$\mathcal{LL}$ – An Object Oriented Library Language Reference Manual

(Preliminary Version)

Tomasz Kowaltowski and Evandro Bacarin

Relatório Técnico DCC–16/93

Julho de 1993

$\mathcal{LL}$ – An Object Oriented Library Language Reference Manual *(Preliminary Version)*

Tomasz Kowaltowski and Evandro Bacarin*

Abstract

This is the preliminary version of the reference manual for the language $\mathcal{LL}$ designed for easy interfacing with complex libraries and testing of algorithms written in a natural way, avoiding complexities of typical system programming languages. Generality is achieved through the fact that the language is completely object-oriented. It provides traditional syntax and extended semantics for its declarations, statements and expressions. Some of the most common data types and data structuring mechanisms are provided within the language, but it is expected that most of them will be defined through application oriented interfaces.

*The language described in this report is part of a Master thesis project of the second author under the supervision of the first author. This research was supported by grants awarded by the Brazilian National Council for Scientific and Technological Development (CNPq) to both authors. **E-mail:** {tomasz,bacarin}@dcc.unicamp.br.

The language designer should be familiar with many alternative features designed by others, and should have excellent judgment in choosing the best and rejecting any that are mutually inconsistent... One thing he should not do is to include untried ideas of his own. His task is consolidation, not innovation.
— C. A. R. Hoare

The principal role of a language designer is that of a judicious collector of features or concepts. Once these concepts are selected, forms of expressing them must be found, i. e. a syntax must be definded. The forms expressing individual concepts must be carefully molded into a whole. This is most important, as otherwise the language will appear as incoherent, as a skeleton onto which individual constructs were grafted, perhaps as after-thoughts.
— N. Wirth

Contents

1 Introduction

Sofisticated libraries of data structures and algorithms are becoming very popular and can increase significantly the productivity of the development of new applications. However, most of these libraries require programming of applications in their original language and the programmer must understand (or at least be aware of) many details of the language which have very little relation with the intended application. One of the most frequently used languages for such purpose is C++. The conventions of this language are quite complex and not always easy to master.

On the other hand, algorithms are shown frequently in books and scientific papers in a quite informal language, but their meaning is usually much clearer than their final versions programmed in a language like C++. The goal of the language described in this document is to provide a new environment which can help to bridge this gap between what is required to describe an algorithm and to implement it in detail when a rich library is available.

Some of the important goals in the design of the language, which is called $\mathcal{LL}$ for $\mathcal{L}$ibrary $\mathcal{L}$anguage, were:

- a small number of simple programming concepts;
- possibility of interfacing with any existing or future libraries;
- simple and readable syntax, close to the traditional mathematical notation.

In its purest form, $\mathcal{LL}$ should rely completely on outside libraries (written in some other implementation language like C++) for its universe of values which can be manipulated. However, in order to allow for traditional denotations of common kinds of objects (like integers, reals, strings) and of common data structuring constructs (like arrays and tuples), $\mathcal{LL}$ has some predeclared types which are known to the system.

$\mathcal{LL}$'s generality and extensibility is achieved through the fact that it is completely object-oriented in the sense that *every* value manipulated by an $\mathcal{LL}$ program, even of a predeclared type, is an object which possesses its own set of methods. Inclusion of application libraries, usually written in another language, is realized through the inclusion of convenient interfaces which specify the outside data types and their associated methods.

Sections 2 through 9 contain a description of the "pure" part of $\mathcal{LL}$. The syntax of the language is treated informally in these sections. The semantic explanation presumes that the reader has good knowledge of some other programming languages and is familiar with the object-oriented programming paradigm.

Section 9 describes some syntactic extensions. The appendices describe precisely the syntax of the language and the predeclared types, besides introducing a simple example. A local implementation guide should provide information about compiler options and implementation details necessary for writing interfaces to libraries implemented in other languages.

2 Lexical conventions

$\mathcal{LL}$ programs are written as sequences of tokens classified as (reserved) keywords, identifiers, operator symbols, constant (integer, real and string) denotations, implementation strings, and special symbols. For improved readability, some printing conventions are used within this document. Keywords are represented in **boldface**, identifiers, digits and string characters in *emphasized* type. Some symbols which would usually be represented by two characters are printed as one character (for instance, '$\leq$' instead of '<=' and '$\neq$' instead of '<>'). When implemented, all these tokens are expected to be represented in their usual ways. Spaces or other equivalent symbols (tabs, newlines) should separate these tokens whenever ambiguity might arise. Identifiers are case sensitive, whereas keywords are not. Thus '`function`', '`FUNCTION`, '`Function`' and '`fuNcTion`' represent all the same keyword **function**, but *box*, *Box* and *BOX* are three distinct identifiers (see Appendices B and C for more details about the syntax and program representation).

Comments in $\mathcal{LL}$ are opened by '(∗' and closed by '∗)'. They can extend over an arbitrary number of lines and can be nested. They can be also used to provide *pragmas*, i. e. some special information to the compiler (see the local implementation guide). Line comments in $\mathcal{LL}$ are opened by '||' and closed by an end-of-line character.

Keywords

all	**endmodule**	**mod**
and	**endprocedure**	**module**
begin	**endtype**	**not**
case	**endwhile**	**of**
const	**exit**	**operators**
constructors	**export**	**or**
div	**forall**	**procedure**
do	**from**	**procedures**
else	**function**	**repeat**
elsif	**functions**	**return**
endcase	**if**	**subtypeof**
endfor	**import**	**then**
endforall	**in**	**type**
endfunction	**is**	**until**
endif	**linking**	**var**
endloop	**loop**	**while**

Operator symbols

$=$	$\neq$	$<$	$\leq$	$>$
$\geq$	$-$	$+$	$*$	$/$
$\uparrow$	..	#	!	?
&	@	%	$\sim$	

Special symbols

$\ll$	$\gg$	{	}	*[*
]	(*	*)	*$*	<:
:	.	|	\\	$\leftarrow$
,	;	(	)	$\Rightarrow$
||				

3 Types

$\mathcal{LL}$ has a uniform type system in which *every* value is an object with its own *type stamp* and a set of constructors and methods. In general, there is no fixed type associated with variables, formal parameters etc, except when explicit type checking is desired.[1] It is expected that most types will be specified by convenient interfaces and implemented in their own

[1]Type restrictions can also be used to guide code optimization.

libraries, but they can also be introduced by any program. Types are described by type declarations which have the following general form:

```
type T subtypeof S is
   constructors
     | spec1
     | spec2
       ...
   procedures
     | spec1
     | spec2
       ...
   functions
     | spec1
     | spec2
       ...
   operators
     | spec1
     | spec2
       ...
endtype;
```

where the part '**subtypeof** S', and each section (**constructors**, **procedures**, **functions** and **operators**) are optional. Each $spec_i$ specifies the implementation of the corresponding constructor or method; methods can be of procedure, function or operator kind. Implementation specifications for constructors, and procedure and function methods are of the form:

ident(arg1; arg2; ...) type-restriction $\Rightarrow$ *impl*

(*type-restriction* must be empty in the case of constructors and procedure methods). Specifications for prefix, infix and postfix operators are of the form:

(op arg1) type-restriction	$\Rightarrow$	*impl*
(arg1 op arg2) type-restriction	$\Rightarrow$	*impl*
(arg1 op) type-restriction	$\Rightarrow$	*impl*

In each specification, the righthand side denotes the implementation of a constructor or method. Method specifications must have at least one argument; constructors may have zero or more arguments. The lefthand side and the righthand side should be thought of as the heading and the body of a routine declaration, defining a new scope (see Section 8) to which the formal parameters and local variables belong. All other identifiers visible in the module in which the type declaration occurs are also accessible. Notice that in a simple case of a function or operator method, the righthand side specification could be of the form '**return** *expr*' where *expr* represents the value to be returned. Methods and constructors of a type are always implemented as routines so that they can be called by their fully qualified names.

