A Revised Leda Language Definition

Rajeev Pandey
Wolfgang Pesch
Jim Shur
Masami Takikawa

rpandey@cs.orst.edu
wpesch@hpbbi4.bbn.hp.com
jims@roguewave.com
takikawm@cs.orst.edu

Technical Report 93–60–02
Department of Computer Science
Oregon State University
Corvallis, OR
97331–3202

January 29, 1993

Contents

Summary

This report describes the revised definition of the multiparadigm programming language Leda.

The first section provides an introduction to Leda and a history of its development. Section 2 covers Leda preliminaries, section 3 details the overall structure of Leda programs. Declarations are discussed in section 4, expressions are the topic of section 5. Section 6 examines Leda statements, with relational programming being the subject of section 7, and section 8 comprises of a very short discussion of constraint logic programming. Leda predefined classes are briefly described in section 9, with a short note on scoping presented in section 10 and libraries in section 11. Compiler usage is touched upon in section 12. Appendices listing reserved words, compile-time errors, run time errors, and a bibliography conclude this report.

1 Introduction

Leda is a strongly typed, compiled, multiparadigm programming language designed by Timothy A. Budd. Programming paradigms included in Leda include the procedural, relational, functional and object-oriented paradigms. Constraint logic programming is also facilitated in the revised version of Leda. Leda was designed and implemented in order "to provide a vehicle for experiments in multi-paradigm programming" and to "explore the advantages of using various programming paradigms on assorted problems" [Bud89a]. Leda is an evolving research language and readers may wish to consult the bibliography for a variety of papers concerning its *raison d'être*. Our purpose here is to guide investigators in the use of a particular Leda compiler, not to suggest what Leda should finally be.

1.1 Background

This document defines the language Leda as currently implemented by Masami Takikawa. This version of Leda is a revised version of the original Leda language implementation done by Vinoo Cherian, Wolfgang Pesch and Jim Shur [PeS91]. The syntax will be given in BNF notation with accompanying semantics in English. Some simple examples are given within the document, usually illustrating specific aspects of the language.

1.2 Availability

Leda is available via anonymous ftp from cs.orst.edu (128.193.32.1), in the directory pub/budd. Both the original version of Leda and the version discussed in this document are available at this location. Both versions are accompanied by sample programs and documentation.

1.3 Copyright

All files in this Leda distribution are distributed under the following copyright notice:

The author can be contacted at:
takikawm@research.cs.orst.edu or
logic@bug.co.jp (from Japan)

1.4 Acknowledgements

The design, development and implementation of Leda has been supported by the Department of Computer Science at Oregon State University. Vinoo Cherian, Rajeev Pandey, Wolfgang Pesch and Jim Shur received support via Graduate Teaching Assistantships from the department. Wolfgang Pesch also received support from the German government.

Timothy Budd, Robert Hunter, Timothy Justice, Sharon Shoshony and Nabil Zamel provided useful comments on preliminary drafts of this document.

2 Preliminaries

2.1 Nothing

<empty> ::=

The non-terminal <empty> represents a sequence of zero symbols.

2.2 Whitespace

Whitespace—spaces, tabs, and newlines—is ignored except where it serves to separate tokens. The language is free-format; as long as the grammar is followed, the placement of the program components, including indentation, is irrelevant to the compiler. [1]

2.3 Comments

There are two ways to introduce comments into a Leda program. The first uses two slashes (//) which indicate to the compiler that from that point until the end of the current line is a comment. The second is to include any number of characters inside curly braces ({}). In the latter, the comment ends at the first closing brace, so the braces may not be nested. The former may be nested within the latter, making the braces useful for commenting out blocks of code which include the single-line comments. Bracketed comments can contain any character other than a '}' i.e. only a single closing bracket is allowed in a multi-line comment.

```
// a comment extending to the end of the line
 { this is a comment that
   may span multiple lines }
```

2.4 Constants

2.4.1 Numbers

<integer constant> ::= <sign><integer part>
<integer part> ::= <nonzero digit> <digits> | **0x** <hex> | **0** <oct>
<digits> ::= <empty> | <digits> <digit>
<hex> ::= <empty> | <hex> <hex digit>
<oct> ::= <empty> | <oct> <oct digit>
<nonzero digit> ::= 1|2|3|4|5|6|7|8|9
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<hex digit> ::= 0|1|2|3|4|5|6|7|8|9|a|b|c|d|e|f|A|B|C|D|E|F
<oct digit> ::= 0|1|2|3|4|5|6|7
<sign> ::= + | − | <empty>

[1]Not to humans however, and we hope that the search for a nifty Leda programming style may soon become yet another lively area of research.

