Specification of a MiniLAX-Interpreter

J. Grosch

GESELLSCHAFT FÜR MATHEMATIK
UND DATENVERARBEITUNG MBH

FORSCHUNGSSTELLE FÜR
PROGRAMMSTRUKTUREN
AN DER UNIVERSITÄT KARLSRUHE

Project

Compiler Generation

Specification of a MiniLAX-Interpreter

Josef Grosch

Nov. 22, 1991

Report No. 22

Copyright © 1991 GMD

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

Specification of a MiniLAX-Interpreter

J. Grosch
GMD Forschungsstelle an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1, D-7500 Karlsruhe, Germany
Tel: +721-6622-26
E-Mail: grosch@karlsruhe.gmd.de

1. Introduction

This paper describes the specification of a MiniLAX interpreter using the following tools from the GMD Toolbox for Compiler Construction [GrE90]: The scanner generator *rex* [Gro87b], the parser generator *lalr* [GrV88], the generator for abstract syntax trees *ast* [Gro91a], the attribute evaluator generator *ag* [Gro89], and the generator for the transformation of attributed trees *puma* [Gro91b]. The target language is Modula-2. The compiler parts which are not generated by the tools are either taken from a library of reusable modules [Gro87a] or are provided as hand-written Modula-2 code.

The rest of this report is organized as follows: Section 2 defines the source language MiniLAX. Section 3 defines the intermediate language ICode which is the input of the interpreter. Section 4 explains the structure of the generated compiler. Section 5 contains the specifications for the compiler.

2. MiniLAX

The programming language MiniLAX (Mini LAnguage eXample) is a Pascal relative. To be more specific it is a subset of the example language LAX [WaG84], which is used to illustrate problems in compiler construction. MiniLAX contains a carefully selected set of language concepts:

- types
- type coercion
- overloaded operators
- arrays
- procedures
- reference and value parameters
- nested scopes

Concepts with a low didactical value and concepts that would make the language unnecessary complex have been left out, along with ‘‘syntactic sugar’’.

2.1. Summary of the Language

A computer program consists of two essential parts, a description of *actions* which are to be performed, and a description of the *data*, which are manipulated by these actions. Actions are described by *statements*, and data are described by *declarations*.

The data are represented by constants and values of *variables*. Every variable occurring in a statement must be introduced by a *variable declaration* which associates an identifier and a data type with that variable. The *data type* essentially defines the set of values which may be assumed by that variable. The data type is directly described in the variable declaration.

There exist three *basic types*: `Boolean`, `integer`, and `real`. The values of the type Boolean are denoted by reserved identifiers, the numeric values are denoted by numbers.

Array types are defined by describing the types of their components and an integer range. A component of an array value is selected by an integer *index*. The type of the component is the component type of the corresponding array type.

The most fundamental statement is the *assignment statement*. It specifies that a newly computed value be assigned to a variable (or a component of a variable). The value is obtained by evaluating an *expression*. Expressions consist of variables, constants and operators operating on the denoted quantities and producing new values. MiniLAX defines a fixed set of operators, each of which can be regarded as describing a mapping from the operand types into the result type. The set of operators is subdivided into

Arithmetic operators: addition and multiplication
Boolean operators: negation
Relational operators: comparison

The result of a comparison is of type `Boolean`. The *procedure statement* causes the execution of the designated procedure (see below). Assignment and procedure statements are the components or building blocks of *structured statements*, which specify sequential, selective, or repeated execution of their components. Sequential execution of statements is specified by *statement sequences*, selective execution by the *if statement*, and repeated execution by the *while statement*. The if statement serves to make the execution of two alternative statements dependent on the value of a Boolean expression. The while statement serves to execute a statement while a Boolean expression is true.

A statement sequence can be given a name (identifier), and be referenced through that identifier. The statement sequence is then called a *procedure*, and its declaration a *procedure declaration*. Such a declaration may additionally contain a set of variable declarations and further procedure declarations. The variables and procedures thus declared can be referenced only within the procedure itself, and are therefore called *local* to the procedure. Their identifiers have significance only within the program text which constitutes the procedure declaration and which is called the *scope* of these identifiers. Since procedures may be declared local to other procedures, scopes may be nested. Entities which are declared in the main program, i.e. not local to some procedure, are called *global*. A procedure has a fixed number of parameters, each of which is denoted within the procedure by an identifier called the *formal parameter*. Upon an activation of the procedure statement, an actual quantity has to be indicated for each parameter which can be referenced from within the procedure through the formal parameter. This quantity is called the *actual parameter*. There are two kinds of parameters: value parameters and variable parameters. In the first case, the actual parameter is an expression which is evaluated once. The formal parameter represents a local variable to which the result of this evaluation is assigned before the execution of the procedure. In the case of a variable parameter, the actual parameter is a variable and the formal parameter stands for this variable. Possible indices are evaluated before execution of the procedure.

2.2. Notation, Terminology, and Vocabulary

The syntax is described in extended Backus-Naur form. Syntactic constructs are denoted by (abbreviated) English words consisting of upper and lower case letters, and containing at least one lower-case letter. The angular brackets < and > are omitted. Strings of letters consisting solely of upper-case letters stand for themselves, i.e. for reserved identifiers of the language. Strings of characters enclosed in single quotes ' ' are also to be taken literally. Square brackets [ ] denote optional constructs. Curly brackets { } stand for zero or more repetitions of the enclosed construct. Alternative constructs are separated by a vertical bar |. Parentheses ( ) are used for grouping.

The basic vocabulary of MiniLAX consists of basic symbols classified into delimiters, identifiers and constants.

Spaces, line ends, and comments may occur anywhere in a program except within a basic symbol. At least one space, line end or comment must occur between any two adjacent identifiers or constants. Otherwise, spaces, line ends, and comments do not influence the meaning of a program.

A *comment* has the form

`'(*'` any sequence of characters not containing “*)” `'*)'`

2.2.1. Delimiters Delimiters are reserved identifiers or (strings of) special characters.

Delim ::= `':'` | `';'` | `':='` | `'('` | `')'` | `'.'` | `','`
| `'..'` | `'['` | `']'` | `'+'` | `'*'` | `'<'`
| `'ARRAY'` | `'BEGIN'` | `'BOOLEAN'` | `'DECLARE'` | `'DO'`
| `'ELSE'` | `'END'` | `'FALSE'` | `'IF'` | `'INTEGER'` | `'NOT'`
| `'OF'` | `'PROCEDURE'` | `'PROGRAM'` | `'READ'` | `'REAL'`
| `'THEN'` | `'TRUE'` | `'VAR'` | `'WHILE'` | `'WRITE'`

2.2.2. Identifiers Identifiers serve to denote variables and procedures. Their association must be unique within their scope of validity, i.e. within the procedure or function in which they are declared.

Id ::= *Letter* { *Letter* | *Digit* }

All letters and digits of an identifier are significant. Upper and lower case letters are distinguished. Delimiters are reserved identifiers that can not be used otherwise.

2.2.3. Numbers The usual decimal notation is used for numbers, which are the constants of the data types `integer` and `real`. The letter ’E’ preceding the scale factor is pronounced as ‘‘’times’’10

IntConst ::= *Digit* { *Digit* }
RealConst ::= [*IntConst*] `'.'` *IntConst* [*ScaleFactor*]
ScaleFactor ::= `'E'` [`'+'` | `'-'`] *IntConst*

Examples:

```
1   100   .1  87.35E-8
```

2.3. Data Types

A data type determines the set of values which variables of that type may assume.

Type ::= *SimpleType* | *ArrayType*

2.3.1. Simple Types

SimpleType ::= `'INTEGER'` | `'REAL'` | `'BOOLEAN'`

The values of type INTEGER are a subset of the whole numbers defined by individual implementations. Its values are the integers.

The values of type REAL are a subset of the real numbers depending on a particular implementation. The values are denoted by real numbers.

The values of type BOOLEAN are the truth values denoted by the reserved identifiers TRUE and FALSE.

2.3.2. Array Types An array type is a structure consisting of a fixed number of components which are all of the same type, called the *component type*. The elements of the array are designated by integer *indices*. The array type specifies the component type as well as a subrange of the integers to be used as indices.

ArrayType ::= `'ARRAY'` `'['` *IntConst* `'..'` *IntConst* `']'` `'OF'` *Type*

Examples:

```
ARRAY [1..100] OF INTEGER
ARRAY [4..7] OF ARRAY [2..2] OF BOOLEAN
```

The index range must contain at least one element, i.e. the lower bound of an index range must not exceed the upper bound.

2.4. Declarations and Denotations of Variables

Variable declarations consist of an identifier denoting the new variable, followed by its type.

VarDecl ::= *Id* `':'` *Type*

Examples:

```
i: INTEGER
r: REAL
b: BOOLEAN
a: ARRAY [4..7] OF ARRAY [2..2] OF INTEGER
```

Denotations of variables either designate an entire variable or a component of an array variable. Variables occurring in examples in subsequent chapters are assumed to be declared as indicated above.

Var ::= *Id* | *Var* `'['` *Expr* `']'`

Examples:

```
i   a[4][2]
```

An entire variable is denoted by its identifier. A component of an array variable is denoted by the variable followed by an index expression. The value of the index expression must lie in the range of the indices of the corresponding array type.

2.5. Expressions

Expressions are constructs denoting rules of computation for obtaining values of variables and generating new values by the application of operators. Expressions consist of operators and operands, i.e. variables and constants.

