle arbitrary attributed trees and graphs. A typical application is the abstract syntax tree in a compiler. The input *file* contains a specification which describes the structure of all possible trees or nodes respectively and the attributes of the nodes. *Ast* generates type declarations to implement the tree and several procedures for tree manipulation including ASCII and binary readers and writers (see options below). If *file* is omitted the specification is read from standard input.

OPTIONS

a	generate all except -ch (default)	
n	generate node constructors	procedures n<node> (node)
m	generate node constructors	procedures m<node> (make)
f	generate node/tree destroyer	procedure ReleaseTREE (free)
F	generate general destroyer	procedure ReleaseTREEModule (FREE)
o	generate ASCII node writer	procedure WriteTREENode (output)
r	generate ASCII graph reader	procedure ReadTREE
w	generate ASCII graph writer	procedure WriteTREE
g	generate binary graph reader	procedure GetTREE
p	generate binary graph writer	procedure PutTREE
R	generate list reverser	procedure ReverseTREE
t	generate top down traversal (reverse depth first)	procedure TraverseTREETD
b	generate bottom up traversal (depth first)	procedure TraverseTREEBU
y	generate graph copy	procedure CopyTREE
=	generate tree equality test	procedure IsEqualTREE
k	generate graph checker	procedure CheckTREE
q	generate graph browser	procedure QueryTREE
d	generate definition module	
i	generate implementation module	
s	generate import statements	
4	generate tree/graph description in file VIEW.TS	
6	generate # line directives	
7	touch output files only if necessary	
8	report storage consumption	
c	generate C code (default is Modula-2)	
h	print help information	
ldir	*dir* is the directory where ast finds its table files	

FILES

if output is in C:

<module>.h	specification of the generated graph module
<module>.c	body of the generated graph module
yy<module>.w	macro file defining type specific operations

if output is in Modula-2:

<module>.md	definition module of the generated graph module
<module>.mi	implementation module of the generated graph module
<module>.imp	import statements

SEE ALSO

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

J. Grosch: "Tool Support for Data Structures", Structured Programming, 12, 31-38 (1991)

References

[Bie89] F. Bieler, An Interpreter for Chandy/Misra's UNITY, internal paper, GMD Forschungsstelle an der Universität Karlsruhe, 1989.

[DMN70] O. Dahl, B. Myrhaug and K. Nygaard, *SIMULA 67 Common Base Language - Publication S-22*, Norwegian Computing Center, Oslo, 1970.

[Gie88] R. Giegerich, Composition and Evaluation of Attribute Coupled Grammars, *Acta Inf. 25*, (1988), 355-423.

[Gol84] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison Wesley, Reading, MA, 1984.

[Gro87] J. Grosch, Reusable Software - A Collection of Modula-Modules, Compiler Generation Report No. 4, GMD Forschungsstelle an der Universität Karlsruhe, Sep. 1987.

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

[GrV88] J. Grosch and B. Vielsack, The Parser Generators Lalr and Ell, Compiler Generation Report No. 8, GMD Forschungsstelle an der Universität Karlsruhe, Apr. 1988.

[Gro88] J. Grosch, Generators for High-Speed Front-Ends, *LNCS 371*, (Oct. 1988), 81-92, Springer Verlag.

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

[Gro91] J. Grosch, Puma - A Generator for the Transformation of Attributed Trees, Compiler Generation Report No. 26, GMD Forschungsstelle an der Universität Karlsruhe, July 1991.

[HHK88] J. Heering, P. R. H. Hendriks, P. Klint and J. Rekers, *The Syntax Definition Formalism SDF - Reference Manual*, ESPRIT Project GIPE, Dec. 1988.

[Joh75] S. C. Johnson, Yacc — Yet Another Compiler-Compiler, Computer Science Technical Report 32, Bell Telephone Laboratories, Murray Hill, NJ, July 1975.