<real constant> ::= <sign>**0 .** <digit> <digits>
| <sign> <nonzero digit> <digits> .<digit> <digits>

There are two sorts of numerical constants—integers and reals. All entities in Leda are objects which are instances of some class and numbers are no exception. Integer constants are instances of the predefined class `integer`; reals belong to the predefined class `real`. An integer constant may be expressed in octal by preceding it with `0`, and in hexadecimal by preceding the integer with `0x`.

2.4.2 Characters

<character constant> ::= '<character>'

<character> ::= <letter>
| <digit>
| <other>
| <escape sequence>

<letter> ::= a|b|c|d|e|f|g|h|i|m|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
| A|B|C|D|E|F|G|H|I|M|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<other> ::= * | + | / | = | (|) | { | } | [|] | < | > | ‘ | ’ | " | ! | @ | # | $ | % | & |
_ | | | ^ | ~ | . | , | ; | :| ?
<escape sequence> ::= \b | \f | \n | \r | \t | \" | \\ | \x<hex2>|\<oct3>
<hex2> ::= <empty> | <hex digit> | <hex digit> <hex digit>
<oct3> ::= <oct digit>|<oct digit> <oct digit> |<oct digit> <oct digit> <oct digit>

A character is defined to be any single printing character enclosed within single quotes. Escape characters like newline and tab are represented as `'\n'` and `'\t'` respectively.

2.4.3 Strings

<string constant> ::= "<string>"
<string> ::= <empty> | <string> <character>

Any sequence of characters that is enclosed within double quote marks is considered a string constant. Note that `"a"` (a string) and `'a'` (a character) are different. To include a double quote character (") or a backslash (\) in a string, one must immediately precede them with a blackslash character. Strings are represented as in C i.e. the string `"abc"` can be thought of as consisting of four characters: `'a'`, `'b'`, `'c'` and `'\0'` (the null character). Strings are pointers to the character type.

2.5 Identifiers

<id> ::= <letter>
| <underscore>
| <id><letter>

| <id><digit>
| <id><underscore>

<letter> ::= a|b|c|d|e|f|g|h|i|m|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
| A|B|C|D|E|F|G|H|I|M|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<digit> ::= 0|1|2|3|4|5|6|7|8|9
<underscore> ::= _

Identifiers, represented in this document by the non-terminal <id>, consist of any number of letters and digits, where the first character must be a letter or an underscore. Note that upper and lower case letters are distinguished, so that `somename` and `someName` are two different identifiers. Since Leda compiles to C, the number of significant characters in an identifier depends on the underlying C compiler.

3 Overall Structure

<program> ::= <empty> | <program> <declarations> | <program> <compound statement> ;

<declarations> ::= <include statement> | <constant declarations>
| <type declarations> | <variable declarations>
| <subprogram declarations>

<compound statement> ::= **begin** <statement list> **end**

<statement list> ::= <statement> ;
| <statement list><statement> ;

The overall structure of a Leda program can be made to resemble the structure of a Pascal program, with a set of declarations followed by a compound statement. Leda is not restricted to the Pascal-like format, however. The compound statement may consist of one or more statements which are executed in turn. As will be seen in the descriptions below, each of the declaration sections may be empty, so that the simplest legal Leda program is the following:

```
begin
  ;
end;
```

The program is made up of only the <compound statement> containing a single, albeit empty, statement.

3.1 Include Statements

<include statement> ::= **include** <string constant>; | **include** <id>;

A Leda program can be contained across several source files. The include statement allows for simple textual substitution of files specified by <string constant> or <id>. Include statements can appear anywhere in a Leda program, though not within other declarations (of type, constants or variables) nor in the body of compound statements.