The rules of composition specify operator *precedences* according to four classes of operators. The operator NOT has the highest precedence, followed by the multiplying operator '*', the adding operator '+', and finally, with the lowest precedence, the relational operator Sequences of operators of the same precedence are executed from left to right.

Expr ::= *Expr* (`'+'` | `'*'` | `'<'`) *Expr* | `'NOT'` *Expr*
| `'('` *Expr* `')'` | *Var* | *IntConst* | *RealConst*
| `'TRUE'` | `'FALSE'`

Examples:

```
i        15        TRUE        2*(i+r)       NOT b        NOT (i<1)
```

The operators are summarized in the following table:

Table of Operators					
Priority	Operator	left Operand	right Operand	Result	Operation
4	`NOT`		BOOLEAN	BOOLEAN	negation
3	`*`	INTEGER REAL	INTEGER REAL	INTEGER REAL	integer multiplication real multiplication
2	+	INTEGER REAL	INTEGER REAL	INTEGER REAL	integer addition real addition
1	<	INTEGER REAL BOOLEAN	INTEGER REAL BOOLEAN	BOOLEAN BOOLEAN BOOLEAN	integer comparison real comparison boolean comparison

Note that, for Boolean values, FALSE < TRUE.

2.6. Statements

Statements denote algorithmic actions, and are said to be *executable*.

Stat ::= *AssignStat* | *CondStat* | *LoopStat* | *ProcStat*

2.6.1. Statement sequences A statement sequence specifies that its component statements are to be executed in the same sequence as they are written.

StatSeq ::= *Stat* { `';'` *Stat* }

2.6.2. Assignment Statements The assignment statement serves to replace the current value of a variable by a new value specified as an expression.

AssignStat ::= *Var* `':='` *Expr*

Examples:

```
i := i+1
r := r*3.141592
b := i<1
a[4][2] := r
```

The variable and the expression must be of identical type, with the following exception being permitted: The type of the variable is REAL, and the type of the expression is INTEGER. In any case, the variable must be of a simple type.

2.6.3. Procedure Statements A procedure statement serves to execute the procedure denoted by the procedure identifier. The procedure statement may contain a list of *actual parameters* which are substituted in place of their corresponding *formal parameters* defined in the procedure declaration. The correspondence is established by the positions of the parameters in the lists of actual and formal parameters respectively. There exist two kinds of parameters: value parameters and variable parameters.

In the case of a *value parameter*, the actual parameter must be an expression (of which a variable is a simple case). The corresponding formal parameter represents a local variable of the called procedure, and the current value of the expression is initially assigned to this variable. Value parameters must have a simple type. In the case of a *variable parameter*, the actual parameter must be a variable of the same type, and the corresponding formal parameter represents this actual variable during the entire execution of the procedure. If this variable is a component of an array, its index is evaluated when the procedure is called. A variable parameter must be used whenever the parameter represents a result of the procedure.

ProcStat ::= *Id* [`'('` *Expr* { `','` *Expr* } `')'`]

Examples:

```
next   Transpose(a,m,n)
```

2.6.4. Conditional Statements The if statement specifies that a statement be executed only if a certain condition (Boolean expression) is true. If it is false, the statement following the delimiter ELSE is to be executed.

CondStat ::= `'IF'` *Expr* `'THEN'` *StatSeq* `'ELSE'` *StatSeq* `'END'`

Examples:

```
IF i < 0 THEN i := 1 ELSE i := 2 END
```

The expression between the delimiters `IF` and `THEN` must be of type Boolean.

2.6.5. Repetitive Statements The while statement specifies that a certain statement is to be executed repeatedly.

LoopStat ::= `'WHILE'` *Expr* `'DO'` *StatSeq* `'END'`

The expression controlling repetition must be of type Boolean. The statement is repeatedly executed as long as the expression is true. If it evaluates to false at the beginning, the statement is not executed at all. The while statement

```
WHILE b DO s END
```

is equivalent to

```
IF b
THEN s; WHILE b DO s END
ELSE (* nothing *)
END
```

Examples:

```
WHILE a [i] < r DO i := i + 1 END

WHILE i < n DO
  r := 2 * r;
  i := i + 1
END
```

2.6.6. Procedure Declarations Procedure declarations serve to define parts of programs and to associate identifiers with them so that they can be activated by procedure statements.

ProcDecl ::= *ProcHead* `';'` *Block*
Block ::= `'DECLARE'` *Decl* { `';'` *Decl* } `'BEGIN'` *StatSeq* `'END'`
Decl ::= *VarDecl* | *ProcDecl*

The *procedure heading* specifies the identifier naming the procedure and the formal parameter identifiers (if any). The parameters are either value or variable parameters.

ProcHead ::= `'PROCEDURE'` *Id* [`'('` *Formal* { `';'` *Formal* } `')'`]
Formal ::= [`'VAR'`] *Id* `':'` *Type*

If a formal starts with the delimiter VAR it specifies a variable parameter, otherwise a value parameter.

The statement sequence of the block specifies the algorithmic actions to be executed upon an activation of the procedure by a procedure statement.

All identifiers introduced in the formal parameter part of the procedure heading and in the declaration part of the associated block are *local* to the procedure declaration which is called the *scope* of these identifiers. They are not known outside their scope. In the case of local variables, their values are undefined at the beginning of the statement part.

The use of the procedure identifier in a procedure statement within its declaration implies recursive execution of the procedure.

Examples of procedure declarations:

```
PROCEDURE ReadPosInteger (VAR i: INTEGER);
DECLARE
  j: INTEGER;
BEGIN
  i := 0;
  WHILE NOT (0 < i) DO READ (i) END
END

PROCEDURE Sort (VAR a: ARRAY [1..10] OF REAL; n: INTEGER);
DECLARE
  i: INTEGER; j: INTEGER; k: INTEGER; h: REAL;
BEGIN
  i := 1;
  WHILE i < n DO   (* a [1], ... , a [i] is sorted *)
    j := i; k := i;
    WHILE j < n DO   (* a [k] = min {a [i], ... , a [j]} *)
      j := j + 1;
      IF a [j] < a [k] THEN k := j ELSE k := k END
    END;
    h := a [i]; a [i] := a [k]; a [k] := h;
    i := i + 1
  END
END
```

2.7. Input and Output

Input and output of values of simple types is achieved by the standard procedures READ and WRITE.

The procedure READ takes one actual parameter which must be a variable of a simple type. It reads a value of the corresponding type from the standard input and assigns it to that variable.

The procedure WRITE takes one actual parameter which must be an expression with a simple type. It writes the value of that expression onto the standard output.

Example:

```
(* read integers and write until a nonpositive number is read *)
READ (i);
WHILE 0 < i DO
  WRITE (i); READ (i)
END
```

2.8. Programs

A MiniLAX program has the form of a procedure declaration except for its heading.

Program ::= `'PROGRAM'` *Id* `';'` *Block* `'.'`

The identifier following the symbol PROGRAM is the program name; it has no further significance inside the program.

Example:

```
PROGRAM test;

  (* read, sort and write an array of n numbers            *)
  (* this program shows the following features:            *)
  (*   procedure calls from main level, to a local, and to *)
  (*     a global procedure                                *)
  (*   access to a global array                            *)
  (*   access to local, global and intermediate variables  *)
  (*   recursion                                           *)
  (*   reading and writing of all types                    *)
  (*   integer to real conversion                          *)

DECLARE
  test : BOOLEAN;
  n    : INTEGER;
  a    : ARRAY [1..100] OF REAL;

  PROCEDURE skip; (* do nothing *)
  DECLARE
    n: INTEGER
  BEGIN
    n := n
  END;

  PROCEDURE read (VAR n: INTEGER; VAR a: ARRAY [1..100] OF REAL);
  DECLARE
    i: INTEGER
  BEGIN
    WRITE (TRUE); READ (test);
    WRITE (5); READ (n);
    i := 1;
    WHILE i < n DO
      i := i + 1; WRITE (1.0E-7); READ (a [i])
    END
  END;

  PROCEDURE write (m: INTEGER); (* write a [m..n] *)
  DECLARE
    x: INTEGER
  BEGIN
    WRITE (a [m]);
    IF m < n THEN write (m + 1) ELSE skip END
  END;

  PROCEDURE sort (VAR a: ARRAY [1..100] OF REAL); (* sort a [1..n] *)
  DECLARE
    i : INTEGER;
    j : INTEGER;
    k : INTEGER;
    h : REAL;
    ok: BOOLEAN;
```

```
    PROCEDURE check (VAR ok: BOOLEAN); (* check order of a [1..n] *)
    DECLARE
      continue: BOOLEAN
    BEGIN
      IF test THEN write (1) ELSE skip END;
      i := 1; continue := TRUE;
      WHILE continue DO
        IF i < n THEN
          continue := NOT (a [i + 1] < a [i]);
          IF continue THEN i := i + 1 ELSE skip END
        ELSE
          continue := FALSE
        END
      END;
      ok := NOT (i < n)
    END

  BEGIN (* sort *)
    i := 1;
    WHILE i < n DO
      write (1);
      j := i; k := i;
      WHILE j < n DO   (* a [k] = MIN a [i..j] *)
        j := j + 1;
        IF a [j] < a [k] THEN k := j ELSE skip END
      END;
      h := a [i]; a [i] := a [k]; a [k] := h;
      i := i + 1
    END;
    check (ok); WRITE (ok)
  END

BEGIN (* main program *)
  a [1] := 2.1415926536;
  a [1] := a [1] + 1.0;
  read (n, a);
  sort (a);
  IF NOT test THEN write (0) ELSE skip END
END.
```

3. ICode

The intermediate code (ICode) for MiniLAX is a subset of the intermediate code for Pascal (P-Code) [NAJ76]. ICode programs consist of simple instructions for a hypothetical computer — a stack machine.