Within *linking modules* (see Section 8) the righthand side can be contained within single quotes (*implementation string*) meaning that it is written in the underlying library implementation language; in this case it is not interpreted by the $\mathcal{LL}$ compiler.[2]

Arguments are specified as distinct identifiers followed optionally by type restrictions (see further ahead). Notice that the first parameter of any method will always be of the type being defined (or its subtype – see below) unless a fully qualified method name was used to call it. *Type-restriction* in a specification is optional; when present it restricts the returned value as to its type the same way as for functions (see Section 7). When no *type-restriction* is present, the parentheses in operator specifications may be omitted.

Regardless of the implementation, the value returning expressions must produce objects as represented in $\mathcal{LL}$, i. e. stamp-value pairs. In

[2] Notice that actual parameters of constructors and methods will be *always* general $\mathcal{LL}$ objects, even if the righthand side is an implementation string.

case of constructor functions, the resulting value must be of a type which is a subtype (see further ahead) of S; the type stamp of this returned object will be replaced by the type stamp of the type being defined.

Names of constructors, function methods, procedure methods and operators can be declared more than once, i. e. *overloaded*, within each section as long as they correspond to specifications with different arities (in the case of operators, also if they are prefix, infix or postfix). They can also be repeated in different sections and in different type declarations, but constructor and function method names with the same arities cannot be overloaded within the same type declaration (see Section 8 for naming rules).

A type declaration of the form shown above defines a new type T, *direct subtype* of S. If S is not specified then it is assumed to be the predeclared type *root*. The type T inherits all the constructors and methods of the type S besides possessing the new constructors and methods introduced by the declaration. In case of a constructor or method name coincidence between that of T and of S, within the same section and with the same arity, only the specification given for T is accessible.[3]

The relation *subtype* on types is defined as the reflexive transitive closure of the relation *direct subtype* as defined above. A *type-restriction* as used in several constructs is written always as '<: T' meaning subtype, or as ': T' meaning the same types. Whenever such a restriction is specified, it must be checked by the run-time system (or by the compiler if possible).

As another side effect of the declaration, the identifier T becomes the name of a new constant of the predeclared type *typetag* known within the module in which the type declaration is contained (see module declarations in Section 8). The type *typetag* has the usual relational operator methods $<$, $\leq$, $=$, $\neq$, $>$ and $\geq$ which characterize the partial ordering of the objects of this type compatible with the *subtype* relation as defined before.[4] (See also Section 10 about type casting.)

[3]In C++ parlance, all methods in $\mathcal{LL}$ are *virtual*.

[4]Notice that '$T \leq root$' for all types T in $\mathcal{LL}$ and that the type *root* possesses a function method *TypeOf* which is thus inherited by all types and which returns the

4 Variables, constants and formal parameters

Variables, constants and formal parameters are denoted in $\mathcal{LL}$ by simple identifiers. In general, there is no fixed type associated with them unless explicit type checking is desired.

Variables do not have to be declared and have as their scope routine bodies (procedures, functions, constructors, methods or module bodies) in which they appear, i. e. there are no global variables. They are bound to objects through the execution of an assignment statement (see Section 6). This binding can change at any time during the execution. An uninitialized variable is bound to a special undefined object of undefined type which has no methods and cannot be used in any context.

Optional variable declarations can appear only at the beginning of routine bodies which become their scopes, and are of the form:

var *ident1*, *ident2*, ... *type-restriction*;

where *type-restriction* is optional. Variables can be used without being declared; an identifier is considered to stand for a variable within the scope of a routine body if it appears as the object (lefthand side) of an assignment statement.

Constants are very similar to variables, except that they are bound to objects through constant declarations of the form:

const *ident* *type-restriction* **is** *expr*;

and this binding is permanent within their scope. Constants can be declared only within modules and can be exported to other modules (see Section 8);[5] *type-restriction* is optional.

type tag of its argument. In particular '*TypeOf(T) = typetag*' for any type T.

[5] Notice that the word *constant* refers within this context to the constant binding; the value of the object to which the constant is bound can be modified by its methods.

Formal parameters of routines (procedures, functions, constructors and methods) can be specified by their identifiers, optionally followed by a *type restriction*:

ident1, *ident2*, ... : *T*;
ident1, *ident2*, ... <: *T*;

The first form declares that the value of the actual parameters must be of the type *T*; the second form declares that their types must be subtypes of *T*. When no type restriction is specified, all these parameters must have types which are subtypes of *root*. The binding between the formal and actual parameters follows the same rules as the binding of constants, i. e. they cannot be assigned to within the routines, but their internal values can be modified through their methods.

5 Expressions

Expressions prescribe computations that produce values (objects). Expressions are either simple (variables, constants or formal parameters), or are formed by operations applied to operands which are simpler expressions. Operations can be functions, constructors[6] or methods (of function or operator kind). The order of argument evaluation of an operation is undefined.

Syntactically, operations have priorities which in decreasing order are:

f(x,y,...)	functional application
$+$, $-$, $*$, #, @, $\sim$, **not**	unary prefix operators
!, ?, $\uparrow$	unary postfix operators
$*$, /, %, **div**, **mod**	binary infix operators
$+$, $-$, &	binary infix operators

[6]Some constructors are invoked by denotations described in Section 10.

$=, \neq, <, \leq, >, \geq$	binary infix operators
and	binary infix operator
or	binary infix operator
..	binary infix operator

Priorities can be overridden by using the parentheses '(' and ')'.

Even though some of the operators denote familiar operations in many programming languages, they have no implicit meaning in $\mathcal{LL}$. They are always treated as methods and their meaning depends on the type of their first argument. All infix operators are left associative.

In particular, function calls have the general form:

qualified-id (expr, expr, ...)

where *qualified-id* denotes a function or a function method name (see Section 8 about qualified names). If *qualified-id* denotes a function name known within the current scope, with the same arity as the number of arguments, then the function is statically selected; otherwise, it is assumed that the type of the first argument expression possesses a function method of that name (a simple identifier in this case), which will be selected dynamically at run time. If *qualified-id* is of the form *type.name* or *module.type.name* then the constructor or function method corresponding to *name* is statically selected. See also Section 7 for the description of parameter passing mechanism and function execution.

Operators are always treated as methods, i. e. their selection is dynamic, and follows the same rules of function methods. Constructors declared in a type declaration are treated like normal functions, i. e. selected statically.[7]

[7] An optimizing compiler may eliminate some of the dynamic method selection if the type of the first argument can be determined statically.

6 Statements

$\mathcal{LL}$ provides statements similar to those available in many languages, but in some cases they have a somewhat different meaning. A statement sequence is a series of statements separated by semi-colons.

Assignment statement

The general form of an *assignment statement* is:

$$ident \leftarrow expr$$

The effect of the execution of this statement is to *bind* the variable denoted by a simple identifier to the object produced by the evaluation of the expression. As a consequence of this definition, two variables can refer to (i. e. share) the *same* object. The execution of the assignment statement may include type checking if the variable *ident* was declared with a type restriction.