Neither separate compilation nor modules are supported at present, although module is a reserved keyword in anticipation of future developments.

4 Declarations

4.1 Constant Declarations

<constant declarations> ::= **const** <constant declaration list>

<constant declaration list> ::= <constant declaration>
| <constant declaration list> <constant declaration>

<constant declaration> ::= <id> **:=** <expression> **;**

The <expression> must evaluate to a constant at compile time. Constants can be integer, real, character, string, boolean, an enumerated type or NIL. The compiler infers the type of the identifier from the type of the constant. The value is assigned to the identifier as if a variable was declared in the <variable declarations> section and the constant declaration appeared as an <assignment statement> within the <compound statement> that follows. Identifiers declared in the <constant declarations> section may not be re-assigned.

```
const
  MAX := 8*10;    // integer constant (80)
  MIN := -6;      // negative integer constant
  PI := 3.14159;  // real constant

begin
  ;
end;
```

4.2 Type Declarations

<type declarations> ::= **type** <type declaration list>
<type declaration list> ::= <type declaration>
| <type declaration list> <type declaration>
<type declaration> ::= <id> **:=** <type> **;** | <id> **:=** <type> **: (** <type parameter list> **);**
<type parameter list> ::= <type parameter> | <type parameter list> **,** <type parameter>
<type parameter> ::= <id> | <id> **<** <type name>
<type> ::= <anonymous type> | <named type> | <pointer type>
<named type> ::= <enumerated type> | <class type>
<anonymous type> ::= <type name> | <function type> | <array>

Type declarations are similar to constant declarations in that their semantics include an implicit variable declaration and an assignment. The left-hand identifiers are implicitly declared variables of type CLASS (spoken "capital class"). CLASS is not a type which is available to the Leda programmer. All types in Leda are in fact *classes* in the object-oriented sense, and the words *type* and *class* will be used interchangeably in this document. CLASS itself is the class of classes, somewhat like a *metaclass* in Smalltalk. So types are objects whose value is determined by the right-hand side of the type declaration. An individual declaration is semantically equivalent to an assignment statement. The <type declaration> section is *not* the same as a list of assignment statements which are to be executed in order. User-defined type declarations may be made in any order, and the compiler will work out the appropriate sequence. The same name may only appear once on the left side of a type delcaration within the same type declaration section.

```
type
  intAlias := integer;  // foo and intAlias will be of the
  foo := intAlias;      // same type in this case
```

4.3 Types

4.3.1 Predefined Types

<type name> ::= <integer> | <real> | <character> | <string> | <boolean> |ˆ<named type>
<id> | <id> **:** **(** <type parameter list> **)**
<integer> ::= **integer**
<real> ::= **real**
<character> ::= **character**
<string> ::= **string**
<boolean> ::= **boolean**
<array> ::= **array** **[**<subrange>**]** **of** <type name>
<subrange> ::= <integer> | <integer> **..** <integer>

Leda has most familiar types built-in. Declaring an array to be of some size [n] is the same as declaring it of size [1..n]. A string is equivalent to a pointer to character:

```
type
  x1 : string;
  x2 : ^character;              // equivalent to the declaration of x1
  a1 : array [10] of integer;
  a2 : array [1..10] of integer;  // equivalent to the declaration of a1
```

4.3.2 Classes

<class type> ::= **class** <type parameters><superclass>
<instance member declarations>
<shared member declarations>
end

<type parameters> ::= **:** **(** <type parameter list> **)** | <empty>
<type parameter list> ::= <type parameter> | <type parameter list> **,** <type parameter>
<type parameter> ::= <id> | <id> **<** <type name>
<superclass> ::= **of** <type name> | <empty>
<instance member declarations> ::= <member declaration list> | <empty>
<shared member declarations> ::= **shared** <member declaration list> | <empty>

<member delcaration list> ::= <member declaration>
| <member declaration list><member declaration>

<member declaration> ::= <id list> **:** <member type> **;**
<member type> ::= <named type> | <method type>