[Knu68] D. E. Knuth, Semantics of Context-Free Languages, *Mathematical Systems Theory 2*, 2 (June 1968), 127-146.

[Knu71] D. E. Knuth, Semantics of Context-free Languages: Correction, *Mathematical Systems Theory 5*, (Mar. 1971), 95-96.

[Lam87] D. A. Lamb, IDL: Sharing Intermediate Representations, *ACM Trans. Prog. Lang. and Systems 9*, 3 (July 1987), 297-318.

[Mar90] M. Martin, Entwurf und Implementierung eines Übersetzers von Modula-2 nach C, Diplomarbeit, GMD Forschungsstelle an der Universität Karlsruhe, Feb. 1990.

[NNG89] J. R. Nestor, J. M. Newcomer, P. Giannini and D. L. Stone, *IDL: The Language and its Implementation*, Prentice Hall, Englewood Cliffs, 1989.

[Par88] J. C. H. Park, y+: A Yacc Preprocessor for Certain Semantic Actions, *SIGPLAN Notices 23*, 6 (1988), 97-106.

[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.

[Wir88a] N. Wirth, The Programming Language Oberon, *Software—Practice & Experience 18*, 7 (July 1988), 671-690.

[Wir88b] N. Wirth, From Modula to Oberon, *Software—Practice & Experience 18*, 7 (July 1988), 661-670.

[Wir88c] N. Wirth, Type Extensions, *ACM Trans. Prog. Lang. and Systems 10*, 2 (Apr. 1988), 204-214.

Appendix 1: Syntax of the Specification Language

```
RULE                                       /* Ast: concrete syntax */

/* parser grammar */

Specification    = <
                 =               TreeCodes PropPart DeclPart RulePart Modules .
                 = 'MODULE' Name TreeCodes PropPart DeclPart RulePart
                   'END' Name Modules .
> .
TreeCodes        = <
                 =             SubUnit .
                 = 'TREE'      SubUnit Codes .
                 = 'TREE' Name SubUnit Codes .
> .
Codes            = <
                 = .
                 = Codes 'EXPORT' tTargetCode .
                 = Codes 'IMPORT' tTargetCode .
                 = Codes 'GLOBAL' tTargetCode .
                 = Codes 'LOCAL'  tTargetCode .
                 = Codes 'BEGIN'  tTargetCode .
                 = Codes 'CLOSE'  tTargetCode .
> .
SubUnit          = <
                 = .
                 = SubUnit 'SUBUNIT' Name .
                 = SubUnit 'VIEW'    Name .
> .
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            = <
                 = .
   Nonterminal   = Types Name BaseTypes '='  AttrDecls Extensions '.' .
   Terminal      = Types Name BaseTypes ':'  AttrDecls Extensions '.' .
```

```
   Abstract      = Types Name BaseTypes ':=' AttrDecls Extensions '.' .
> .
BaseTypes        = <
                 = .
                 = '<-' Names .
> .
Extensions       = <
                 = .
                 = '<' Types '>' .
> .
AttrDecls        = <
                 = .
   ChildSelct    = AttrDecls     Name ':' Name Properties .
   ChildNoSelct  = AttrDecls              Name Properties .
   AttrTyped     = AttrDecls '[' Name ':' Name Properties ']' .
   AttrInteger   = AttrDecls '[' Name          Properties ']' .
> .
Properties       = <
                 = .
                 = Properties 'INPUT' .
                 = Properties 'OUTPUT' .
                 = Properties 'SYNTHESIZED' .
                 = Properties 'INHERITED' .
                 = Properties 'THREAD' .
                 = Properties 'REVERSE' .
                 = Properties 'IGNORE' .
                 = Properties 'VIRTUAL' .
> .
Modules          = <
                 = .
                 = Modules 'MODULE' Name TreeCodes DeclPart RulePart 'END' Name .
> .
Name             = <
                 = tIdent .
                 = tString .
> .

/* lexical grammar */

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

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

Characters       : <
                 = .
                 = Characters Character .
> .
```