Conditional statement

The general form of a conditional statement is:

```
if  expr1
    then   StatementSequence1
    elsif  expr2  then  StatementSequence2
    elsif  expr3  then  StatementSequence3
    ...
    else   StatementSequence
endif
```

where parts **elsif** and **else** are optional. During the execution of an **if** statement, the expressions $expr_1$, $expr_2$, ... are evaluated in this order. If $expr_k$ is the first one of them such that '$expr_k$=*true*' produces the constant object *true* then the $\mathit{StatementSequence}_k$ is executed and the conditional statement is terminated. If no such expr_k is found then the *StatementSequence* following **else** is executed. Finally, if no **else** part is specified then it is equivalent to an empty statement.

Selection statement

The general form of a selection statement is:

```
case  expr  of
  | expr1 ⇒ StatementSequence1
  | expr2 ⇒ StatementSequence2
  | expr3 ⇒ StatementSequence3
  ...
  else  StatementSequence
endcase
```

Such a statement can be approximately described by the conditional statement:

```
X ← expr;
if X=expr1
   then  StatementSequence1
   elsif  X=expr2  then  StatementSequence2
   elsif  X=expr3  then  StatementSequence3
   ...
   else   StatementSequence
endif
```

where *X* is a new identifier not used within the same scope. However if the **else** part is missing and none of the tests is satisfied, then a run time error occurs. Also, the order of the tests of the expressions $expr_1$, $expr_2$, ... is undefined, so that if two of them produce objects which satisfy the method '=' for *expr*, then the result may be unpredictable.[8]

Iterative statements

The general forms of iterative statements are:

```
loop
   StatementSequence
endloop

while expr do
   StatementSequence
endwhile

repeat
   StatementSequence
until expr

forall var in expr do
   StatementSequence
endfor
```

A **loop** statement denotes an endless repetition of its *StatementSequence* until it is interrupted, in general by the execution of an **exit** or of a **return** statement; **while** and **repeat** statements can be expressed by equivalent **loop** statements:

[8] An optimizing compiler may take advantage of this definition when the case labels are constants of a known scalar type, and use a jump table implementation.

```
loop (* while *)
    if not $expr$ then exit endif;
    $StatementSequence$
endloop

loop (* repeat *)
    $StatementSequence$;
    if $expr$ then exit endif
endloop
```

A **forall** statement can be expressed as an equivalent **while** statement:

```
$X \leftarrow expr$;
while not $Empty(X)$ do
    $var \leftarrow Next(X)$;
    $StatementSequence$
endwhile
```

where X is a new identifier not used within the same scope. It should be noticed that the language assumes that the expression *expr* of a **forall** statement produces an object which possesses methods *Empty* and *Next*. Such objects are called *iterators* but they do not have any special status within the language. The programmer should remember that the state of an iterator will be usually modified by the application of its method *Next*, so that the same iterator cannot be used, in general, more than once; it can however be recalculated.[9]

[9] Notice however the *Reset* method of the predeclared iterator type *range*.

Break statements

The ocurrence of an **exit** statement terminates the execution of the current innermost iterative statement; it is an error to use this statement otherwise.

The ocurrence of a statement of the form '**return** *expr*' terminates the execution of the current routine (procedure or function) or of the routine underlying the implementation of a constructor or a method of a type, and returns to the calling routine. The expression *expr* is mandatory in case of a function and cannot appear in a procedure. The object resulting from its evaluation is returned by the corresponding function call.

Procedure call

The general form of a procedure call is:

qualified-id (expr, expr, ...)

where *qualified-id* denotes a procedure (or a procedure method) name (see Section 8 about qualified names). If *qualified-id* denotes a procedure name known within the current scope, with the same arity as the number of arguments, then the procedure is statically selected; otherwise, it is assumed that the type of the first argument expression possesses a procedure method of that name (a simple identifier in this case), which will be selected dynamically at run time.[10] If *qualified-id* is of the form *type.name* or *module.type.name* then the procedure method corresponding to *name* is statically selected. See also Section 7 for the description of parameter passing mechanism and procedure execution.

Empty statement

Empty statements can be used as a syntactic convenience.

[10]See however the footnote about optimization on page 13.

7 Routines

Routines (procedures and functions) in *𝓛𝓛* are similar to those found in other languages. Their declarations have the following general forms:

> **procedure** *ident*(*arg1*; *arg2*; ...) **is**
>   *RoutineBody*
> **endprocedure**;

and

> **function** *ident*(*arg1*; *arg2*; ...) *type-restriction* **is**
>   *RoutineBody*
> **endfunction**;

where *arg1*, *arg2*, ..., denote sequences of formal parameters optionally followed by their type restrictions as described in Section 4; *type-restriction* in a function declaration is also optional (if not present it is assumed to be '<: *root*') and specifies the type restriction on the value returned by the function. Routine body usually consists of optional variable declarations and of a statement sequence.

Another kind of routines is introduced by type declarations and their constructors and methods as described in Section 3. Also, the executable body of a module (see Section 8) is considered for the purpose of this section to be an anonymous parameterless procedure.

When a routine is called, its formal parameters are bound as constants to the objects resulting from the evaluation of the actual parameters and control passes to the invoked routine. The execution of the routine is usually terminated by the execution of a **return** statement. In case of a function (or function or operator method), the **return** statement must contain an expression whose evaluation produces the value to be returned. Notice that even though the parameters are passed as constant values, they denote objects whose internal state can be modified during the execution of the routine by some of the objects' methods.

All formal parameters of a routine and all variable names used within a routine have local scopes limited to the *RoutineBody*. There are no non-local variables in $\mathcal{LL}$.

Within *linking modules*, the routine body can be replaced by an implementation string which will be assumed to be written in an underlying implementation language and not interpreted by the compiler (see the local implementation guide).

Function calls are described in Section 5 and procedure calls in Section 6.

8 Modules

Modules are the basic units of compilation and control identifier visibility. Their general form is:

> **module** *ident* **is**
>   *ImportExportClauses*
>   *DeclarationSequence*
> **begin**
>   *RoutineBody*
> **endmodule**

where *ident* is the name of the module. Usually the module will be implemented as a file named *ident.ll* but a specific implementation of ℒℒ may use a different approach. The symbol **linking** may precede the symbol **module**. In this case, implementation strings can be used to specify routine bodies and type constructors and methods. It is assumed that the compiler will be given the information where to find the necessary modules written in an implementation language to be linked together with the generated code (see the local implementation guide). The symbol **begin** may be omitted if *RoutineBody* is empty.

Import and export clauses

Import and export clauses control visibility of identifiers declared within and outside the given module. Their general forms are:

> **import** *ident1*, *ident2*, ... ;
>
> **from** *ident0* **import** *ident1*, *ident2*, ... ;
>
> **from** *ident0* **import** **all**;
>
> **export** *ident1*, *ident2*, ...;
>
> **export** **all**;

The first form of an import clause makes the names of the modules $ident_1$, $ident_2$, ... visible within the module in which the clause appears. Any identifiers exported by such imported modules can be referred to by their qualified names (see naming rules further ahead). The second form of an import clause makes the identifiers $ident_1$, $ident_2$, ... exported by the module $ident_0$ visible within the module in which the clause appears; it is an error to import an unexported identifier. Finally, the third form of an import clause makes all the identifiers exported by the module $ident_0$ visible within the module in which the clause appears. In both cases of the **from** clause, the identifier $ident_0$ does not become visible, but it can be imported separately by an **import** clause. No direct or indirect circularity in imported modules is allowed.

The first form of an export clause exports the identifiers $ident_1$, $ident_2$, ... from the module in which the clause appears; only identifiers declared within the module can be exported. If an exported identifier is declared more than once (see overloaded routine names further ahead), all its instances are exported. The second form of an export clause exports all the identifiers declared within the module in which the clause appears.