All objects in Leda are instances of some class. The definition of the class determines the form of the object. Class types allow the programmer to define a new class from which instances may be created in the code section. An object consists of members which are declared within the class definition. Each instance of a class has its own instance members to which only that instance has access. A separate set of instance members exist for each instance of the class. These members define the properties which make a particular object unique within its class. Each instance of a class also has access to a set of shared members, which exist singularly for *all* instances of the class. The shared members express the commonality of the class instances. All member names within a class must be unique. The idea of dividing class members into those that define unique vs. common properties is more general than distinctions such as data vs. methods or state vs. behavior. In Leda, an object may have some behavior which is different than other members of its class, as it may share with them information in the form of any <member type>.

The value of the class definition itself, which will be assigned to the left-hand identifier in the type declaration in which it appears, is an object of type CLASS. One might imagine an implicit variable declaration taking place of the form:

```
var
  Point : CLASS;
```

All classes have instance members `parent` and `myClass` predefined. These two members are not intended for use by the user, however. The object `Point`, above, is an instance of class CLASS and consists of a shared member, `filter`, which allows for conversion between instances of user defined classes. The instance members of the class object are exactly those members which are defined in the <shared member declarations>. Put another way, the shared members of an *instance* of a class are the instance members of the object which *is* the class. The corresponding members are aliases of each other. Consider the following class definition:

```
type
  Point := class
    x : real;
    y : real;
  shared
    distance : method(Point)->real;    // method types explained below
  end;
```

Here `x` and `y` are instance members while `distance` is a shared member. If the variable `p` is an instance of class `Point`, then `p`'s shared member `distance` and `Point`'s instance member `distance` are the same object.

Class hierarchies may be constructed by including a <superclass> in the class definition. Classes may only have a single superclass. Classes inherit all instance members from their ancestor classes. It is an error to declare an instance or shared member of the same name as an instance member which is to be inherited. Shared members may be inherited as well. When a shared member is inherited, the object that is common to all instances of the class is the exact same object that is shared among instances of the superclass. Inherited shared members are not instance members of the class object as described above, only shared members explicitly declared in the class have that property. It is an error to declare an instance member of the same name as a shared member which is to be inherited. A shared member that would be inherited may be overridden by redeclaring the shared member. This provides a break with the ancestor classes in that a new object is created which is to be shared among only the instances of the class being defined, and any descendent classes that choose to inherit rather than override the shared member. All shared members from ancestor classes must be either inherited or overridden:

```
type
  parent := class
     :    // some declarations here
  shared
    func : method(argument)->result;
  end;

  child := class of parent
     :   // maybe some declarations here
  shared
    func : method(argument')->result';    // method func is being overriden
  end;
```

Note that class child is less generalized than class parent, a situation which is denoted by child $\leq$ parent. For method `func` to be overridden in class child above, type `argument'` must be the same or more general than type `argument`, i.e. argument $\leq$ argument', and type `result'` must be the same or less general than type `result` i.e. result $\geq$ result'. A paper by Budd [Bud91b] gives a more complete explanation of overriding in Leda.

The following example shows the definition of a subclass of class `Point` defined earlier:

```
type
  Point := ... // as defined previously

  Colors := (violet, blue, green, yellow, orange, red);   // enumerated type

  ColorPoint := class of Point
    color : Colors;
  shared
    invertColor : method();
  end;
```

Type parameters may be used to create a more general class description than is otherwise possible. The scope of the type parameters is the class definition, including any methods declared within the class. When type parameters are used, the class is called a *parameterized class*. Type parameters may be used freely as types within the class definition, with the exception that shared members may not be declared to be of a type-parameter type. Type parameters are defined to be classes with no instance or shared members, and those members that are declared to be instances of them can only be used accordingly. Type parameters may be *restricted* using the "<" symbol along with another type that is within the same name-space as the class being defined. In the case of these restricted parameterized classes, the type parameter is defined to be a subclass of (or in the same class as) the restricting type. The type parameter inherits, but neither adds-to nor overrides, the members of its ancestors. A parameterized class implicitly creates an entire set of classes which are then available to the programmer. These are the classes that can be formed by substituting actual type arguments for each formal type parameter as explained in the section covering type names. This higher level of abstraction allows the programmer to capture some of the commonality of these classes in a single definition.