3.1. The ICode Machine

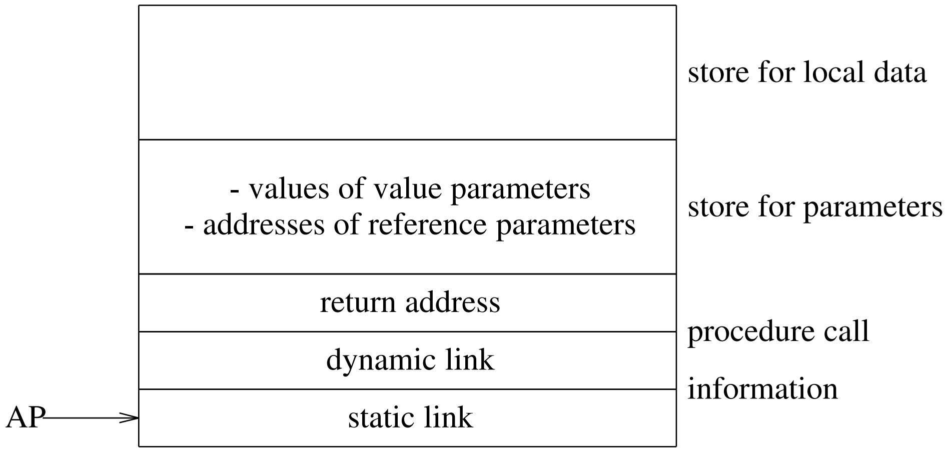
The ICode Machine consists of three registers and memory. The registers are

- PC the program counter
- SP the stack pointer
- AP the activation record pointer

The program counter points to the current instruction in the memory. The stack pointer points to the highest occupied stack cell. The activation record pointer points to the 'static link' field of the current activation record.

The memory is divided in two parts, one containing the program (Code) and the other containing data (Store). Code is an array of ICode instructions. Store is organized as stack (growing upwards) which contains the data of the program executed. Each activation of a procedure results in pushing an activation record on the stack, which contains storage for parameters and local data.

An activation record has the following layout:



At initialization time, the static and dynamic links and the return address of the main program are all set to 0. The registers are initialized as follows: PC := 0, SP := 3, and AP := 1. The start address is 0, i.e. Code [0] contains the first ICode instruction to be executed. PC is incremented before the according instruction is executed. The interpreter stops at return from the main program. The stop condition is: (PC = 0).

A procedure call enforces

- the creation of static and dynamic links of the new activation record (ICode instruction: MST)
- parameter passing: The values of value parameters and the addresses of reference parameters are evaluated and pushed on the stack.
- storing the return address and a jump to the procedure (ICode instruction: JSR)
- reservation of store for local data of the new activation record (ICode instruction: ENT)

A return from a procedure enforces

- discarding the current activation record by updating the registers

3.2. ICode Instructions

For each ICode instruction its operation code, its parameters and its meaning is given in the following. The meaning is given as text and as formula which describe operations on the runtime stack. To simplify the description, within formulas it is not taken care about the types of the stack elements.

If not further mentioned, the operations apply to the top of the stack, which contains the actual element. The following shorthand notations are used:

S	runtime stack
base(P)	returns a pointer to the P'th static predecessor of the current activation record

An instruction may have up to two parameters with the following meaning:

o	offset
c, c1, c2	constants

a	address (index of code section)
t	indicates type integer (1), real (2) or boolean (3)
l	block level difference between current and referenced activation record

Note: The types integer, real and boolean are encoded with 1, 2, and 3. The boolean values FALSE and TRUE are encoded by 0 and 1.

1. Load instructions:

LDA l o	SP:=SP+1; S[SP]:=base(l)+o;	load address with base and offset
LDC t c	SP:=SP+1; S[SP]:=c;	load constant c of type t
LDI	S[SP]:=S[S[SP]];	load indirect

2. Store instructions:

STI	S[S[SP-1]]:=S[SP]; SP:=SP-1;	store into address contained in the element below the top

3. Jump instructions:

JMP a	PC:=a;	unconditional jump
FJP a	if not S[SP] then PC:=a; SP:=SP-1;	conditional jump

4. Arithmetic instructions:

ADD t	SP:=SP-1; S[SP]:=S[SP]+S[SP+1];	addition of type t
SUB	SP:=SP-1; S[SP]:=S[SP]-S[SP+1];	integer subtraction
MUL t	SP:=SP-1; S[SP]:=S[SP]*S[SP+1];	multiplication of type t

5. Logic instructions:

INV	S[SP]:=not S[SP];	
LES t	SP:=SP-1; S[SP]:=S[SP]<S[SP+1];	less operation of type t

6. Address calculation instructions:

IXA c	SP:=SP-1; S[SP]:=c*S[SP+1]+S[SP];	compute indexed address

7. Convert instructions:

FLT	S[SP]:=real(S[SP]);	converts from integer to real

8. Input-output instructions:

WRI t	write(S[SP]); SP:=SP-1;
REA t	SP:=SP+1; read(S[SP]);

9. Subroutine handling instructions:

MST l		activation record initialization:
	S[SP+1]:=base(l);	- store static predecessor
	S[SP+2]:=AP;	- store dynamic predecessor
	SP:=SP+3;	return address (=S[SP+3]) is stored by JSR
JSR o a	AP:=SP-(o+2);	set AP to point to new activation record
		o = #locations for parameters
	S[AP+2]:=PC;	store return address
	PC:=a	set PC to first instruction of subroutine
ENT o	SP:=SP+o	storage reservation for new block
		o = length of local data segment
RET	SP:=AP-1;	return from subroutine:
	PC:=S[SP+3];	- fetch return address to restore PC
	AP:=S[SP+2];	- restore activation record pointer AP

10. check instructions:

CHK c1 c2	if (S[SP]<c1) or	check against upper and lower bounds
	(S[SP]>c2) then error	

4. Compiler Structure

Figure 1 gives an overview of the modules of the compiler.

4.1. Scanning

The scanner module as well as the source module are generated by the scanner generator *Rex*. The source module provides access to the source file. The scanner specification in the file *minilax.scan* contains only six rules describing comments, numbers, and identifiers. Except for comments, a rule consists of a regular expression, an attribute computation, and a RETURN statement. The rules for the other tokens including the keywords are extracted automatically from the context-free grammar of the language and inserted at the line `INSERT RULES` using the preprocessors *cg -xz* and *rpp* [Gro91c]. The source position of all tokens is passed automatically in the variable *Attribute* to later compiler phases to be used for error messages.

4.2. Conversion

The three conversion operations used by the scanner are taken from a library of reusable modules. The procedures *StringToInt* and *StringToReal* map strings to integer and real numbers. The procedure *MakeIdent* unambiguously maps identifiers (strings) to integer numbers using hashing and a string memory.

4.3. Parsing

The parser module as well as the error reporting module are generated using the LALR(1) parser generator *Lalr*. The parser specification in the file *minilax.pars* contains an LALR(1) grammar for MiniLAX. As the subgrammar for expressions is ambiguous operator precedences are given in the PREC section to solve this problem. The grammar is written in a style that reflects the semantic structure of the language: Almost every grammar rule corresponds to one node type in the abstract syntax tree.