Appendix 2: Concrete Syntax of the Example Language MiniLAX

```
RULE

Prog            = PROGRAM tIdent ';' 'DECLARE' Decls 'BEGIN' Stats 'END' '.' .
Decls           = <
   Decls1       = Decl .
   Decls2       = Decls ';' Decl .
> .
Decl            = <
   Var          = tIdent ':' Type .
   Proc0        = PROCEDURE tIdent ';' 'DECLARE' Decls 'BEGIN' Stats 'END' .
   Proc         = PROCEDURE tIdent '(' Formals ')' ';'
                                       'DECLARE' Decls 'BEGIN' Stats 'END' .
> .
Formals         = <
   Formals1     = Formal .
   Formals2     = Formals ';' Formal .
> .
Formal          = <
   Value        = tIdent ':' Type .
   Ref          = VAR tIdent ':' Type .
> .
Type            = <
   Int          = INTEGER .
   Real         = REAL .
   Bool         = BOOLEAN .
   Array        = ARRAY '[' Lwb: tIntegerConst '..' Upb: tIntegerConst ']' OF Type .
> .
Stats           = <
   Stats1       = Stat .
   Stats2       = Stats ';' Stat .
> .
Stat            = <
   Assign       = Adr ':=' Expr .
   Call0        = tIdent .
   Call         = tIdent '(' Actuals ')' .
   If           = IF Expr THEN Then: Stats ELSE Else: Stats 'END' .
   While        = WHILE Expr DO Stats 'END' .
   Read         = READ '(' Adr ')' .
   Write        = WRITE '(' Expr ')' .
> .
Actuals         = <
   Expr1        = Expr .
   Expr2        = Actuals ',' Expr .
> .
Expr            = <
   Less         = Lop: Expr '<' Rop: Expr .
   Plus         = Lop: Expr '+' Rop: Expr .
   Times        = Lop: Expr '*' Rop: Expr .
   Not          = NOT Expr .
   '()'         = '(' Expr ')' .
   IConst       = tIntegerConst .
   RConst       = tRealConst .
   False        = FALSE .
   True         = TRUE .
   Adr          = <
      Name      = tIdent .
      Index     = Adr '[' Expr ']' .
   > .
> .
tIdent          : [Ident: tIdent] .
tIntegerConst   : [Integer      ] .
tRealConst      : [Real : REAL  ] .
```

Appendix 3: Abstract Syntax of the Example Language MiniLAX

```
TREE                                    /* MiniLAX: abstract syntax */
IMPORT  { FROM Idents IMPORT tIdent;
          TYPE tPosition = RECORD Line, Column: CARDINAL; END; }

GLOBAL  { FROM Idents IMPORT tIdent; }

RULE

MiniLax         = Proc .
Decls           = <
   NoDecl       = .
   Decl         = Next: Decls REV [Ident: tIdent] [Pos: tPosition] <
      Proc      = Formals Decls Stats .
      Var       = Type .
   > .
> .
Formals         = <
   NoFormal     = .
   Formal       = Next: Formals REV [Ident: tIdent] [Pos: tPosition] Type .
> .
Type            = <
   Integer      = .
   Real         = .
   Boolean      = .
   Array        = Type                [Lwb] [Upb] [Pos: tPosition] .
   Ref          = Type .
> .
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] .
   Actual       = Next: Actuals REV Expr .
> .
Expr            =                     [Pos: tPosition] <
   Binary       = Lop: Expr Rop: Expr [Operator] .
   Unary        = Expr                [Operator] .
   IntConst     =                     [Value         ] .
   RealConst    =                     [Value: REAL   ] .
   BoolConst    =                     [Value: BOOLEAN] .
   Adr          = <
      Index     = Adr Expr .
      Ident     =                     [Ident: tIdent] .
   > .
> .
```