To allow parameterized types while still maintaining type-safety, the compiler prohibits overriding a shared member which makes use of a type parameter.

```
type
  List := class:(T)
    first : T;
    rest : List:(T);
  shared
    head : method()->T;      // these methods are not allowed
    append : method(T);      // to be overridden
  end;
```

The `List` class above defines a whole set of `List` classes, each representing a list of some definite type. All will have `first` and `rest` as instance members, and each will be able to utilize the common generic methods `head` and `append`, to which the particular type of list is irrelevant.

4.3.3 Enumerated Types

<enumerated type> ::= **(** <id list> **)**
<id list> ::= <id> | <id list> **,** <id>

The value of an enumerated type is an object of type CLASS and is defined to be a subclass of the predefined abstract class `enum`. The section on predefined classes gives the members which are inherited or overridden by all enumerated types. The list of identifiers which comprise an enumerated type are the set of enumerated constant values which variables declared to be of the enumerated type may take on as values:

```
type
  Point := ...

  Colors := (violet, blue, green, yellow, orange, red);   // enumerated type

  ColorPoint := class of Point
    color : Colors;
  shared
    invertColor : method();
  end;
```

In the example above, the member `color` of class `ColorPoint` is declared to be of the enumerated type `Colors`. Thus it may be assigned any of the constants `red`, `yellow`, `blue`, etc. The same enumerated constant may not be declared twice throughout a program scope, i.e., they must all be unique throughout the main program or within a subprogram.

4.3.4 Function Types

<function type> ::= **function**<type arguments><return type>
                    | **function:(**<type parameter list>**)**<type arguments><return type>

<type arguments> ::= **( )** | **(** <type argument list> **)**

<type argument list> ::= <type argument>
                         | <type argument list> **,** <type argument>

<type argument> ::= <mode><named type>
<mode> ::= **var** | **lazy** | <empty>
<return type> ::= **->** <named type> | <empty>

Functions in Leda are first-class values and the <function type> is used to declare variables of type function. In addition, the types of the parameters and return value must be given. The return type may be excluded if the function does not return a value. If no <mode> is given, it means the parameter is to be passed by value. The keyword `var` denotes pass by reference. The `lazy` keyword is used for passing lazy evaluated or demand-driven functions.

```
type
  f := function(integer, real)->boolean;
```

4.3.5 Method Types

<method type> ::= **method**<type arguments><return type>
                  | **method:(** <type parameter list> **)** <type arguments><return type>

A method is a subprogram abstraction similar to functions. Variables of type method may only be declared in member declarations of class definitions. Methods in parameterized classes have the type parameter specified along with argument and return types. The difference between a function and a method is that a method may refer directly to the members of the class in which it is defined and functions may not. When a method is invoked, there is a receiver which will always be of the class in which the method is defined, or one of its derived classes. In either case, whenever the method refers to a class member, it is the corresponding member of the receiver which is used.

```
type
  foo := class
    x : integer;
  shared
    m : method()->integer;    // m may refer directly x
    f : function()->integer;  // f may not!
  end;
```

4.3.6 Type Names

<type name> ::= <id> | <id>(<named type list>)
<named type list> ::= <named type> | <named type list>,<named type>

Type names are identifiers which refer to the left-hand side of some type declaration. If the value of the identifier is a parameterized class, a <named type> argument is provided for each formal type parameter in the parameterized class. If the formal parameter is restricted, the actual type argument must be the same as, or a descendant of, the restricting type.