It is assumed that for every implementation of $\mathcal{LL}$ there exists a module called *System* (see Appendix A) and that every normal module imports automatically all the identifiers exported by that module (this is equivalent to '**from** *System* **import** **all**'). This convention can be overwridden by an explicit import clause for the module *System*.

Declarations, scopes and naming rules

Each declaration within a module can introduce a *constant*, a *type* or a *routine* (procedure or function). Constant declarations are described in Section 4, type declarations in Section 3 and routine declarations in Section 7. The scope of the identifier being introduced by a declaration is the whole module. The scope of an identifier can be extended by the export/import clauses as described before.

Formal parameters and variables declared, implicitly or explicitly, in a routine body (see Section 7) are local, i. e. their scope is the body itself. Any declarations of these identifiers as constants appearing within the module or imported into it become unaccessible, unless their names are qualified. This rule does not apply to routine names.

In general an identifier can be declared only once within the same scope. However an identifier can be overloaded and stand for several functions and/or procedures as long as all the functions (and all the procedures) have different arities.[11]

A type declaration introduces only its name into the local scope.[12] All its constructors must be called by a *qualified* name of the form '*type.constructor*'. The methods of a type are usually selected dynamically unless fully qualified as described in Sections 4 and 6.

Imported identifiers must be all distinct and cannot conflict with local identifiers, except for overloaded routine names as noted above. An identifier *ident* exported by an imported module *mod* can be referred to by the *qualified* name *mod.ident*. In particular, if *type* denotes a type which declares a constructor or a method *m*, then it can be referred to by the *doubly qualified* name *mod.type.m*. The name of the current module is visible within the module itself and can be used to qualify its identifiers.

Module execution

The $\mathcal{LL}$ compiler produces executable code for each module. This code starts with the creation of the constants of type *typetag* introduced by type declarations (see Section 3), is followed by the evaluation and binding of locally declared constants (in order in which they are declared)[13] and then followed by the execution of the module body

[11] Notice that the same identifier can denote a function and a procedure with the same arity.

[12] As noticed in Section 3, this name denotes also a constant of type *typetag*.

[13] If a constant is used before it is calculated, a run-time error must occur because unevaluated constants have undefined values.

RoutineBody (optional variable declarations and a statement sequence). The modules of a program are linked in such a way that the executable code of all imported modules is executed before the executable code of the importing one. However, the code of a module is executed only once, even if it is imported by several different modules. The order of execution is undefined except for the topological order implied by this definition.

9 Programs

An $\mathcal{LL}$ program can be executed under two different modes: *batch* and *interactive*.

Under the *batch* mode, one $\mathcal{LL}$ module must be designated to the compiler as the *main* module. The linking process of the main module and of all the imported and library modules will produce the executable program. As a consequence of the rules given in Section 8, the execution of the main module will start after the execution of the bodies of the imported modules.

Under the *interactive* mode, one $\mathcal{LL}$ module must be designated to the compiler as the *interpreting* module. The linking process of the interpreting module and of all the imported and library modules will produce an executable interpreting statement oriented procedure which can be linked with an appropriate environment. Before the procedure can be used, the executable code of the interpreting module will be executed, after the execution of the bodies of the imported modules.

See the local implementation guide for for compiler options and and for details about linking with other environments.

10 Syntactic extensions

This section describes some syntactic extensions available in $\mathcal{LL}$ so that traditional denotations for constants and structured value construction and selection can be used. These extensions are translated by the $\mathcal{LL}$ compiler into predeclared constructor calls (known to the compiler) or method applications (with fixed names but determined dynamically). All these expressions return general $\mathcal{LL}$ objects, i. e. stamp-value pairs. See also Appendix A for predeclared types.

Integers, reals and strings

Integer, real and string constants can be represented by their traditional denotations like *-135*, *3.1415* or *6.03e+23* and *"this is a string"*. See Appendix B for the syntax of these denotations. Notice that some special characters appearing within strings must be preceded by the character \\.

Array constructors and indexing

Values of type *array* can be built by the constructor *array.New* or can be specified through a special denotation of the form $\{e_0,e_2,\ldots\}$, where either all e_i's are normal $\mathcal{LL}$ expressions[14] or they all are array denotations with the same depth of nesting. If d is the level of nesting of such an array denotation counting from the outermost level, then a d-dimensional array value will be constructed, with integer indices starting with the value *0* for each dimension. The ending value of the index for each dimension will be determined by the maximum number of elements existing at each dimension; missing values will be set to the same undefined value used for uninitialized variables (see Section 4). For example, the expression

$$\{\{1,2,3\},\{4,5\},\ \{6,7,8,9\}\}$$

builds a 3×4 array object which corresponds to:

$$\begin{pmatrix} 1 & 2 & 3 & \mathit{undefined} \\ 4 & 5 & \mathit{undefined} & \mathit{undefined} \\ 6 & 7 & 8 & 9 \end{pmatrix}$$

On the other hand, the expression

[14]They may however include operations on arguments which are written as special denotations.

$$\{(\{1,2,3\}),(\{4,5\}),\ (\{6,7,8,9\})\}$$

corresponds to the object:

$$\left(\begin{pmatrix} 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} 4 & 5 \end{pmatrix} \begin{pmatrix} 6 & 7 & 8 & 9 \end{pmatrix} \right)$$

which is a one-dimensional array with three elements, each one of them being an array. The expression '{}' denotes a *0*-dimensional array initialized with an undefined object (see also the predeclared type *array* in the Appendix A).

A statement of the form '$expr_0[expr_1,expr_2,\ldots] := expr$' is translated by the compiler into '$Update(expr_0,expr_1,expr_2,\ldots,expr)$'. An expression of the form '$expr_0[expr_1,expr_2,\ldots]$' is translated by the compiler into the expression '$Value(expr_0,expr_1,expr_2,\ldots)$'.

Notice that even though the predeclared type *array* owns methods named *Update* and *Value* with arbitrary arities, this extended notation can be used for any types which possess methods with these names.

Tuple constructors and field selection

Values of type *tuple* can be specified through a denotation of the form '$\ll expr_0,expr_1,\ldots \gg$' which is translated by the compiler into '$tuple.New(expr_0,expr_1,\ldots)$'.

A statement of the form '$expr_1\$ident := expr_2$' is translated by the compiler into '$Update\_ident(expr_1,expr_2)$'. An expression of the form '*expr$ident*' is translated by the compiler into the expression '*ident(expr)*'.

Notice that even though the predeclared type *tuple* owns methods named *Update_Field0*, *Update_Field1*, ... and *Field0*, *Field1*, ..., this extended notation can be used for any types which possess corresponding methods.

A Predeclared types and the module *System*

This section specifies all the predeclared types and some related constants and functions existing in the language. These declarations are introduced only when such types or functions cannot be implemented without their characteristics, like special denotations, being known to the compiler.