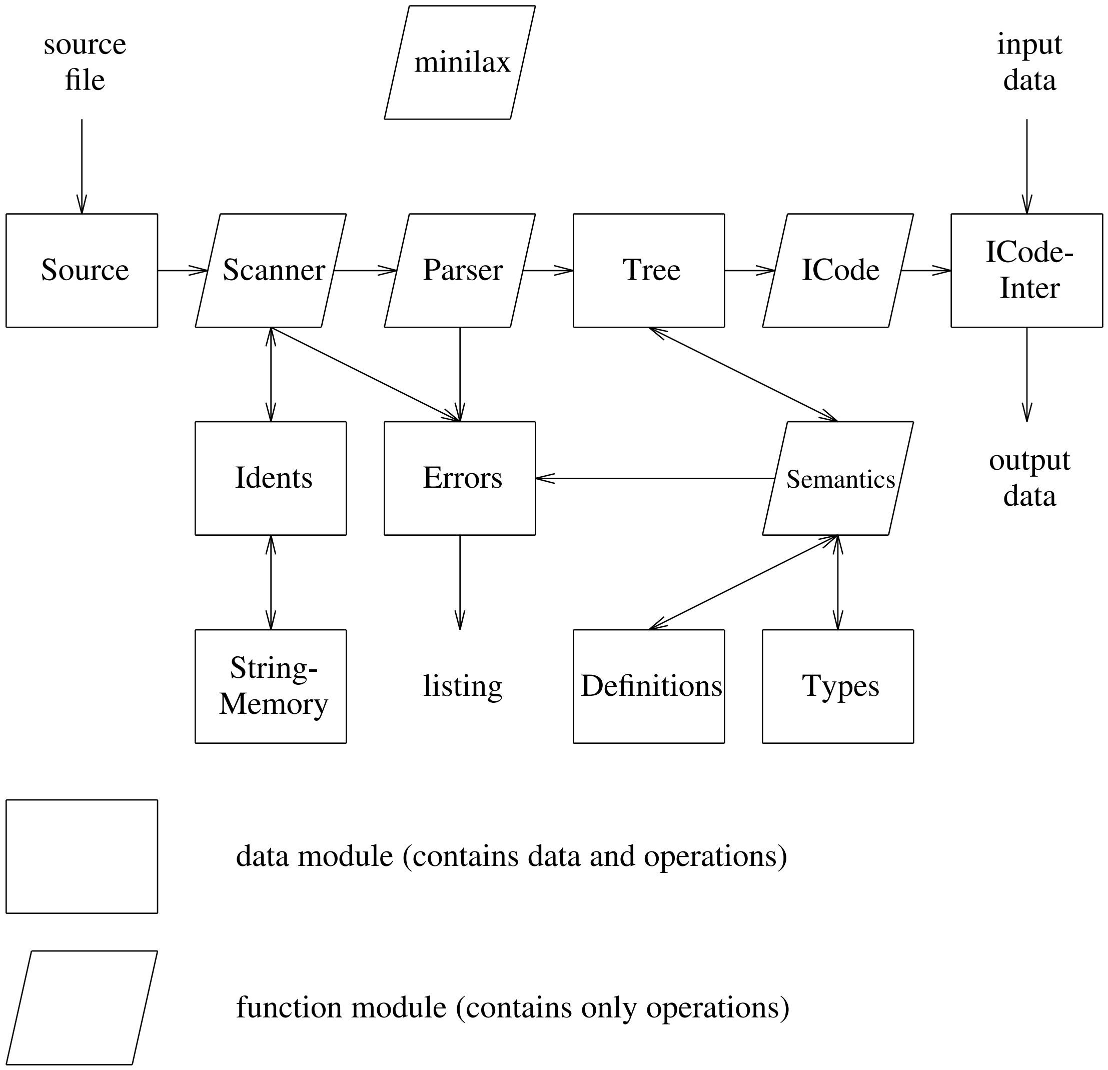


Fig. 1: Module structure – data flow (simplified)

4.4. Tree Construction

The tree module is generated by the generator for abstract syntax trees *Ast*. This module implements an abstract data type for trees. It primarily defines the structure of the trees and provides procedures to construct the tree nodes. The specification for the syntax tree is contained in the file *minilax.cg* (first page). It constitutes a context-free grammar augmented by several features like rule names, selector names, typed attributes, and inheritance which are described in detail in [Gro91a]. Each grammar rule corresponds to a node type in the abstract syntax tree. For every node type a constructor procedure is provided whose name consists of the rule name with the prefix 'm'.

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 node types called *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.

The mapping of the concrete syntax to the abstract syntax tree is specified in the parser specification by the actions associated with the grammar rules. The underlying principle is an attribute grammar with only one synthesized attribute called *Tree*. Except for (semantic) chain rules, the action of every grammar rule constructs one tree node by calling a generated constructor procedure.

For sequences of declarations, statements, etc. left recursive grammar rules are used in order to avoid overflow of the parse stack. (This is historic because the latest version of *Lalr* has a flexible stack which will never overflow.) This leads to syntax trees where the elements of sequences are in the wrong order. The calls of the procedure *ReverseTree* which is generated by *Ast* at certain places solves this problem.

4.5. Semantic Analysis

Semantic analysis which consists of name analysis, type checking, and computation of code generation attributes is generated using the attribute grammar generator *Ag* [Gro89]. It generates evaluators for ordered attribute grammars. The attribute computations have to be specified in a functional style. They are written in the desired target language. They can use external functions of separately compiled abstract data types. *Ag* cooperates with *Ast* in the sense that both tools understand the same specification language and the generated attribute evaluators operate on the trees generated by *Ast*. *Ag* offers many features which are improvements over traditional attribute grammar systems like attributes local to rules, modules, tree-valued attributes, and inheritance. Attributes of left-hand side symbols are denoted just by the attribute name. For attributes of right-hand side symbols the attribute name is preceded by the selector name of the symbol and a colon.

The semantic analysis is specified by the attribute grammar in the file *minilax.cg*. The attribute grammar is based on the abstract syntax. It is divided into modules where each module describes the computation of one attribute. The context conditions are also contained in a separate module. 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.

4.6. Name Analysis

Name analysis is controlled by the modules *Decls* and *Env*. The attributes *DeclsIn* and *DeclsOut* collect the declarations in a scope. The attribute *Env* describes the environment information. It is distributed over all declarations and statements where it is valid. Used identifiers are mapped to the denoted objects by the procedure *Identify*.

The data structure and procedures used in name analysis are provided by the external module *Definitions*. This module is generated by *Ast* out of the specification in the file *Definitions.cg*. The last seven lines describe the data structure and the corresponding constructor procedures. Some additional operations like e. g. *Identify* are provided as Modula-2 code.

4.7. Operator Identification

Operator identification for MiniLAX is trivial. It is handled in an ad hoc manner in the module *TypeCode* by computing the attribute *TypeCode* for certain node types.

4.8. Semantic Checks

The attribute grammar modules *Formals* and *Types* control the type determination for every subexpression. The module *Conditions* contains all context checks for MiniLAX. The above modules use the operations of the external module *Types* for manipulation of types. This module is conveniently generated by *Puma* [Gro91b] out of a specification contained in the file *Types.puma*. The reporting of error messages is completely expressed in the target language. The source position is treated like any other attribute. This allows to combine error messages with precise source positions.

4.9. Action Mapping

The mapping of the abstract syntax tree to the intermediate language is carried out in two steps. First, using the attribute grammar code generation attributes are computed. Second, a transformation module generated by *Puma* [Gro91b] traverses the attributed tree and emits the code.

The module *Co* computes for every subexpression an attribute describing the coercions to be applied. Coercions are operations which are not explicitly given in the source program. The procedure *Coerce* is imported from the external module *Types*. The module *DataSize* computes the offset of all variables. The module *CodeSize* computes the code address of every target code instruction. The module *Level* distributes the nesting level over the complete syntax tree. The module *Label* copies the values of certain code addresses into local attributes. The intention is to allow an optimizer to turn the attribute pair *CodeSizeIn/Out* into a global variable. The information gathered in the modules *DataSize, CodeSize*, and *Level* is added to the definition table (see module *Decls*) in a functional way.

The program module *ICode* is generated by *Puma* [Gro91b] out of the specification in the file *ICode.puma*. It recursively traverses the attributed tree and emits the intermediate code. For this purpose it uses the operations of the hand-written module *ICodeInter* which stores the intermediate code in an array and provides an interpreter and a printing routine for it. The reason for not using the attribute grammar for code generation is the following: The attribute grammar would compute the code as a list- or tree-valued attribute of the root. As attribute grammars have no notion of output an external function would have to transform this attribute value into the array structure required by the interpreter. We save storage and runtime to compute the code attribute by constructing the array directly. Specifying the mapping to intermediate code using a separate module is competitive to an attribute grammar version in terms of clarity and size.

The module *minilax* is the main program that glues it all together.

5. Specification

5.1. minilax.scan

```
EXPORT  {
FROM Idents     IMPORT tIdent;
FROM Positions  IMPORT tPosition;

INSERT tScanAttribute
}

GLOBAL  {
FROM SYSTEM     IMPORT ADR;
FROM Strings    IMPORT tString, StringToInt, StringToReal;
FROM Idents     IMPORT tIdent, NoIdent, MakeIdent;
FROM Errors     IMPORT Message, MessageI, Error, String;

INSERT ErrorAttribute
}

LOCAL   { VAR Word: tString; }

DEFAULT {
   GetWord (Word);
   MessageI ("illegal character", Error, Attribute.Position, String, ADR (Word));
}

EOF     {
   IF yyStartState = Comment THEN
      Message ("unclosed comment", Error, Attribute.Position);
   END;
}

DEFINE  digit   = {0-9} .
        letter  = {a-z A-Z} .

START   Comment

RULE
          "(*"  :- {yyStart (Comment);}
#Comment# "*)"  :- {yyStart (STD);}
#Comment# "*" | - {*\t\n} + :- {}

#STD# digit +   : {GetWord (Word);
                   Attribute.IntegerConst.Integer := StringToInt (Word);
                   RETURN IntegerConst;}

#STD# digit * "." digit + (E {+\-} ? digit +) ?
                : {GetWord (Word);
                   Attribute.RealConst.Real := StringToReal (Word);
                   RETURN RealConst;}

INSERT RULES #STD#

#STD# letter (letter | digit) *
                : {GetWord (Word);
                   Attribute.Ident.Ident := MakeIdent (Word);
                   RETURN Ident;}
```