Providing arguments for a parameterized class definition is called *instantiation*. It is an error for a type name to refer to an uninstantiated parameterized class. The value of the type name is the value of the identifier, that is at run-time, both refer to the exact same object of type CLASS. However when instantiated types are used, at compile-time, for the purposes of type checking, the class definition referred to by the named type is taken to be the class definition associated with the identifier after substituting the named-type arguments for the corresponding formal type parameters.

```
type
  ListElement := class:(X)
      datum : X;
      next : ListElement:(X);
    shared
      add : method(X);
      remove : method(X)->ListElement:(X);
    end;

  List := class:(X)
      data : ListElement:(X);
    shared
      add : method(X);
      addNew : method(X)->List:(X);
      includes : method(X)->boolean;
      remove : method(X);
      reduce : method:(Z)(Z, function(Z, X)->Z)->Z;
    end;

// definitions for the shared methods such as add and reduce here.

var
  aList : List:(integer);

begin
  aList := List:(integer)();
  aList.add(3);  aList.add(7);  aList.add(-1);
  aList.reduce:(integer)(0, integer.plus).print();  '\n'.print();
     :
     :
```

4.3.7 Pointers

Leda has pointer semantics similar to those of C. Pointers contain the address of variable locations of a declared type. Pointer operations are carried out via two unary operators: the address operator (&) and the dereference operator (^). The address operator, when applied to a variable, returns the address of the variable. The dereference operator allows access to the variable whose memory location is stored in the pointer variable being dereferenced.

```
type
  intPtr := ^integer;            // pointer to type integer

var
  n : integer;
  h : intPtr;

cfunction printf();              // explained later: call C language printf

begin
  n := 100;
  h := &n;                       // h contains the address of n
  printf ("%d \n", h);           // print the address of n
  h^.print();                    // print the contents of the location pointed
                                 //  to by h. (i.e. value of n)
  '\n'.print();
  h^ := h^ + 100;                // add 100 to value in address pointed to by h
  h^.print();                    // print contents of location pointed to by h
  '\n'.print();
end;
```

4.4 Variable Declarations

<variable declarations> ::= **var** <variable declaration list> | <empty>

<variable declaration list> ::= <variable declaration> ;
| <variable declaration list> <variable declaration> ;

<variable declaration> ::= <id list> **:** <named type>
<id list> ::= <id> | <id list> , <id>

Leda is a strongly typed language. Each variable must be declared to be of some single, definite class before it can be used. All variables upon declaration are *undefined*; they have no value, refer to no object. No space is allocated at this time to hold an object of the declared class.

```
var
  i, j, k : integer;
  degrees : Celsius;
  p : MixedPair;
```

4.5 Subprogram Declarations

<subprogram declarations> ::= <subprogram declaration list>

<subprogram declaration list> ::= <subprogram declaration> ;
| <subprogram declaration list> <subprogram declaration> ;
<subprogram declaration> ::= <method declaration> | <function declaration>
| <cfunction declaration>

<function declaration> ::= <function header>
<declaration part>
<compound statement>

<function header> ::= **function** <name><argument list><return type> ;
| **function:(** <type parameter list> **)** <name><argument list><return type> ;

<method declaration> ::= <method header>
<declaration part>
<compound statement>

<method header> ::= **method** <method name><argument list><return type> ;
| **method:(**<type parameter list> **)** <method name><argument list>
<return type> ;

<name> ::= <identifier> | <method name>
<method name> ::= <class name>**.**<identifier>
<argument list> ::= **()** | **(**<argument declarations>**)**

<argument declarations> ::= <argument declaration>
| <argument declarations>**;** <argument declaration>

<argument declaration> ::= <mode><id list> **:** <named type>

The non-terminal <method name> makes use of the membership (dot) operator to connect identifiers so that class members may be referred to. The value of some variable `x.y` is `x`'s member `y` as defined by the class of the object which is currently the dynamic value of `x`. (In the case that `x` has no value, i.e., is undefined, an error occurs.) Thus all shared members of a class are *virtual* in the C++ sense.