The specifications are given as incomplete ℒℒ type declarations, since the compiler knows their meaning. Some of the predeclared type objects are created by predefined pseudo-functions also known to the compiler. All the predeclared types and related constants and pseudo-functions are considered to have been specified within the module *System* which is automatically imported by any other module, unless specified otherwise (see Section 8). The module *System* is sketched below showing just the identifiers which are exported:

```
module System is
   export all;

   const false : boolean is ...;
   const true : boolean is ...;

   type root is ...;
   type integer <: root is ...;
   type real <: root is ...;
   type boolean <: root is ...;
   type string <: root is ...;
   type array <: root is ...;
   type procref <: root is ...;
   type funcref <: root is ...;
   type typetag <: root is ...;
   type tuple <: root is ...;
   type range <: root is ...;

   function ProcRef(f;n) : procref is ...;
   function ProcRef(f) : procref is ...;
   function FuncRef(f;n) : funcref is ...;
   function FuncRef(f) : funcref is ...;
   function TypeCast(t <: typetag; e) is ...;

   procedure SystemError(s <: String) is ...;

endmodule
```

The procedure *SystemError* is implementation dependent. Other identifiers are explained further ahead. Most of the predeclared types possess operator methods '=' and '≠'. In all of them, the second argument can be of any type. However, '=' will return *false* if the type of the second argument is not a subtype of the type of the first one; similarly, '≠' will return *true* in this case. Otherwise the actual values will be compared in their usual ways, unless noted otherwise.

In general, it is not necessary to specify any type restriction for the first argument of a method, since this type is used dynamically to determine the method to be called. However, methods can be determined statically through their qualified names and in this case a type restriction might be advisable. For this reason it is included in most predefined types.

Type *root*

For all types T in $\mathcal{LL}$ 'T <: *root*'. Consequently, all the methods of *root* are applicable to any type.

```
type root is
   constructors
     | New()
   functions
     | TypeOf(x) : typetag
     | Same(x;y) : boolean
     | Debug(x)
   procedures
     | Debug(x)
   operators
     | (x = y) : boolean
     | (x ≠ y) : boolean
endtype;
```

The two methods named *Debug* are implementation dependent and fixed for the predeclared types, but can be conveniently redeclared for any user defined type.[15] The method *Same* checks whether its two arguments *share* the same object. It should be used with care since an optimizing compiler may produce unexpected results. The constructor *New* can be used in some special situations but is in general useless.

[15] Remember that all methods are *virtual* – Section 3.

Type *integer*

There are no explicit constructors for building objects of this type so that traditional denotations should be used instead (see also Section 10).

```
**type** $integer$ **is**
    **constructors**
      | $Max() : integer$
      | $Min() : integer$
    **functions**
      | $ToReal(x <: integer) \quad : real$
      | $ToString(x <: integer) : string$
      | $Pred(x <: integer) : integer$
      | $Succ(x <: integer) : integer$
    **operators**
      | $(-\ x <: integer) : integer$
      | $(+\ x <: integer) : integer$
      | $x <: integer + y$
      | $x <: integer - y$
      | $x <: integer * y$
      | $(x <: integer$ **div** $y <: integer) : integer$
      | $(x <: integer$ **mod** $y <: integer) : integer$
      | $(x <: integer\ /\ y) : real$
      | $(x <: integer \uparrow y)$
      | $(x <: integer = y) : boolean$
      | $(x <: integer \neq y) : boolean$
      | $(x <: integer < y <: integer) : boolean$
      | $(x <: integer \leq y <: integer) : boolean$
      | $(x <: integer > y <: integer) : boolean$
      | $(x <: integer \geq y <: integer) : boolean$
      | $(x <: integer\ ..\ y <: integer) : range$
**endtype**;
```

The constructors are used to provide implementation limitations. Unless noted otherwise, all functions and operators have their usual meanings.

The resulting types of the binary operators $+$, $-$, $*$ and $\uparrow$ (exponentiation) can be *integer* or *real* depending on the type of the second argument y. The exppression 'x .. y' produces an object of the type *range* which is an iterator (see further ahead).

Type *real*

There are no explicit constructors for building objects of this type so that traditional denotations should be used instead (see also Section 10).

```
type real is
   functions
     | Floor(x <: real) : integer
     | Ceiling(x <: real) : integer
     | Round(x <: real) : integer
     | ToString(x <: real) : string
     | Log(x <: real) : real
     | Sin(x <: real) : real
     | Cos(x <: real) : real
     | Tan(x <: real) : real
     | Atan(x <: real; y <: real) : real
   operators
     | (− x <: real) : real
     | (+ x <: real) : real
     | (x <: real + y) : real
     | (x <: real − y) : real
     | (x <: real ∗ y) : real
     | (x <: real / y) : real
     | (x <: real ↑ y) : real
     | (x <: real = y) : boolean
     | (x <: real ≠ y) : boolean
     | (x <: real < y <: real) : boolean
     | (x <: real ≤ y <: real) : boolean
     | (x <: real > y <: real) : boolean
     | (x <: real ≥ y <: real) : boolean
endtype;
```

All functions and operators have their usual meanings. The second argument y of the binary operators $+$, $-$, $*$ and $\uparrow$ (exponentiation) can be of types *integer*, *real* or their subtypes; the result is always *real*.

Type *boolean*

There exist only two constant values of type *boolean* predeclared as constants *true* and *false*.

```
type boolean is
   operators
      (x <: boolean = y) : boolean;
      (x <: boolean ≠ y) : boolean;
      (x <: boolean < y <: boolean) : boolean;
      (x <: boolean ≤ y <: boolean) : boolean;
      (x <: boolean > y <: boolean) : boolean;
      (x <: boolean ≥ y <: boolean) : boolean;
      (x <: boolean and y <: boolean) : boolean;
      (x <: boolean or y <: boolean) : boolean
endtype;
```

The results of the relational operators are compatible with the order *false* $<$ *true*.

Type *string*

There are no explicit constructors for building objects of this type so that traditional denotation should be used instead (see Section 10).

```
type string is
   functions
     | Length(x <: string) : integer
     | Char(x <: string; p <: integer) : string
     | Segment(x <: string; p,l <: integer) : string
     | ToInt(x <: string) : integer
     | ToReal(x <: string) : real
     | ToArray(x <: string) : array
   operators
     | (x <: string & y <: string) : string
     | (x <: string  = y) : boolean
     | (x <: string ≠ y) : boolean
     | (x <: string < y <: string) : boolean
     | (x <: string ≤ y <: string) : boolean
     | (x <: string > y <: string) : boolean
     | (x <: string ≥ y <: string) : boolean
endtype;
```

All functions and operators have their obvious meanings. Characters in a string are considered to be numbered starting from *0*. *Char(x,p)* returns (as a one character string) the pth character in x; *Segment(x,p,l)* returns the segment (substring) of x starting at the position p, of length l; *ToArray(x)* returns an array whose elements are the characters of x (as one character strings), numbered from *0* to *Length(x)-1*. The operator & denotes string concatenation and the relational operators perform comparison according to the lexicographic order of the strings.

Type *range*

Objects of this type can be built by the operator '..' (see type *integer*) or its constructors *New*; see Section 6 for its use in an iterative statement **forall**.

```
type range is
   constructors
     | New(x,y <: integer)
     | New(x,y,s <: integer)
   procedures
     | Reset(x <: range)
   functions
     | First(x <: range) : integer
     | Last(x <: range) : integer
     | Empty(x <: range) : boolean
     | Current(x <: range) : integer
     | Next(x <: range) : integer
   operators
      (−x <: range) : range
endtype;
```

New(x,y,s) builds an iterator which produces values starting with x, ending with y, with the step equal to s; if s is not supplied, it is assumed to be equal to *1*. Methods *First* and *Last* produce the first and the last value in the range. The method *Current* produces the current value in the range and *Next* produces the same value but advances, as a side effect, the current value by the value of the step. The first value produced by *Current(x)* and by *Next(x)* is equal to *First(x)*, if *Empty(x)* produces *false*; otherwise, it produces the undefined object which should normally cause a run time error. Notice that the method *Next* modifies the internal state of the object. The expression '$-x$' produces a new iterator of the type *range*, with the first and last values interchanged and with a reversed step. The procedure *Reset(x)* resets the internal values of the iterator so that it can produce again the same sequence of values.