Appendix 4: Generated Header File for C

```
# ifndef yyTREE   /* throughout replace TREE by the name of the tree module */
# define yyTREE
<import_declarations>
# define bool char
# define NoTREE (tTREE) NULL
# define k<type_1> 1
# define k<type_1> 2
   ...
typedef unsigned short TREE_tKind;   /* or char */
typedef unsigned short TREE_tMark;
typedef unsigned short TREE_tLabel;
typedef union TREE_Node * tTREE;
typedef void (* TREE_tProcTree) ();
<export_declarations>
# ifndef TREE_NodeHead
# define TREE_NodeHead
# endif
typedef struct { TREE_tKind yyKind; TREE_tMark yyMark; TREE_NodeHead } TREE_tNodeHead;
typedef struct { TREE_tNodeHead yyHead;
                 <children_and_attributes_of_type_1> } y<type_1>;
typedef struct { TREE_tNodeHead yyHead;
                 <children_and_attributes_of_type_2> } y<type_2>;
   ...
union TREE_Node {
   TREE_tKind Kind;
   TREE_tNodeHead yyHead;
   y<type_1> <type_1>;
   y<type_2> <type_2>;
    ...
};
extern tTREE TREERoot;
extern unsigned long TREE_HeapUsed;
extern unsigned short TREE_NodeSize [];
extern char * TREE_NodeName [];
extern tTREE n<type_1>          ();
extern tTREE n<type_2>          ();
   ...
extern tTREE m<type_1>          (<input_children_and_attributes_of_type_1>);
extern tTREE m<type_2>          (<input_children_and_attributes_of_type_2>);
   ...
extern tTREE MakeTREE           (TREE_tKind Kind);
extern bool  TREE_IsType        (tTREE t, TREE_tKind Kind);
extern void  ReleaseTREE        (tTREE t);
extern void  ReleaseTREEModule  ();
extern void  WriteTREENode      (FILE * f, tTREE t);
extern void  WriteTREE          (FILE * f, tTREE t);
extern tTREE ReadTREE           (FILE * f);
extern void  PutTREE            (FILE * f, tTREE t);
extern tTREE GetTREE            (FILE * f);
extern void  TraverseTREETD     (tTREE t, void (* Procedure) (tTREE t));
extern void  TraverseTREEBU     (tTREE t, void (* Procedure) (tTREE t));
extern tTREE ReverseTREE        (tTREE t);
extern tTREE CopyTREE           (tTREE t);
extern bool  CheckTREE          (tTREE t);
extern void  QueryTREE          (tTREE t);
extern bool  IsEqualTREE        (tTREE t1, tTREE t2);
extern void  BeginTREE          ();
extern void  CloseTREE          ();
# endif
```