The form of a subprogram mirrors the overall structure of a Leda program, except for the header which declares the name of the subprogram, the formal parameters, and return type. If the variable consists only of a single identifier, it must be a name which was not declared within the type or variable declarations of the immediate scope. The name will be implicitly defined as a variable of type function with the appropriate parameters and return type. Method declarations for variables consisting only of a single identifier are not allowed, since methods only make sense when associated with a class member. Instance members may not be given subprogram definitions in this section since they can only be accessed via variables which have undefined values at this point in the program.

```
type
  Foo := class
    i : integer;
    f : function(integer)->boolean;
  shared
    m : method(real)->real;
  end;

var
  x : Foo;
  g : function(Foo)->boolean;

// subprogram declarations:

  method Foo.m(r : real)->real;      // a valid method definition
  var
    // ... var declarations
  begin
    // ... code for the method
  end;

  function h(y : Foo)->integer;      // a valid function definition
  begin                              // h is implicitly declared as a variable
    // ... code for the function     //   of type function(Foo)->integer
  end;
  function x.i(...       // ILLEGAL: x.i is an instance member

  method x.m(...         // ILLEGAL: must use class name Foo to
                         //          refer to shared member in this context

  function g(...         // ILLEGAL: g already explicitly declared
```

4.5.1 The Main Function

Leda allows for a function to be declared as `main`, as well as allowing the main function to simply be a compound statement with no function header. The function `main` takes two arguments, the number of arguments on the command line prompt, and a pointer to an array of characters that contains the command line prompt. This function also must return an integer type.

```
function main(ac:integer, av:^^character)->integer;
  :          // declarations
begin
  :          // function code here
  return 0;
end;
```

4.6 External C Function Declarations

<cfunction declaration> ::= **cfunction** <id>**()** <return type> ;

Leda can incorporate functions defined in the C language. These functions are declared as above. Parameter type-checking is not strict, and hence need not be specified, although the return type is used for type-checking in Leda programs. As Leda argument types may have no corresponding analog in the C world, the following type conversions implicitly occur between Leda arguments and what is passed to the C function:

Leda **argument:**	**in C becomes:**
integer	long
real	double
^ integer	long *
^ real	double *
character	int
string	char *
boolean	int

Similarly, there is some disparity in terms of what type C functions may return and how they might be declared in Leda:

C return type:	**in Leda can become:**
long	integer
int	character
double	real
int	boolean
void	nothing
any type	nothing

In the above table, "nothing" denotes the fact that no return value needs to be declared for C functions returning results of the type in question. If the return value is not going to be used, none needs to be declared.

Some examples of declaring C functions for use in Leda programs:

```
cfunction printf();
cfunction strlen()->integer;
cfunction atol(^character)->integer;
```

Since the C function parameter types are ignored, the last function declaration above is equivalent to:

```
cfunction atol()->integer;
```

5 Expressions

```
<expression> ::= <unary expression>
               | <expression><binary operator><unary expression>

<unary expression> ::= <simple expression> | <unary operator><unary expression>

<simple expression> ::= <integer constant>
                      | <real constant>
                      | <character constant>
                      | <string constant>
                      | <lvalue>
                      | <function expression>
                      | <method invocation>
                      | <function invocation>
                      | <function expression><parameters>
                      | defined <expression>
                      | [<statement list> ]
                      | NIL

<binary operator> ::= + | - | * | / | > | < | >= | <= | = | <> | << | >> | % | & | | | <- |
                      == | ~=

<unary operator> ::= + | - | ~ | & | ^
```

```
<lvalue> ::= <id>
           | <expression>.<id>
           | <expression>-><id>
           | <expression>^.<id>
           | <expression>[<subscript expression>]
           | <expression>^

<function expression> ::= <function header2>
                          <declaration part>
                          <compound statement>
<function header2> ::= function<argument list><return type>;
                     | function:(<type parameter list>) <argument list><return type>;
```

Expressions in Leda, like in most other languages, have a value. In the case of integer and real constants, the value is self-evident. The value of a variable is the object to which it refers. When a variable refers to the member of some class, it must be a shared member accessed via the class object itself. It is an error to change a shared member using some name defined by a variable declaration to be of that class. This restriction is made to make clear that the shared member definition will apply to all instances of the class. Expressions of the form *expr1* `<-` *expr2* and the form `[`<statement list>`]` return the boolean value `true`. A function expression can be thought of as a constant of type `function`, as it too exhibits its own value. A function expression may be followed by actual parameters, in which case the nameless function is invoked and the value of the expression is the value returned by the function. Invocations, which include constructors, are examined in the section on statements. Finally, the special built-in `defined` predicate takes an expression of any type, and returns the boolean value `true` when the result of the expression (usually a variable) is defined, and `false` when the result of the expression is undefined.