Type *array*

Besides the constructors *New* described below, a special denotation can be used for building objects of type *array* (see Section 10 for these denotations and also for extended index notations).

```
type array is
   constructors
     | New(l1,u1,l2,u2,...,lk,uk <: integer)
     | New(l1,u1,l2,u2,...,lk,uk <: integer; e)
   functions
     | Dim(x <: array) : integer
     | First(x <: array; n <: integer) : integer
     | Last(x <: array; n <: integer) : integer
     | First(x <: array) : integer
     | Last(x <: array) : integer
     | Value(x <: array; i1,i2,...,ik : integer)
     | Subarray(x <: array; i1,i2,...,ip <: integer) : array
     | Values(x <: array)
     | Items(x <: array)
   procedures
     | Update(x <: array; i1,i2,...,ik <: integer; e)
   operators
     | (x <: array = y) : boolean
     | (x <: array ≠ y) : boolean
endtype;
```

There exist families of constructors and methods for this type, for all $k \geq 0$. Notice that an array of dimension *0* behaves like a pointer in other programming languages. A constructor of the form $New(l_1,u_1,l_2,u_2,\ldots,l_k,u_k,e)$ returns an array object of dimension k, lower and upper bounds for each index given by l_i and u_i, and with all its elements initialized to the *same* object produced by the expression e. When the expression e is missing, the same undefined value used for uninitialized variables is assumed (see Section 4)

Dim(x) returns the dimension of the array x; *First(x,n)* and *Last(x,n)* return the lower and upper bounds of the nth index; when n is not specified it is assumed to be *1*. Methods named *Value* produce value of the component of the array with given indices; *Update* update the corresponding component with the value of the expression e.

Given an object x of type *array* with dimension n, *Subarray*$(x, i_1, i_2, \ldots, i_p)$, $p \leq n$, returns an array object of dimension $n - p$ produced after the indices i_1, i_2, $\ldots$, i_p are applied. For instance, if x is a rectangular array then *Subarray(x,i)* returns its ith line. The elements of the array returned by this function are *shared* with those of the original array. Notice also that if $n = p$, the returned object is a *0*-dimensional array equivalent to a pointer to the indexed element.

Values and *Items* are *iterators* (see Section 6) which produce, in lexicographic order of the indices, the values and the references to all elements of the array; these references are of the type *array* with dimension *0*, so that the functions *Update* and *Value* can be used and consequently extended indexed notation like for instance '$x[] \leftarrow e$'. In this last example, an element of the original array would be updated.

The expression '$x = y$' produces *true* if y is of a type which is subtype of *array*, with the same dimension and index bounds as x, and if all the corresponding components of x and y produce *true* under *their* method '='. Otherwise, the result is *false*. The result of '$x \neq y$' is the complement of the one described above.

Type *tuple*

There are constructors for building objects of this type but the extended denotations can also be used (see Section 10).

```
type  tuple  is
    constructors
      | New(e1;e2;...)
    functions
      | Length(x <: tuple)
      | Field0(x <: tuple)
      | Field1(x <: tuple)
        ...
    procedures
      | Update_Field0(x <: tuple;e)
      | Update_Field1(x <: tuple;e)
        ...
    operators
      | (x <: tuple = y) : boolean
      | (x <: tuple ≠ y) : boolean
endtype;
```

There exist families of constructors and methods for this type, for all $k \geq 0$. The mehod *Length* returns the number of components in a tuple. Methods *Field0*, *Field1*, ... return the value of the corresponding components; methods *Update_Field0*, *Update_Field1*, ... update the corresponding component with the value of the expression e.[16] Operators '=' and '≠' are similar to those for arrays and applied componentwise to tuples of the same length.

[16]See the extended syntax in Section 10.

Types *procref* and *funcref*

$\mathcal{LL}$ provides *pseudo-constructors* which produce references to procedures and functions (closures) ; they are invoked by *ProcRef(proc-name,nargs)* and *FuncRef(func-name,nargs)* and produce objects of the predeclared types *procref* and *funcref*. The second argument *nargs* denotes the number of formal parameters and is optional; it must be included if the routine name is overloaded.

```
type procref is
   functions
     | Arity(p <: procref) : integer
   procedures
     | Execute(p <: procref;e1;e2;...)
endtype;

type funcref is
   functions
     | Arity(f <: funcref) : integer
     | Value(f <: funcref;e1;e2;...)
endtype;
```

Closures of type *procref* and *funcref* can be executed with arguments e_1, e_2, ... by invoking the methods *Execute* or *Value*. It is a run-time error to invoke these methods with the number of parameters which does not agree with the closure; this number can obtained through the method *Arity*.[17]

[17] Notice that if f is of the type *funcref*, then '*f[*e_1,e_2,...*]*' can be used instead of '*Value(f,*e_1,e_2,...*)*' (see Section 10).

Type *typetag*

Values of this type are produced by type declarations and can be used as constants with the same names (see Section 3).

```
type typetag is
   operators
      | $(x$ <: $typetag = y)$ : $boolean$
      | $(x$ <: $typetag \neq y)$ : $boolean$
      | $(x$ <: $typetag < y$ <: $typetag)$ : $boolean$
      | $(x$ <: $typetag \leq y$ <: $typetag)$ : $boolean$
      | $(x$ <: $typetag > y$ <: $typetag)$ : $boolean$
      | $(x$ <: $typetag \geq y$ <: $typetag)$ : $boolean$
endtype;
```

The results of the relational operators are consistent with the partial subtype ordering defined in Section 3.

Type casting

The function *TypeCast(type,expr)* returns the *same* object produced by the expression *expr* but with the type stamp *type*. However, the type stamp of any other variable, parameter, constant, or structured value component, bound to the same object does *not* change. The casting function is applicable only if '*TypeOf(expr)* $\leq$ *type*', i. e. the type of *expr* is a subtype of *type*.

B Syntax of $\mathcal{LL}$

Conventions

The syntax of $\mathcal{LL}$ is described in the usual extended context-free notation of the form:

$$
\begin{array}{ll}
\mathit{NonTerminal} \longrightarrow & \\
 & \mathit{Alternative1} \\
\Big| & \mathit{Alternative2} \\
\Big| & \mathit{Alternative3} \\
 & \ldots
\end{array}
$$

Within each alternative, the following conventions are used:

$X \;\; Y$	X followed by Y
$X \mid Y$	X or Y
$[X]$	X or empty
$\{X\}$	A possibly empty sequence of X's

where "followed by" has higher precedence than $\mid$; parentheses **(** and **)** can be used to override this precedence. Notice that the symbols $\Big|$ and $\mid$ have the same meaning, but the first one is used to stress some more distinct alternatives. Nonterminals whose names start with *Special* denote syntactic extensions described in Section 10.