Appendix 5: Generated Definition Module for Modula-2

```
DEFINITION MODULE TREE;
   IMPORT IO;     (* throughout replace TREE by the name of the tree module *)
   <import_declarations>
   CONST
      NoTREE       = NIL;
      <type_1>     = 1;
      <type_2>     = 2;
      ...
   TYPE
      tTREE        = POINTER TO yyNode;
      tProcTree    = PROCEDURE (tTREE);
   <export_declarations>
   TYPE
      yytNodeHead  = RECORD yyKind, yyMark: SHORTCARD; END;
   TYPE
      y<type_1>    = RECORD yyHead: yytNodeHead;
                            <children_and_attributes_of_type_1> END;
      y<type_2>    = RECORD yyHead: yytNodeHead;
                            <children_and_attributes_of_type_2> END;
      ...
      yyNode       = RECORD
         CASE : SHORTCARD OF
         | 0        : Kind: SHORTCARD;
         | ...      : yyHead: yytNodeHead;
         | <type_1> : <type_1>    : y<type_1>;
         | <type_2> : <type_2>    : y<type_2>;
         ...
         END;
      END;
   VAR TREERoot  : tTREE;
   VAR HeapUsed  : LONGCARD;
   VAR yyNodeSize: ARRAY [ ... ] OF SHORTCARD;
   PROCEDURE n<type_1>      (): tTREE;
   PROCEDURE n<type_2>      (): tTREE;
   ...
   PROCEDURE m<type_1>      (<input_children_and_attributes_of_type_1>): tTREE;
   PROCEDURE m<type_2>      (<input_children_and_attributes_of_type_2>): tTREE;
   ...
   PROCEDURE MakeTREE       (Kind: SHORTCARD): tTREE;
   PROCEDURE IsType         (Tree: tTREE; Kind: SHORTCARD): BOOLEAN;
   PROCEDURE ReleaseTREE    (Tree: tTREE);
   PROCEDURE ReleaseTREEModule;
   PROCEDURE WriteTREENode  (f: IO.tFile; Tree: tTREE);
   PROCEDURE WriteTREE      (f: IO.tFile; Tree: tTREE);
   PROCEDURE ReadTREE       (f: IO.tFile): tTREE;
   PROCEDURE PutTREE        (f: IO.tFile; Tree: tTREE);
   PROCEDURE GetTREE        (f: IO.tFile): tTREE;
   PROCEDURE ReverseTREE    (Tree: tTREE): tTREE;
   PROCEDURE TraverseTREETD (Tree: tTREE; Proc: tProcTree);
   PROCEDURE TraverseTREEBU (Tree: tTREE; Proc: tProcTree);
   PROCEDURE CopyTREE       (Tree: tTREE): tTree;
   PROCEDURE CheckTREE      (Tree: tTREE): BOOLEAN;
   PROCEDURE QueryTREE      (Tree: tTREE);
   PROCEDURE IsEqualTREE    (Tree1, Tree2: tTREE): BOOLEAN;
   PROCEDURE BeginTREE;
   PROCEDURE CloseTREE;
END TREE.
```