5.2. minilax.pars

```
PARSER minilax

PREC    LEFT    '<'                                 /* operator precedence  */
        LEFT    '+'
        LEFT    '*'
        LEFT    NOT

PROPERTY INPUT

RULE                                                /* concrete syntax      */

Prog            = PROGRAM Ident ';' 'DECLARE' Decls 'BEGIN' Stats 'END' '.' .
```

```
Decls           = <
   Decls1       = Decl .
   Decls2       = Decls ';' Decl .
> .
Decl            = <
   Var          = Ident ':' Type .
   Proc0        = PROCEDURE Ident ';' 'DECLARE' Decls 'BEGIN' Stats 'END' .
   Proc         = PROCEDURE Ident '(' Formals ')' ';'
                                      'DECLARE' Decls 'BEGIN' Stats 'END' .
> .
Formals         = <
   Formals1     = Formal .
   Formals2     = Formals ';' Formal .
> .
Formal          = <
   Value        = Ident ':' Type .
   Ref          = VAR Ident ':' Type .
> .
Type            = <
   Int          = INTEGER .
   Real         = REAL .
   Bool         = BOOLEAN .
   Array        = ARRAY '[' Lwb: IntegerConst '..' Upb: IntegerConst ']' OF Type .
> .
Stats           = <
   Stats1       = Stat .
   Stats2       = Stats ';' Stat .
> .
Stat            = <
   Assign       = Adr ':=' Expr .
   Call0        = Ident .
   Call         = Ident '(' 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       = IntegerConst .
   RConst       = RealConst .
   False        = FALSE .
   True         = TRUE .
   Adr          = <
      Name      = Ident .
      Index     = Adr '[' Expr ']' .
   > .
> .

                                            /* terminals (with attributes)  */

Ident           : [Ident: tIdent] { Ident       := NoIdent      ; } .
IntegerConst    : [Integer      ] { Integer     := 0            ; } .
RealConst       : [Real : REAL  ] { Real        := 0.0          ; } .

MODULE Tree
                                /* external functions for tree construction     */
PARSER GLOBAL   {

FROM Tree       IMPORT
   tTree        , TreeRoot      , ReverseTree   , mMiniLax      , mNoDecl       ,
   mDecl        , mProc         , mVar          , mNoFormal     , mFormal       ,
   mType        , mInteger      , mReal         , mBoolean      , mArray        ,
   mRef         , mNoStat       , mStat         , mAssign       , mCall         ,
   mIf          , mWhile        , mRead         , mWrite        , mNoActual     ,
   mActual      , mExpr         , mBinary       , mUnary        , mIntConst     ,
   mRealConst   , mBoolConst    , mAdr          , mIndex        , mIdent        ,
   Plus         , Times         , Less          , Not           , NoTree        ;
```

```
VAR nInteger, nReal, nBoolean   : tTree;
}

BEGIN   {
   nInteger     := mInteger     ();
   nReal        := mReal        ();
   nBoolean     := mBoolean     ();
}
                                /* attributes for tree construction            */
DECLARE
  Decls Decl Formals Formal Type Stats Stat Actuals Expr = [Tree: tTree] .

RULE                            /* tree construction =                         */
                                /* mapping: concrete syntax -> abstract syntax  */

Prog    = { => {TreeRoot := mMiniLax (mProc (mNoDecl (), Ident:Ident,
                Ident:Position, mNoFormal (), ReverseTree (Decls:Tree),
                ReverseTree (Stats:Tree)));};                              } .
Decls1  = { Tree := {Decl:Tree^.\Decl.Next := mNoDecl (); Tree := Decl:Tree;};  } .
Decls2  = { Tree := {Decl:Tree^.\Decl.Next := Decls:Tree; Tree := Decl:Tree;};  } .
Var     = { Tree := mVar (NoTree, Ident:Ident, Ident:Position, mRef (Type:Tree));}.
Proc0   = { Tree := mProc (NoTree, Ident:Ident, Ident:Position, mNoFormal (),
                    ReverseTree (Decls:Tree), ReverseTree (Stats:Tree));       } .
Proc    = { Tree := mProc (NoTree, Ident:Ident, Ident:Position, ReverseTree
           (Formals:Tree), ReverseTree (Decls:Tree), ReverseTree (Stats:Tree)); } .
Formals1= { Tree := {Formal:Tree^.\Formal.Next := mNoFormal ();
                    Tree := Formal:Tree;};                                     } .
Formals2= { Tree := {Formal:Tree^.\Formal.Next := Formals:Tree;
                    Tree := Formal:Tree;};                                     } .
Value   = { Tree := mFormal (NoTree, Ident:Ident, Ident:Position,
                    mRef (Type:Tree));                                         } .
Ref     = { Tree := mFormal (NoTree, Ident:Ident, Ident:Position,
                    mRef (mRef (Type:Tree)));                                  } .
Int     = { Tree := nInteger;                                                  } .
Real    = { Tree := nReal;                                                     } .
Bool    = { Tree := nBoolean;                                                  } .
Array   = { Tree := mArray (Type:Tree, Lwb:Integer, Upb:Integer, Lwb:Position); } .
Stats1  = { Tree := {Stat:Tree^.\Stat.Next := mNoStat (); Tree := Stat:Tree;};  } .
Stats2  = { Tree := {Stat:Tree^.\Stat.Next := Stats:Tree; Tree := Stat:Tree;};  } .
Assign  = { Tree := mAssign (NoTree, Adr:Tree, Expr:Tree, ':=':Position);      } .
Call0   = { Tree := mCall (NoTree, mNoActual (Ident:Position), Ident:Ident,
                    Ident:Position);                                           } .
Call    = { Tree := mCall (NoTree, ReverseTree (Actuals:Tree), Ident:Ident,
                    Ident:Position);                                           } .
If      = { Tree := mIf (NoTree, Expr:Tree, ReverseTree (Then:Tree),
                    ReverseTree (Else:Tree));                                  } .
While   = { Tree := mWhile (NoTree, Expr:Tree, ReverseTree (Stats:Tree));      } .
Read    = { Tree := mRead (NoTree, Adr:Tree);                                  } .
Write   = { Tree := mWrite (NoTree, Expr:Tree);                                } .
Expr1   = { Tree := mActual (mNoActual (Expr:Tree^.\Expr.Pos), Expr:Tree);     } .
Expr2   = { Tree := mActual (Actuals:Tree, Expr:Tree);                         } .
Less    = { Tree := mBinary ('<':Position, Lop:Tree, Rop:Tree, Less);          } .
Plus    = { Tree := mBinary ('+':Position, Lop:Tree, Rop:Tree, Plus);          } .
Times   = { Tree := mBinary ('*':Position, Lop:Tree, Rop:Tree, Times);         } .
Not     = { Tree := mUnary (NOT:Position, Expr:Tree, Not);                     } .
IConst  = { Tree := mIntConst (IntegerConst:Position, IntegerConst:Integer);   } .
RConst  = { Tree := mRealConst (RealConst:Position, RealConst:Real);           } .
False   = { Tree := mBoolConst (FALSE:Position, \FALSE);                       } .
True    = { Tree := mBoolConst (TRUE:Position, \TRUE);                         } .
Name    = { Tree := mIdent (Ident:Position, Ident:Ident);                      } .
Index   = { Tree := mIndex ('[':Position, Adr:Tree, Expr:Tree);                } .

END Tree
```

5.3. minilax.cg

```
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, mNoCoercion;
}

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 := mNoCoercion ();
               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;

PROCEDURE Error (Text: ARRAY OF CHAR; Position: 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;

ROCEDURE 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
```

5.4. Definitions.cg

```
TREE Definitions

IMPORT  {
   FROM Idents  IMPORT tIdent;
   FROM SYSTEM  IMPORT ADDRESS;
}

EXPORT  {
CONST NoEnv     = NoDefinitions;

TYPE
   tObjects     = tDefinitions; (* type to represent sets of objects    *)
   tEnv         = tDefinitions; (* type to represent environments       *)

PROCEDURE Identify      (Ident: tIdent; Env: tEnv): tObjects;
                        (* return the object associated with 'Ident' in *)
                        (* the environment 'Env'                        *)

PROCEDURE IsDeclared    (Ident: tIdent; Objects: tObjects): BOOLEAN;
                        (* check whether an object having the name      *)
                        (* 'Ident' is element of the set of objects     *)
                        (* 'Objects'                                    *)

PROCEDURE mProc2        (Object: tObjects; Level, Label, ParSize, DataSize: INTEGER);
                        (* extend the description 'Object' of a         *)
                        (* procedure by the 4 given attributes          *)

PROCEDURE mVar2         (Object: tObjects; Level, Offset: INTEGER);
                        (* extend the description 'Object' of a         *)
                        (* variable by the 2 given attributes           *)

PROCEDURE IsObjectKind  (Object: tObjects; Kind: SHORTCARD): BOOLEAN;
                        (* returns TRUE if the kind of the 'Object'     *)
                        (* is equal to parameter 'Kind'                 *)

PROCEDURE GetFormals    (Object: tObjects): ADDRESS;
                        (* returns the list of formal parameters        *)
                        (* from the description 'Object' of a procedure *)

PROCEDURE GetType       (Object: tObjects): ADDRESS;
                        (* returns the type of the description 'Object' *)
                        (* of a variable                                *)
}

GLOBAL  {

FROM Idents     IMPORT tIdent;
FROM SYSTEM     IMPORT ADDRESS;
FROM Tree       IMPORT mNoType, mNoFormal;

VAR nNoObject   : tObjects;

PROCEDURE IsDeclared    (Ident: tIdent; Objects: tObjects): BOOLEAN;
   BEGIN
      WHILE Objects^.Kind # NoObject DO
         IF Objects^.Object.Ident = Ident THEN
            RETURN TRUE;
         END;
         Objects := Objects^.Object.Next;
```

```
      END;
      RETURN FALSE;
   END IsDeclared;

PROCEDURE Identify      (Ident: tIdent; Env: tEnv): tObjects;
   VAR Objects  : tObjects;
   BEGIN
      WHILE Env # NoEnv DO
         Objects := Env^.Env.Objects;
         WHILE Objects^.Kind # NoObject DO
            IF Objects^.Object.Ident = Ident THEN
               RETURN Objects;
            END;
            Objects := Objects^.Object.Next;
         END;
         Env := Env^.Env.Hidden;
      END;
      RETURN nNoObject;
   END Identify;

PROCEDURE mProc2        (Object: tObjects; Level, Label, ParSize, DataSize: INTEGER);
   BEGIN
      Object^.Proc.Level        := Level;
      Object^.Proc.Label        := Label;
      Object^.Proc.ParSize      := ParSize;
      Object^.Proc.DataSize     := DataSize;
   END mProc2;

PROCEDURE mVar2         (Object: tObjects; Level, Offset: INTEGER);
   BEGIN
      Object^.Var.Level         := Level;
      Object^.Var.Offset        := Offset;
   END mVar2;

PROCEDURE IsObjectKind  (Object: tObjects; Kind: SHORTCARD): BOOLEAN;
   BEGIN
      RETURN (Object^.Kind = Kind) OR (Object^.Kind = NoObject);
   END IsObjectKind;

PROCEDURE GetFormals    (Object: tObjects): ADDRESS;
   BEGIN
      IF Object^.Kind = Proc THEN
         RETURN Object^.Proc.Formals;
      ELSE
         RETURN mNoFormal ();
      END;
   END GetFormals;

PROCEDURE GetType       (Object: tObjects): ADDRESS;
   BEGIN
      IF Object^.Kind = Var THEN
         RETURN Object^.Var.Type;
      ELSE
         RETURN mNoType ();
      END;
   END GetType;
}

BEGIN   { nNoObject := mNoObject (); }

RULE

Env             = Objects Hidden: Env .
Objects         = <
   NoObject     = .
   Object       = Next: Objects [Ident: tIdent] <
      Proc      = [Formals: ADDRESS] -> [Level: SHORTINT] [Label] [ParSize] [DataSize] .
      Var       = [Type: ADDRESS] -> [Level: SHORTINT] [Offset] .
   > .
> .
```