Modules

Module $\Longrightarrow$

  [**linking**] **module** *Ident* **is**
  ImportExportClauses
  DeclarationSequence
  [**begin**
    RoutineBody]
  endmodule

ImportExportClauses $\Longrightarrow$

  [**from** *Ident*] **import** (*IdentList* | **all**);
| **export** (*IdentList* | **all**);

Declarations

$\mathit{DeclarationSequence} \longrightarrow$
$\quad \{\ \mathit{ConstantDeclaration} \mid \mathit{TypeDeclaration} \mid \mathit{RoutineDeclaration}\ \}$

$\mathit{ConstantDeclaration} \longrightarrow \mathbf{const}\ \mathit{Ident}\ [\mathit{TypeRestriction}]\ \mathbf{is}\ \mathit{Expression};$

$\mathit{TypeDeclaration} \longrightarrow$
$\quad \mathbf{type}\ \mathit{Ident}\ [\ \mathbf{subtypeof}\ \mathit{QualifiedIdent}\ ]\ \mathbf{is}$
$\quad\ [\ \mathbf{constructors}\ \{\ \mathit{ConstrSpec}\ \}\ ]$
$\quad\ [\ \mathbf{procedures}\ \{\ \mathit{ProcSpec}\ \}\ ]$
$\quad\ [\ \mathbf{functions}\ \{\ \mathit{FuncSpec}\ \}\ ]$
$\quad\ [\ \mathbf{operators}\ \{\mathit{OpSpec}\ \}\ ]$
$\quad \mathbf{endtype};$

$\mathit{ConstrSpec} \longrightarrow \mid \mathit{Ident}(\mathit{Args})\ [\mathit{TypeRestriction}]\ \Rightarrow\ \mathit{RoutineBody}$

$\mathit{FuncSpec} \longrightarrow \mid \mathit{Ident}(\mathit{MethArgs})\ [\mathit{TypeRestriction}]\ \Rightarrow\ \mathit{RoutineBody}$

$\mathit{ProcSpec} \longrightarrow \mid \mathit{Ident}(\mathit{MethArgs})\ \Rightarrow\ \mathit{RoutineBody}$

$\mathit{OpSpec} \longrightarrow$
$\quad \mid (\mathit{PrefixOp}\ \mathit{OpArg})\ [\mathit{TypeRestriction}]\ \Rightarrow\ \mathit{RoutineBody}$
$\mid \quad \mid \mathit{PrefixOp}\ \mathit{OpArg}\ \Rightarrow\ \mathit{RoutineBody}$
$\mid \quad \mid (\mathit{OpArg}\ \mathit{InfixOp}\ \mathit{OpArg})\ [\mathit{TypeRestriction}]\ \Rightarrow\ \mathit{RoutineBody}$
$\mid \quad \mid \mathit{OpArg}\ \mathit{InfixOp}\ \mathit{OpArg}\ \Rightarrow\ \mathit{RoutineBody}$
$\mid \quad \mid (\mathit{OpArg}\ \mathit{PostfixOp})\ [\mathit{TypeRestriction}]\ \Rightarrow\ \mathit{RoutineBody}$
$\mid \quad \mid \mathit{OpArg}\ \mathit{PostfixOp}\ \Rightarrow\ \mathit{RoutineBody}$

$Arg \longrightarrow IdentList\ TypeRestriction \mid Ident$

$Args \longrightarrow \{\ Arg;\ \}\ Arg \mid Empty$

$MethArgs \longrightarrow \{\ Arg;\ \}\ Arg$

$OpArg \longrightarrow Ident\ [TypeRestriction]$

$TypeRestriction \longrightarrow\ <: Ident \mid : Ident$

Statements

```
StatementSequence ⟶ { Statement; } Statement

Statement ⟶
          AssignmentStatement
        | SpecialAssignment
        | ConditionalStatement
        | SelectionStatement
        | BreakStatement
        | IterativeStatement
        | ProcedureCallStatement
        | Empty

AssignmentStatement ⟶ Ident ← Expression

SpecialAssignment ⟶
          Expression[ExpressionList] ← Expression
        | Expression $ Ident ← Expression

ConditionalStatement ⟶
        if Expression
          then StatementSequence
         { elsif Expression then StatementSequence }
         [ else StatementSequence ]
        endif

SelectionStatement ⟶
        case Expression of
         { | Expression ⇒ StatementSequence }
         [ else StatementSequence ]
        endcase

BreakStatement ⟶ exit | return [ Expression ]
```

IterativeStatement $\Longrightarrow$
  loop *StatementSequence* **endloop**
| **while** *Expression* **do** *StatementSequence* **endwhile**
| **repeat** *StatementSequence* **until** *Expression*
| **forall** *Ident* **in** *Expression* **do** *StatementSequence* **endforall**

ProcedureCallStatement $\Longrightarrow$ *QualifiedIdent*(*ExpressionList*)

Routines

$RoutineDeclaration \Longrightarrow FunctionDeclaration \mid ProcedureDeclaration$

```
FunctionDeclaration ⟹
        function Ident(Args) TypeRestriction is
          RoutineBody
        endfunction;

ProcedureDeclaration ⟹
        procedure Ident(Args) is
          RoutineBody
        endprocedure;

RoutineBody ⟹
        VariableDeclarations StatementSequence
    |   ImplementationString

VariableDeclarations ⟹ { var IdentList [TypeRestriction]; }
```

Expressions

ExpressionList $\longrightarrow$ { *Expression*, } *Expression* | *Empty*

Expression $\longrightarrow$ *SpecialExpression*

SpecialExpression $\longrightarrow$ *SpecialExpression1* [*[ExpressionList]*]

SpecialExpression1 $\longrightarrow$ *NormalExpression* {*$Ident*}

NormalExpression $\longrightarrow$ *Expression1* { *RangeOp Expression1* }

Expression1 $\longrightarrow$ *Expression2* { **or** *Expression2* }

Expression2 $\longrightarrow$ *Expression3* { **and** *Expression3* }

Expression3 $\longrightarrow$ *Expression4* { *RelationalOp Expression4* }

Expression4 $\longrightarrow$ *Expression5* { *AdditiveOp Expression5* }

Expression5 $\longrightarrow$ *Expression6* { *MultiplicativeOp Expression6* }

Expression6 $\longrightarrow$ *Expression7* { *PostfixOp* }

Expression7 $\longrightarrow$ { *PrefixOp* } *Expression8*

Expression8 $\longrightarrow$
QualifiedIdent | *QualifiedIdent*(*ExpressionList*) | *SpecialDenotation*

RangeOp $\Rightarrow$ ..

RelationalOp $\Rightarrow$ $= \mid \neq \mid < \mid \leq \mid > \mid \geq$

AdditiveOp $\Rightarrow$ $+ \mid - \mid \&$

MultiplicativeOp $\Rightarrow$ $* \mid / \mid \% \mid$ **div** $\mid$ **mod**

BooleanOp $\Rightarrow$ **or** $\mid$ **and**

InfixOp $\Rightarrow$
RangeOp $\mid$ *RelationalOp* $\mid$ *AdditiveOp* $\mid$ *MultiplicativeOp* $\mid$ *BooleanOp*

PrefixOp $\Rightarrow$ $+ \mid - \mid * \mid \# \mid @ \mid \sim \mid$ **not**

PostfixOp $\Rightarrow$ $! \mid ? \mid \uparrow$

Miscellanea

$$IdentList \Longrightarrow \{\ Ident,\ \}\ Ident$$

$$QualifiedIdent \Longrightarrow [Ident.[Ident.]]Ident$$

$$Ident \Longrightarrow Letter\ \{\ Letter \mid Digit \mid Underscore\ \}$$

$$\begin{aligned} SpecialDenotation \Longrightarrow\ & \\ & IntegerDenotation \\ \mid\ & RealDenotation \\ \mid\ & StringDenotation \\ \mid\ & ArrayDenotation \\ \mid\ & TupleDenotation \end{aligned}$$

$$IntegerDenotation \Longrightarrow Digit\{Digit\}$$

$$RealDenotation \Longrightarrow IntegerDenotation.[IntegerDenotation][Exponent]$$

$$Exponent \Longrightarrow (E \mid e)[+ \mid -]IntegerDenotation$$

$$StringDenotation \Longrightarrow {}''\{StringChar \mid EscapeChar \mid {}'\ \}''$$

$$ImplementationString \Longrightarrow {}'\{ImplStringChar\}'$$

$$ArrayDenotation \Longrightarrow \{\text{ExpressionList}\}$$

$$TupleDenotation \Longrightarrow \ll ExpressionList \gg$$

Letter $\Longrightarrow$ A | B | ... | Z | a | b | .. | z

Digit $\Longrightarrow$ 0 | 1 | ... | 9

Underscore $\Longrightarrow$ _

EscapeChar $\Longrightarrow$ \\$n$ | \\$t$ | \\$r$ | \\' | \\$f$ | \\\\ | \\"

StringChar $\Longrightarrow$

Any character allowed by the implementation except for the escape characters.