Appendix 6: Predefined Type Operations for C

```
/* int */
# define beginint(a)
# define closeint(a)
# define readint(a)              (void) fscanf (yyf, "%d", & a);
# define writeint(a)             (void) fprintf (yyf, "%d", a);
# define getint(a)               yyGet ((char *) & a, sizeof (a));
# define putint(a)               yyPut ((char *) & a, sizeof (a));
# define copyint(a, b)
# define equalint(a, b)          a == b
/* short */
# define beginshort(a)
# define closeshort(a)
# define readshort(a)            (void) fscanf (yyf, "%hd", & a);
# define writeshort(a)           (void) fprintf (yyf, "%hd", a);
# define getshort(a)             yyGet ((char *) & a, sizeof (a));
# define putshort(a)             yyPut ((char *) & a, sizeof (a));
# define copyshort(a, b)
# define equalshort(a, b)        a == b
/* long */
# define beginlong(a)
# define closelong(a)
# define readlong(a)             (void) fscanf (yyf, "%ld", & a);
# define writelong(a)            (void) fprintf (yyf, "%ld", a);
# define getlong(a)              yyGet ((char *) & a, sizeof (a));
# define putlong(a)              yyPut ((char *) & a, sizeof (a));
# define copylong(a, b)
# define equallong(a, b)         a == b
/* unsigned */
# define beginunsigned(a)
# define closeunsigned(a)
# define readunsigned(a)         (void) fscanf (yyf, "%u", & a);
# define writeunsigned(a)        (void) fprintf (yyf, "%u", a);
# define getunsigned(a)          yyGet ((char *) & a, sizeof (a));
# define putunsigned(a)          yyPut ((char *) & a, sizeof (a));
# define copyunsigned(a, b)
# define equalunsigned(a, b)     a == b
/* float */
# define beginfloat(a)
# define closefloat(a)
# define readfloat(a)            (void) fscanf (yyf, "%g", & a);
# define writefloat(a)           (void) fprintf (yyf, "%g", a);
# define getfloat(a)             yyGet ((char *) & a, sizeof (a));
# define putfloat(a)             yyPut ((char *) & a, sizeof (a));
# define copyfloat(a, b)
# define equalfloat(a, b)        a == b
/* double */
# define begindouble(a)
# define closedouble(a)
# define readdouble(a)           (void) fscanf (yyf, "%lg", & a);
# define writedouble(a)          (void) fprintf (yyf, "%lg", a);
# define getdouble(a)            yyGet ((char *) & a, sizeof (a));
# define putdouble(a)            yyPut ((char *) & a, sizeof (a));
# define copydouble(a, b)
# define equaldouble(a, b)       a == b
/* bool */
# define beginbool(a)
# define closebool(a)
# define readbool(a)             a = fgetc (yyf) == 'T';
```

```
# define writebool(a)           (void) fputc (a ? 'T' : 'F', yyf);
# define getbool(a)             yyGet ((char *) & a, sizeof (a));
# define putbool(a)             yyPut ((char *) & a, sizeof (a));
# define copybool(a, b)
# define equalbool(a, b)        a == b
/* char */
# define beginchar(a)
# define closechar(a)
# define readchar(a)            a = fgetc (yyf);
# define writechar(a)           (void) fputc (a, yyf);
# define getchar(a)             yyGet ((char *) & a, sizeof (a));
# define putchar(a)             yyPut ((char *) & a, sizeof (a));
# define copychar(a, b)
# define equalchar(a, b)        a == b
/* tString */
# define begintString(a)
# define closetString(a)
# define readtString(a)
# define writetString(a)        (void) fputs (a, yyf);
# define gettString(a)
# define puttString(a)
# define copytString(a, b)
# define equaltString(a, b)     strcmp (a, b)
/* tStringRef */
# define begintStringRef(a)
# define closetStringRef(a)
# define readtStringRef(a)
# define writetStringRef(a)     WriteString (yyf, a);
# define gettStringRef(a)
# define puttStringRef(a)
# define copytStringRef(a, b)
# define equaltStringRef(a, b)  a == b
/* tIdent */
# define begintIdent(a)
# define closetIdent(a)
# define readtIdent(a)          a = yyReadIdent ();
# define writetIdent(a)         WriteIdent (yyf, a);
# define gettIdent(a)           yyGetIdent (& a);
# define puttIdent(a)           yyPutIdent (a);
# define copytIdent(a, b)
# define equaltIdent(a, b)      a == b
/* tSet */
# define begintSet(a)
# define closetSet(a)
# define readtSet(a)            ReadSet (yyf, & a);
# define writetSet(a)           WriteSet (yyf, & a);
# define gettSet(a)
# define puttSet(a)
# define copytSet(a, b)
# define equaltSet(a, b)        IsEqual (& a, & b)
/* tPosition */
# define begintPosition(a)
# define closetPosition(a)
# define readtPosition(a)
# define writetPosition(a)      WritePosition (yyf, a);
# define gettPosition(a)
# define puttPosition(a)
# define copytPosition(a, b)
# define equaltPosition(a, b)   Compare (a, b) == 0
```