5.5. Types.puma

```
TRAFO Types PUBLIC
```

```
Reduce                  /* return type without any ref levels          */

ReduceToRef             /* return type with ref level 1                */

Reduce1                 /* return type with 1 ref level removed        */

RefLevel                /* return number of ref levels of a type       */

IsSimpleType            /* check whether a type is simple              */

IsCompatible            /* check whether two types are compatible      */

IsAssignmentCompatible  /* check whether two types are                 */
                        /* assignment compatible                       */

ResultType              /* return the type of the result of            */
                        /* applying an operator to two operands        */

CheckParams             /* check a formal list of parameters           */
                        /* against an actual list of parameters        */

GetElementType          /* return the type of the elements of          */
                        /* an array type                               */

TypeSize                /* return the number of bytes used for         */
                        /* the internal representation of an           */
                        /* object of a certain type                    */

Coerce                  /* returns the coercion necessary to convert   */
                        /* an object of type 't1' to type 't2'         */

EXTERN Error

GLOBAL {

FROM Positions  IMPORT tPosition;
FROM Tree       IMPORT
   tTree        , Array         , Ref           , NoType        ,
   Plus         , Times         , Less          , Not           ,
   mBoolean     , mNoType       , mNoCoercion   , Error         ;

VAR nBoolean, nNoType, nNoCoercion      : tTree;
}

BEGIN {
   nBoolean     := mBoolean     ();
   nNoType      := mNoType      ();
   nNoCoercion  := mNoCoercion  ();
}

FUNCTION Reduce (Type) Type
   Ref (t)      RETURN Reduce (t) ?.
   t            RETURN t ?.

FUNCTION ReduceToRef (Type) Type
   Ref (t:Ref)  RETURN ReduceToRef (t) ?.
   t:Ref        RETURN t ?.
   t            RETURN t ?.

FUNCTION Reduce1 (Type) Type
   Ref (t)      RETURN t ?.
   t            RETURN t ?.

FUNCTION RefLevel (Type) INTEGER
   Ref (t)      RETURN RefLevel (t) + 1 ?.
   _            RETURN 0 ?.

PREDICATE IsSimpleType (Type)
   Array        ? FAIL; .
   _            ?.

PREDICATE IsCompatible (Type, Type)
   Integer      , Integer       ?.
   Real         , Real          ?.
   Boolean      , Boolean       ?.
   Array (t1, Lwb, Upb, _), Array (t2, Lwb, Upb, _) ;
```

```
   Ref (t1)     , t2            ;
   t1           , Ref (t2)      ? IsCompatible (t1, t2); .
   NoType       , _             ?.
   _            , NoType        ?.
   ErrorType    , _             ?.
   _            , ErrorType     ?.

PREDICATE IsAssignmentCompatible (Type, Type)
   Integer      , Integer       ?.
   Real         , Real          ?.
   Real         , Integer       ?.
   Boolean      , Boolean       ?.
   Ref (t1)     , t2            ;
   t1           , Ref (t2)      ? IsAssignmentCompatible (t1, t2); .
   NoType       , _             ?.
   _            , NoType        ?.
   ErrorType    , _             ?.
   _            , ErrorType     ?.

FUNCTION ResultType (Type, Type, INTEGER) Type  EXTERN Plus Times Less Not nBoolean;
   t:Integer    , Integer       , { Plus }      RETURN t        ?.
   t:Real       , Real          , { Plus }      RETURN t        ?.
   t:Integer    , Integer       , { Times }     RETURN t        ?.
   t:Real       , Real          , { Times }     RETURN t        ?.
   Integer      , Integer       , { Less }      RETURN nBoolean ?.
   Real         , Real          , { Less }      RETURN nBoolean ?.
   t:Boolean    , Boolean       , { Less }      RETURN t        ?.
   t:Boolean    , _             , { Not }       RETURN t        ?.
   Ref (t1)     , t2            , o             ;
   t1           , Ref (t2)      , o             RETURN ResultType (t1, t2, o) ?.
   t:NoType     , _             , _             RETURN t        ?.
   _            , t:NoType      , _             RETURN t        ?.
   ErrorType    , _             , _             RETURN NoType   ?.
   _            , ErrorType     , _             RETURN NoType   ?.
   ..                                           RETURN ErrorType?.

PROCEDURE CheckParams (Actuals, Formals)
   NoActual     , NoFormal      ?.
   NoActual (Pos), _            ?
      Error ("too few actual parameters"        , Pos); .
   Actual (_, Expr (Pos, ..)), NoFormal ?
      Error ("too many actual parameters"       , Pos); .

/* alternative 1 */

   Actual (NextA, Expr (Pos, TypeA, ..)), Formal (_, _, NextF, _, _, TypeF) ?
      {
         IF NOT IsCompatible (TypeA, TypeF) THEN
            Error ("parameter type incompatible", Pos);
         END;
         IF NOT (RefLevel (TypeF) - 1 <= RefLevel (TypeA)) THEN
            Error ("variable required"          , Pos);
         END;
      };
      CheckParams (NextA, NextF); .

/* alternative 2 */

   Actual (NextA, Expr (Pos, TypeA, ..)), Formal (_, _, NextF, _, _, TypeF) ?
      NOT IsCompatible (TypeA, TypeF);
      Error ("parameter type incompatible"      , Pos);
      REJECT; .

   Actual (NextA, Expr (Pos, TypeA, ..)), Formal (_, _, NextF, _, _, TypeF) ?
      NOT (RefLevel (TypeF) - 1 <= RefLevel (TypeA));
      Error ("variable required"                , Pos);
      REJECT; .

   Actual (NextA, Expr (Pos, TypeA, ..)), Formal (_, _, NextF, _, _, TypeF) ?
      CheckParams (NextA, NextF); .

/* alternative 3 */

   Actual (NextA, Expr (Pos, TypeA, ..)), Formal (_, _, NextF, _, _, TypeF) ?
      CheckCompatible (Pos, TypeA, TypeF);
      CheckRefLevel (Pos, TypeA, TypeF);
```

```
      CheckParams (NextA, NextF); .

PROCEDURE CheckCompatible (tPosition, Type, Type)
   _    , t1    , t2    ? IsCompatible (t1, t2); .
   Pos  , ..            ? Error ("parameter type incompatible"  , Pos); .

PROCEDURE CheckRefLevel (tPosition, Type, Type)
   _    , t1    , t2    ? RefLevel (t2) - 1 <= RefLevel (t1); .
   Pos  , ..            ? Error ("variable required"            , Pos); .

FUNCTION GetElementType (Type) Type
   Array (t, ..)        RETURN t ?.
   _                    RETURN NoType ?.

FUNCTION TypeSize (Type) INTEGER
   Array (t, Lwb, Upb, _)       RETURN (Upb - Lwb + 1) * TypeSize (t) ?.
   _                            RETURN 1 ?.

FUNCTION Coerce (t1: Type, t2: Type) Coercions  EXTERN nNoCoercion;
   Ref (T1)     , Ref (T2)      RETURN Coerce (T1, T2) ?.
   Integer      , Real          RETURN IntToReal (nNoCoercion) ?.
   Ref (T1)     , T2            RETURN Content (Coerce (T1, T2)) ?.
   ..                           RETURN nNoCoercion ?.
```