ImplStringChar $\Longrightarrow$

Any printible character allowed by the implementation except for the character '.

Empty $\Longrightarrow$

C Symbol representation

This report uses some printing conventions which improve its readability (see Section 2). However, when actual program files are prepared, some symbols must be replaced by sequences of common ASCII characters. In the case of reserved words, they may be written as any combination of lower and upper case letters. Some of the special symbols and operators are replaced by character sequences shown below:

Symbol	Representation
$\neq$	`<>`
$\leq$	`<=`
$\geq$	`>=`
$\uparrow$	`^`
..	`..`
$\ll$	`<<`
$\gg$	`>>`
$\leftarrow$	`:=`
$\Rightarrow$	`=>`

All other symbols and operators remain unchanged.

D Example

```
linking module List is

import Algorithms;

export all;

type reference subtypeof array is  endtype;

type  List  is
   constructors
     | New()  ⇒  'LL_RTS_Push(implNewList()'
   functions
     | Empty(L <: List) <: boolean  ⇒  'LL_RTS_Push(implEmpty(L));'
     | First(L <: List) <: reference  ⇒  'LL_RTS_Push(implFirst(L));'
     | Last(L <: List) <: reference  ⇒  'LL_RTS_Push(implLast(L));'
     | Succ(L <: List;I <: reference) <: reference  ⇒
                                       'LL_RTS_Push(implSucc(L,I));'
     | Pred(L <: List;I <: reference) <: reference  ⇒
                                       'LL_RTS_Push(implPred(L,I));'
     | Value(L <: List;I <: reference)  ⇒  'LL_RTS_Push(implValue(L,I));'
     | Insert(L <: List;E) <: reference  ⇒  'LL_RTS_Push(implInsert(L,E));'
     | Search(L <: List;E) <: reference  ⇒  'LL_RTS_Push(implSearch(L,E));'
     | Concat(L1 <: List;L2 <: List) <: List  ⇒
                                       'LL_RTS_Push(implConcat(L1,L2));'
     | Items(L <: List)  ⇒  'LL_RTS_Push(implItems(L));'
     | Values(L <: List)  ⇒  'LL_RTS_Push(implValues(L));'
     | Max(L <: List)  ⇒  return Algorithms.MaxElement(L)
   procedures
     | Insert(L <: List;E)  ⇒  'implInsert(L,E);'
     | Delete(L <: List;E)  ⇒  'implDelete(L,E);'
     | Update(L <: List;I <: reference; E)  ⇒  'implUpdate(L,I,E);'
     | Sort(L <: List)  ⇒  Algorithms.BubbleSort(L)
   operators
     | (L1 <: List & L2 <: List) <: List  ⇒
                                       'LL_RTS_Push(implConcat(L1,L2));'
endtype;

endmodule
```

It is assumed in this example that a list manipulation library implemented in a language like C++ is available, and that the module *List* implements a simple $\mathcal{LL}$ interface for this library.

By convention, all the routines denoted here by identifiers starting with '*impl-*' are written in the implementation language and provided by the interface implementor – this is the reason for the occurrence of the symbol **linking**. Several of these routines use and return objects of type *reference*, which in this case is a subtype of the predeclared type *array*. It is expected that all these objects are zero-dimensional arrays which behave like references in other programming languages (see type *array* in Appendix A). *LL_RTS_Push* is a procedure provided by the implementation which pushes its argument (necessarily an $\mathcal{LL}$ object) on top of the run-time stack (see the local implementation guide).

Most of the methods are self-explanatory. *Items* is intended to return an iterator which produces references to all elements of the list, whereas *Values* is an iterator which produces all of its elements. The methods *Sort* and *Max* are implemented by calling $\mathcal{LL}$ routines *BubbleSort* and *MaxElement*, presumably declared in a module called *Algorithms*, and imported by this module. The routines could be coded as:

```
procedure BubbleSort(L) is
   forall i in Last(L) .. Succ(First(L)) do
      forall j in First(L) .. Pred(i) do
         if L[j]>L[Succ(j)] then
            t ← L[j]; L[j] ← L[Succ(j)]; L[Succ(j)] ← t
         endif
      endforall
   endforall
endprocedure;
```

and

```
function $MaxElement(L)$ is
    $max \leftarrow L[First(L)]$;
    forall $v$ in $Values(L)$ do
        if $max<v$ then $max \leftarrow v$ endif
    endforall;
    return $max$
endfunction;
```

It should be noticed that these routines will work for a variety of objects as long as they possess certain methods with a convenient semantics. In particular, they can be used for ordinary arrays of objects.

Relatórios Técnicos – 1992

01/92 **Applications of Finite Automata Representing Large Vocabularies,** *C. L. Lucchesi, T. Kowaltowski*

02/92 **Point Set Pattern Matching in d-Dimensions,** *P. J. de Rezende, D. T. Lee*

03/92 **On the Irrelevance of Edge Orientations on the Acyclic Directed Two Disjoint Paths Problem,** *C. L. Lucchesi, M. C. M. T. Giglio*

04/92 **A Note on Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams,** *W. Jacometti*

05/92 **An (l, u)-Transversal Theorem for Bipartite Graphs,** *C. L. Lucchesi, D. H. Younger*

06/92 **Implementing Integrity Control in Active Databases,** *C. B. Medeiros, M. J. Andrade*

07/92 **New Experimental Results For Bipartite Matching,** *J. C. Setubal*

08/92 **Maintaining Integrity Constraints across Versions in a Database,** *C. B. Medeiros, G. Jomier, W. Cellary*

09/92 **On Clique-Complete Graphs,** *C. L. Lucchesi, C. P. Mello, J. L. Szwarcfiter*

10/92 **Examples of Informal but Rigorous Correctness Proofs for Tree Traversing Algorithms,** *T. Kowaltowski*

11/92 **Debugging Aids for Statechart-Based Systems,** *V. G. S. Elias, H. Liesenberg*

12/92 **Browsing and Querying in Object-Oriented Databases,** *J. L. de Oliveira, R. de O. Anido*

13/93 **Modelling Geographic Information Systems using an Object Oriented Framework,** *Fatima Pires, Claudia Bauzer Medeiros, Ardemiris Barros Silva*

14/93 **Managing Time in Object-Oriented Databases,** *Lincoln M. Oliveira, Claudia Bauzer Medeiros*

15/93 **Using Extended Hierarchical Quorum Consensus to Control Replicated Data: from Traditional Voting to Logical Structures,** *Nabor das Chagas Mendonça, Ricardo de Oliveira Anido*

Departamento de Ciência da Computação — IMECC
Caixa Postal 6065
Universidade Estadual de Campinas
13081-970 – Campinas – SP
BRASIL
`reltec@dcc.unicamp.br`