Appendix 7: Predefined Type Operations for Modula-2

```
(* INTEGER *)
# define beginINTEGER(a)
# define closeINTEGER(a)
# define readINTEGER(a)          a := IO.ReadI (yyf);
# define writeINTEGER(a)         IO.WriteI (yyf, a, 0);
# define getINTEGER(a)           yyGet (a);
# define putINTEGER(a)           yyPut (a);
# define copyINTEGER(a, b)
# define equalINTEGER(a, b)      a = b
(* SHORTINT *)
# define beginSHORTINT(a)
# define closeSHORTINT(a)
# define readSHORTINT(a)         a := IO.ReadI (yyf);
# define writeSHORTINT(a)        IO.WriteI (yyf, a, 0);
# define getSHORTINT(a)          yyGet (a);
# define putSHORTINT(a)          yyPut (a);
# define copySHORTINT(a, b)
# define equalSHORTINT(a, b)     a = b
(* LONGINT *)
# define beginLONGINT(a)
# define closeLONGINT(a)
# define readLONGINT(a)          a := IO.ReadI (yyf);
# define writeLONGINT(a)         IO.WriteI (yyf, a, 0);
# define getLONGINT(a)           yyGet (a);
# define putLONGINT(a)           yyPut (a);
# define copyLONGINT(a, b)
# define equalLONGINT(a, b)      a = b
(* CARDINAL *)
# define beginCARDINAL(a)
# define closeCARDINAL(a)
# define readCARDINAL(a)         a := IO.ReadI (yyf);
# define writeCARDINAL(a)        IO.WriteI (yyf, a, 0);
# define getCARDINAL(a)          yyGet (a);
# define putCARDINAL(a)          yyPut (a);
# define copyCARDINAL(a, b)
# define equalCARDINAL(a, b)     a = b
(* SHORTCARD *)
# define beginSHORTCARD(a)
# define closeSHORTCARD(a)
# define readSHORTCARD(a)        a := IO.ReadI (yyf);
# define writeSHORTCARD(a)       IO.WriteI (yyf, a, 0);
# define getSHORTCARD(a)         yyGet (a);
# define putSHORTCARD(a)         yyPut (a);
# define copySHORTCARD(a, b)
# define equalSHORTCARD(a, b)    a = b
(* LONGCARD *)
# define beginLONGCARD(a)
# define closeLONGCARD(a)
# define readLONGCARD(a)         a := IO.ReadI (yyf);
# define writeLONGCARD(a)        IO.WriteI (yyf, a, 0);
# define getLONGCARD(a)          yyGet (a);
# define putLONGCARD(a)          yyPut (a);
# define copyLONGCARD(a, b)
# define equalLONGCARD(a, b)     a = b
(* REAL *)
# define beginREAL(a)
# define closeREAL(a)
# define readREAL(a)             a := IO.ReadR (yyf);
# define writeREAL(a)            IO.WriteR (yyf, a, 0, 6, 1);
```

```
# define getREAL(a)              yyGet (a);
# define putREAL(a)              yyPut (a);
# define copyREAL(a, b)
# define equalREAL(a, b)         a = b
(* LONGREAL *)
# define beginLONGREAL(a)
# define closeLONGREAL(a)
# define readLONGREAL(a)         a := IO.ReadR (yyf);
# define writeLONGREAL(a)        IO.WriteR (yyf, a, 0, 6, 1);
# define getLONGREAL(a)          yyGet (a);
# define putLONGREAL(a)          yyPut (a);
# define copyLONGREAL(a, b)
# define equalLONGREAL(a, b)     a = b
(* BOOLEAN *)
# define beginBOOLEAN(a)
# define closeBOOLEAN(a)
# define readBOOLEAN(a)          a := IO.ReadB (yyf);
# define writeBOOLEAN(a)         IO.WriteB (yyf, a);
# define getBOOLEAN(a)           yyGet (a);
# define putBOOLEAN(a)           yyPut (a);
# define copyBOOLEAN(a, b)
# define equalBOOLEAN(a, b)      a = b
(* CHAR *)
# define beginCHAR(a)
# define closeCHAR(a)
# define readCHAR(a)             a := IO.ReadC (yyf);
# define writeCHAR(a)            IO.WriteC (yyf, a);
# define getCHAR(a)              yyGet (a);
# define putCHAR(a)              yyPut (a);
# define copyCHAR(a, b)
# define equalCHAR(a, b)         a = b
(* BITSET *)