5.6. ICode.puma

```
TRAFO ICode TREE Tree Definitions PUBLIC Code

EXTERN
   ADD BoolType CHK ENT Emit EmitReal FJP FLT FalseCode INV IXA IntType JMP JSR
   LDA LDC LDI LES MST MUL REA RET RealType STI SUB TrueCode TypeSize WRI

GLOBAL {
FROM Tree       IMPORT tTree, Times, Plus, Less;
FROM Definitions IMPORT tObjects;
FROM Types      IMPORT TypeSize;

FROM ICodeInter IMPORT
   IntType      , RealType      , BoolType      , OpCode        ,
   Emit         , EmitReal      , TrueCode      , FalseCode     ;
}

PROCEDURE Code (t: Tree)

MiniLax (Proc) ?
        Code (Proc);
        .
Proc (Next := Next:Decls (Definitions.Proc (ParSize := ParSize,
                DataSize := DataSize), ..), Decls := Decls, Stats := Stats) ?
        Emit (ENT, DataSize - ParSize, 0);
        Code (Stats);
        Emit (RET, 0, 0);
        Code (Decls);
        Code (Next);
        .
Var (Next := Next) ?
        Code (Next);
        .
Assign (Next, Adr, Expr, _) ?
        Code (Adr); Code (Adr::Co);
        Code (Expr); Code (Expr::Co);
        Emit (STI, 0, 0);
        Code (Next);
        .
Call (Next, Actuals, _, _, Definitions.Proc (Level := Level, Label := Label,
                ParSize := ParSize), level) ?
        Emit (MST, level - Level, 0);
        Code (Actuals);
        Emit (JSR, ParSize - 3, Label);
        Code (Next);
        .
If (Next, Expr, Then, Else, Label1, Label2) ?
        Code (Expr); Code (Expr::Co);
        Emit (FJP, Label1, 0);
        Code (Then);
```

```
        Emit (JMP, Label2, 0);
        Code (Else);
        Code (Next);
        .
While (Next, Expr, Stats, Label1, Label2) ?
        Emit (JMP, Label2, 0);
        Code (Stats);
        Code (Expr); Code (Expr::Co);
        Emit (INV, 0, 0);
        Emit (FJP, Label1, 0);
        Code (Next);
        .
Read (Next, Adr, TypeCode) ?
        Code (Adr); Code (Adr::Co);
        Emit (REA, TypeCode, 0);
        Emit (STI, 0, 0);
        Code (Next);
        .
Write (Next, Expr, TypeCode) ?
        Code (Expr); Code (Expr::Co);
        Emit (WRI, TypeCode, 0);
        Code (Next);
        .
Actual (Next, Expr) ?
        Code (Expr); Code (Expr::Co);
        Code (Next);
        .
Binary (_, _, _, Lop, Rop, {Times}, TypeCode) ?
        Code (Lop); Code (Lop::Co);
        Code (Rop); Code (Rop::Co);
        Emit (MUL, TypeCode, 0);
        .
Binary (_, _, _, Lop, Rop, {Plus}, TypeCode) ?
        Code (Lop); Code (Lop::Co);
        Code (Rop); Code (Rop::Co);
        Emit (ADD, TypeCode, 0);
        .
Binary (_, _, _, Lop, Rop, {Less}, TypeCode) ?
        Code (Lop); Code (Lop::Co);
        Code (Rop); Code (Rop::Co);
        Emit (LES, TypeCode, 0);
        .
Unary (Expr := Expr) ?
        Code (Expr); Code (Expr::Co);
        Emit (INV, 0, 0);
        .
IntConst (Value := Value) ?
        Emit (LDC, IntType, Value);
        .
RealConst (Value := Value) ?
        EmitReal (LDC, RealType, Value);
        .
BoolConst (Value := {TRUE}) ?
        Emit (LDC, BoolType, TrueCode);
        .
BoolConst (Value := {FALSE}) ?
        Emit (LDC, BoolType, FalseCode);
        .
Index (_, _, _, Adr, Expr, Array (Type, Lwb, Upb, _)) ?
        Code (Adr); Code (Adr::Co);
        Code (Expr); Code (Expr::Co);
        Emit (CHK, Lwb, Upb);
        Emit (LDC, IntType, Lwb);
        Emit (SUB, IntType, 0);
        Emit (IXA, TypeSize (Type), 0);
        .
Ident (_, _, _, Ident, Definitions.Var (Level := Level, Offset := Offset), level) ?
        Emit (LDA, level - Level, Offset);
        .
Content (Next) ?
        Emit (LDI, 0, 0);
        Code (Next);
        .
IntToReal (Next) ?
        Emit (FLT, 0, 0);
        Code (Next);
```

.

5.7. ICodeInter.md

```
DEFINITION MODULE ICodeInter;

CONST (* coding of OpCode parameters *)
      IntType   = 1;
      RealType  = 2;
      BoolType  = 3;
      FalseCode = 0;
      TrueCode  = 1;

          (* ICode instructions *)
TYPE OpCode =  (LDA, LDC, LDI, STI, JMP, FJP, ADD, SUB, MUL, INV,
                LES, IXA, FLT, WRI, REA, MST, JSR, ENT, RET, CHK);

PROCEDURE Emit (oc: OpCode; Param1, Param2: CARDINAL);
PROCEDURE EmitReal (oc: OpCode; Param1: CARDINAL; Param2: REAL);
  (* repeated calls of 'Emit' and 'EmitReal' write  *)
  (* the program into 'Code', starting at Code [0]. *)

PROCEDURE WriteCode; (* 'Code' is written on StdOut *)
PROCEDURE Interpret; (* executes ICode program *)

END ICodeInter.
```

5.8. ICodeInter.mi

```
IMPLEMENTATION MODULE ICodeInter;

FROM StdIO IMPORT ReadI, ReadR, WriteCard, WriteI, WriteR, WriteS, WriteNl;

CONST MaxCode   = 30000;
      MaxStore  = 10000;

      SL        = 1; (* static link    *) (* activation record organization *)
      DL        = 2; (* dynamic link   *)
      RA        = 3; (* return address *)

TYPE Ptype  = CARDINAL;    (* type of first parameter *)
     Qtype  = RECORD
                CASE : INTEGER OF
                | 1: qc: CARDINAL
                | 2: qr: REAL
                END
              END;

     CodeRange  = [0..MaxCode];         (* type of second parameter *)
     StoreRange = [0..MaxStore];
     StoreType  = (Undef, Int, Real, Bool, Adr, Mark);

VAR Code : ARRAY CodeRange OF RECORD    (* the program *)
                                 OP : OpCode;
                                 P  : Ptype;
                                 Q  : Qtype;
                              END;

    Store : ARRAY StoreRange OF RECORD  (* the data *)
                    CASE Stype: StoreType OF
                      Int  : Vi : INTEGER
                    | Real : Vr : REAL
                    | Bool : Vb : BOOLEAN
                    | Adr  : Va : StoreRange
                    | Mark : Vm : CodeRange
                    END;
                  END;

    PC,         (* program address register  *)
    LastPC      (* highest used code address *)
          : CodeRange;

        (* address registers *)
    AP,   (* points to the beginning of an activation record *)
```

```
    SP    (* points to top of the stack *)
        : StoreRange;

    OpCodeText  : ARRAY OpCode OF ARRAY [0..2] OF CHAR;

PROCEDURE Emit (oc: OpCode; Param1, Param2: CARDINAL);
BEGIN
  WITH Code [PC] DO
    OP   := oc;
    P    := Param1;
    Q.qc := Param2
  END;
  LastPC := PC;
  INC (PC);
END Emit;

PROCEDURE EmitReal (oc: OpCode; Param1: CARDINAL; Param2: REAL);
BEGIN
  WITH Code [PC] DO
    OP   := oc;
    P    := Param1;
    Q.qr := Param2
  END;
  LastPC := PC;
  INC (PC);
END EmitReal;

PROCEDURE WriteInstr (Code: OpCode; Param1: Ptype; Param2: Qtype);
BEGIN
  WriteS (OpCodeText [Code]);
  CASE Code OF
    LDC:                        (* two parameters *)
      WriteI (Param1, 5);
      CASE Param1 OF
        IntType,
        BoolType: WriteI (Param2.qc, 5);
      | RealType: WriteR (Param2.qr, 5, 5, 1);
      END

  | LDA, JSR, CHK:              (* two parameters *)
      WriteI (Param1, 5);
      WriteI (Param2.qc, 5);

  | MST, ENT, IXA, LES, JMP,
    FJP, ADD, MUL, REA, WRI:    (* one parameter  *)
      WriteI (Param1, 5);

  | LDI, STI, RET, FLT, INV,    (* no parameter   *)
    SUB:
  END;
  WriteNl;
END WriteInstr;

PROCEDURE WriteCode;
  VAR pc : CARDINAL;
BEGIN
  WriteNl;
  WriteS ("Code: (Codelength ="); WriteCard (LastPC+1,4);
  WriteS (")");
  WriteNl;
  IF LastPC # 0 THEN
    FOR pc := 0 TO LastPC DO
      WriteI (pc, 5);
      WriteS (":   ");
      WITH Code [pc] DO
        WriteInstr (OP, P, Q)
      END
    END
  END;
  WriteNl;
END WriteCode;

PROCEDURE WriteStore;
  VAR Sptr : StoreRange;
BEGIN
  WriteNl;
```

```
  WriteS ("Store: (Index, Elementtype, Contents)"); WriteNl;

  FOR Sptr := SP TO 0 BY -1 DO
    IF AP = Sptr THEN
      WriteS ("   AP ->"); WriteI (Sptr, 4);
    ELSE
      WriteI (Sptr, 12);
    END;
    WITH Store [Sptr] DO
      CASE Stype OF
      | Int  :  WriteS ("        Int ");
                WriteI (Vi, 8);
      | Real :  WriteS ("        Real");
                WriteR (Vr, 8, 8, 1);
      | Bool :  WriteS ("        Bool");
                IF Vb THEN
                  WriteS ("    TRUE")
                ELSE
                  WriteS ("   FALSE")
                END
      | Adr  :  WriteS ("        Adr ");
                WriteI (Va, 8);
      | Mark :  WriteS ("        Mark");
                WriteI (Vm, 8);
      ELSE WriteS ("    Stype not defined");
      END;
      WriteNl;
    END;
  END;
  WriteNl
END WriteStore;

PROCEDURE Interpret;

VAR OP  : OpCode; (* instruction register *)
    P   : Ptype;
    Q   : Qtype;
    sr1, sr2 : StoreRange;

  PROCEDURE CheckStore (p:CARDINAL);
  BEGIN
    IF p > MaxStore THEN WriteS ("Store overflow"); END;
  END CheckStore;

  PROCEDURE Base (Ld : CARDINAL): StoreRange;
      (* walks Ld times back the static link chain *)
    VAR Sr : StoreRange;
  BEGIN
    Sr := AP;
    WHILE Ld>0 DO
      Sr := Store [Sr].Vm;
      DEC (Ld);
    END;
    RETURN Sr;
  END Base;

BEGIN
  PC := 0;
  REPEAT
    OP := Code [PC].OP;                 (* fetch instruction *)
    P  := Code [PC].P;
    Q  := Code [PC].Q;

    INC(PC);

    CASE OP OF                          (* execute instruction *)

       (* load instructions *)
    | LDA  : INC (SP);
             Store [SP].Va := Base(P) + Q.qc;
             Store [SP].Stype := Adr;

    | LDC  : INC (SP);
             CASE P OF
               IntType  : Store [SP].Vi := Q.qc;
                          Store [SP].Stype := Int;
```

```
        | RealType : Store [SP].Vr := Q.qr;
                     Store [SP].Stype := Real;
        | BoolType : Store [SP].Vb := Q.qc = TrueCode;
                     Store [SP].Stype := Bool;
        END

| LDI  : Store [SP] := Store [Store [SP].Va];

  (* store instructions *)
| STI  : Store [Store [SP-1].Va] := Store [SP];
       SP := SP-2;

  (* jump instructions *)
| JMP  : PC := P

| FJP  : IF NOT Store [SP].Vb THEN
         PC := P;
       END;
       DEC (SP);

  (* arithmetic instructions *)
| ADD  : DEC (SP);
       CASE P OF
         IntType  : Store[SP].Vi := Store[SP].Vi + Store[SP+1].Vi;
       | RealType : Store[SP].Vr := Store[SP].Vr + Store[SP+1].Vr;
       END

| SUB  : DEC (SP);
       Store[SP].Vi := Store[SP].Vi - Store[SP+1].Vi;

| MUL  : DEC (SP);
       CASE P OF
         IntType  : Store[SP].Vi := Store[SP].Vi * Store[SP+1].Vi;
       | RealType : Store[SP].Vr := Store[SP].Vr * Store[SP+1].Vr;
       END

  (* logic  instructions *)
| INV  : Store[SP].Vb := NOT Store[SP].Vb;

| LES  : DEC (SP);
       CASE P OF
         IntType  : Store[SP].Vb := Store[SP].Vi < Store[SP+1].Vi;
       | RealType : Store[SP].Vb := Store[SP].Vr < Store[SP+1].Vr;
       | BoolType : Store[SP].Vb := Store[SP].Vb < Store[SP+1].Vb;
       END;
       Store [SP].Stype := Bool

  (* address calculating instructions *)
| IXA  : DEC (SP);  (* P=number of storage units *)
       Store [SP].Va := P*Store [SP+1].Va + Store [SP].Va;

  (* convert instructions *)
| FLT  : Store[SP].Vr := FLOAT(CARDINAL(Store[SP].Vi));
       Store[SP].Stype := Real;

  (* input-output instructions *)
| WRI  : CASE P OF
         IntType  : WriteI (Store[SP].Vi,5);     WriteNl;
       | RealType : WriteR (Store[SP].Vr,5,5,1); WriteNl;
       | BoolType : IF Store [SP].Vb THEN
                      WriteS ("    1"); WriteNl;
                    ELSE
                      WriteS ("    0"); WriteNl;
                    END
       END;
       DEC (SP)

| REA  : INC (SP);
       CASE P OF
         IntType  : Store[SP].Vi := ReadI();
                    Store[SP].Stype := Int
       | RealType : Store[SP].Vr := ReadR();
                    Store[SP].Stype := Real
       | BoolType : Store[SP].Vb := ReadI() = 1;
                    Store[SP].Stype := Bool;
       END
```

```
      (* subroutine handling instructions *)
    | MST  :    (* P=(level difference of calling and called procedure)+1 *)
           Store [SP+SL].Stype := Adr;
           Store [SP+SL].Va    := Base (P);
           Store [SP+DL].Stype := Adr;
           Store [SP+DL].Va    := AP;
           Store [SP+RA].Stype := Mark;
              (* return address is patched in JSR (after  *)
           SP := SP+3;              (* parameter passing) *)

    | JSR  : AP := SP-(P+2); (* P=number of locations for parameters *)
           Store [AP+2].Vm := PC;
           PC := Q.qc;      (* Q=entry point *)

    | ENT  : sr2 := SP+P;  (* P=length of local data segment *)
           FOR sr1 := SP+1 TO sr2 DO
             Store [sr1].Stype := Undef;
           END;
           CheckStore(sr2);
           SP := sr2;

    | RET  : SP := AP-1;
           PC := Store [SP+RA].Vm;
           AP := Store [SP+DL].Va;

      (* check instructions *)
    | CHK  : IF (Store [SP].Vi < INTEGER(P)) OR
              (Store [SP].Vi > INTEGER(Q.qc)) THEN
              WriteS ("range check error");
              WriteNl;
           END

    ELSE WriteS ("wrong OpCode :"); WriteS (OpCodeText [OP]); WriteNl;
    END;
  UNTIL PC = 0;
END Interpret;

BEGIN
  OpCodeText [LDA] := "LDA";
  OpCodeText [LDC] := "LDC";
  OpCodeText [LDI] := "LDI";
  OpCodeText [STI] := "STI";
  OpCodeText [JMP] := "JMP";
  OpCodeText [FJP] := "FJP";
  OpCodeText [ADD] := "ADD";
  OpCodeText [SUB] := "SUB";
  OpCodeText [MUL] := "MUL";
  OpCodeText [INV] := "INV";
  OpCodeText [LES] := "LES";
  OpCodeText [IXA] := "IXA";
  OpCodeText [FLT] := "FLT";
  OpCodeText [WRI] := "WRI";
  OpCodeText [REA] := "REA";
  OpCodeText [MST] := "MST";
  OpCodeText [JSR] := "JSR";
  OpCodeText [ENT] := "ENT";
  OpCodeText [RET] := "RET";
  OpCodeText [CHK] := "CHK";

  Store [0].Stype  := Undef;
  Store [SL].Stype := Adr;  Store [SL].Va := 0;
  Store [DL].Stype := Adr;  Store [DL].Va := 0;
  Store [RA].Stype := Mark; Store [RA].Vm := 0;

  PC := 0;
  SP := 3;
  AP := 1;
END ICodeInter.
```

5.9. minilax.mi

```
MODULE minilax;

FROM IO         IMPORT CloseIO;
FROM Parser     IMPORT Parse;
```

```
FROM Tree       IMPORT TreeRoot;
FROM Semantics  IMPORT BeginSemantics, Eval;
FROM ICode      IMPORT Code;
FROM ICodeInter IMPORT Interpret;

VAR ErrorCount : CARDINAL;

BEGIN
   ErrorCount := Parse ();
   BeginSemantics;
   Eval (TreeRoot);
   IF ErrorCount = 0 THEN
      Code (TreeRoot);
      Interpret;
   END;
   CloseIO;
END minilax.
```

References

[Gro87a] J. Grosch, Reusable Software - A Collection of Modula-Modules, Compiler Generation Report No. 4, GMD Forschungsstelle an der Universität Karlsruhe, Sep. 1987.

[Gro87b] J. Grosch, Rex - A Scanner Generator, Compiler Generation Report No. 5, GMD Forschungsstelle an der Universität Karlsruhe, Dec. 1987.

[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.

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

[GrE90] J. Grosch and H. Emmelmann, A Tool Box for Compiler Construction, Compiler Generation Report No. 20, GMD Forschungsstelle an der Universität Karlsruhe, Jan. 1990.

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

[Gro91c] J. Grosch, Preprocessors, Compiler Generation Report No. 24, GMD Forschungsstelle an der Universität Karlsruhe, Feb. 1991.

[NAJ76] K. V. Nori, U. Ammann, K. Jensen, H. H. Nägeli and C. Jacobi, The Pascal-P Compiler: Implementation Notes, Bericht 10, Eidgenössische Technische Hochschule, Zürich, July 1976.

[WaG84] W. M. Waite and G. Goos, *Compiler Construction*, Springer Verlag, New York, NY, 1984.

Contents