# define beginBITSET(a)
# define closeBITSET(a)
# define readBITSET(a)           yyReadHex (a);
# define writeBITSET(a)          yyWriteHex (a);
# define getBITSET(a)            yyGet (a);
# define putBITSET(a)            yyPut (a);
# define copyBITSET(a, b)
# define equalBITSET(a, b)       a = b
(* BYTE *)
# define beginBYTE(a)
# define closeBYTE(a)
# define readBYTE(a)             yyReadHex (a);
# define writeBYTE(a)            yyWriteHex (a);
# define getBYTE(a)              yyGet (a);
# define putBYTE(a)              yyPut (a);
# define copyBYTE(a, b)
# define equalBYTE(a, b)         a = b
(* WORD *)
# define beginWORD(a)
# define closeWORD(a)
# define readWORD(a)             yyReadHex (a);
# define writeWORD(a)            yyWriteHex (a);
# define getWORD(a)              yyGet (a);
# define putWORD(a)              yyPut (a);
# define copyWORD(a, b)
# define equalWORD(a, b)         a = b
(* ADDRESS *)
# define beginADDRESS(a)
# define closeADDRESS(a)
```

```
# define readADDRESS(a)          yyReadHex (a);
# define writeADDRESS(a)         yyWriteHex (a);
# define getADDRESS(a)           yyGet (a);
# define putADDRESS(a)           yyPut (a);
# define copyADDRESS(a, b)
# define equalADDRESS(a, b)      a = b
(* tString *)
# define begintString(a)
# define closetString(a)
# define readtString(a)          Strings.ReadL (yyf, a);
# define writetString(a)         Strings.WriteL (yyf, a);
# define gettString(a)           yyGet (a);
# define puttString(a)           yyPut (a);
# define copytString(a, b)
# define equaltString(a, b)      Strings.IsEqual (a, b)
(* tStringRef *)
# define begintStringRef(a)
# define closetStringRef(a)
# define readtStringRef(a)
# define writetStringRef(a)      StringMem.WriteString (yyf, a);
# define gettStringRef(a)
# define puttStringRef(a)
# define copytStringRef(a, b)
# define equaltStringRef(a, b)   a = b
(* tIdent *)
# define begintIdent(a)
# define closetIdent(a)
# define readtIdent(a)           a := yyReadIdent ();
# define writetIdent(a)          Idents.WriteIdent (yyf, a);
# define gettIdent(a)            yyGetIdent (a);
# define puttIdent(a)            yyPutIdent (a);
# define copytIdent(a, b)
# define equaltIdent(a, b)       a = b
(* tText *)
# define begintText(a)
# define closetText(a)
# define readtText(a)
# define writetText(a)           Texts.WriteText (yyf, a);
# define gettText(a)
# define puttText(a)
# define copytText(a, b)
# define equaltText(a, b)        FALSE
(* tSet *)
# define begintSet(a)
# define closetSet(a)
# define readtSet(a)             Sets.ReadSet (yyf, a);
# define writetSet(a)            Sets.WriteSet (yyf, a);
# define gettSet(a)
# define puttSet(a)
# define copytSet(a, b)
# define equaltSet(a, b)         Sets.IsEqual (a, b)
(* tRelation *)
# define begintRelation(a)
# define closetRelation(a)
# define readtRelation(a)        Relations.ReadRelation (yyf, a);
# define writetRelation(a)       Relations.WriteRelation (yyf, a);
# define gettRelation(a)
# define puttRelation(a)
# define copytRelation(a, b)
# define equaltRelation(a, b)    Relations.IsEqual (a, b)
(* tPosition *)
```

```
# define begintPosition(a)
# define closetPosition(a)
# define readtPosition(a)
# define writetPosition(a)      Positions.WritePosition (yyf, a);
# define gettPosition(a)
# define puttPosition(a)
# define copytPosition(a, b)
# define equaltPosition(a, b)   Positions.Compare (a, b) = 0
```

Contents