5.1 Operators

Operators in Leda are actually an alternative syntax for invoking member subprograms. Many Leda operators have a corresponding method name. The table below shows the method name that goes with each operator. In the case of binary operators, the left-hand expression acts as the receiver. The name corresponding to the operator is looked up in the class of the left hand expression. It is an error if the method is not found, or does not take one parameter which can legally accept the value of the right-hand expression as the actual argument. Otherwise, the method is invoked as any invocation, and the value of the expression is the value returned by the subprogram.

Unary operators have one expression which acts as the receiver. A method with a name corresponding to the operator should exist within the class of the result of the expression, and should take no parameters.

Binary operators:	
+	plus
-	minus
*	times
/	slash
%	mod
=	equal
<>	notEqual
>	greater
<	less
>=	greaterEqual
<=	lessEqual
<<	leftShift
>>	rightShift
&	and
|	or
<-	
Unary operators:	
+	unaryPlus
-	unaryMinus
~	not
Pointer operators:	
&	
^	
Object operators:	
==	
~=	
Field operators:	
.	
->	

Some of the operators in the table do not have corresponding method names. The operators == and ~= are address-based object comparisons, and test whether two objects have the same or dissimilar addresses. The method names may be used in place of the operators:

```
var
  i, j : integer;
  b1, b2, b3 : boolean;
begin
  i := j + 5;  // equivalent to i := j.plus(5);
  i := -j;     // equivalent to i := j.minus();

  b1 := b2 & (b1 | ~b2);  // equivalent to b2.and(b1.or(b2.not()))
```

Programmers can use these semantics to overload operators for use with their own classes or change the semantics. By using the names from the above table for methods, the infix notation may be used in place of the method invocation syntax:

```
type
  Point := class
    x : real;
    y : real;
  shared
    plus : method(Point)->Point;
  end;

var
  p1, p2, p3 : Point;

  method Point.plus(p : Point)->Point;
  begin
    return Point(x + p.x, y + p.y);
  end;

begin
  p1 := Point(1,2);
  p2 := Point(4,5);
  p3 := p1 + p2; // equivalent to p3 := p1.plus(p2);
end;
```

5.1.1 Operator Precedence

With the exception of the bit-wise operations, operator precedence in `Leda` is the same as operator precedence in C.

```
    i := 3 + 4 * 5;     // i gets the value 23 (not 35)
```

The hierarchy of operator precedence is as follows, from lowest to highest precedence:

Operators:	**Associativity:**
`|`	left
`&`	left
`~`	right
`<-`	left
`=  <>  ==  ~=`	left
`<  >  >=  <=`	left
`<<  >>`	left
`+  -`	left
`*  /  %`	left
unary operators	right
`^  .  ->  (  [`	left

5.2 Filter

Sometimes it is necessary to convert the type of a variable to a more specific type—a subclass. Since Leda does not provide any casting mechanisms, the method **filter**, which is understood by all class objects, can effectively perform this conversion. The message **filter**, sent to a class object with an expression as an argument, returns an object of the type represented by the receiver. The object returned is the argument itself, if its dynamic type is a subclass or the same class as the receiver type, otherwise it is the value NIL.

```
cfunction printf();

type
  foo:=class
    i : integer;
  shared
    print : method();
  end;

  bar:=class of foo
  end;

  cat:=class of bar
  end;

var
  f : foo;
  b : bar;

  method foo.print();
  begin
    printf ("%d \n",i);  // print value of i
  end;


begin
  f:=foo(1);
  b:=bar.filter(f);
  if defined(b) then
    b.print();         // nothing happens, b is undefined

  f:=bar(2);
  b:=filter(bar, f); // alternate calling style, successful conversion
  b.print();         // prints '2'

  f:=cat(3);
  b:=bar.filter(f);  // successful conversion
  b.print();         // prints '3'
end;
```

6 Statements

```
<statement> ::= <compound statement>
              | <assignment statement>
              | <expression>
              | <conditional statement>
              | <return statement>
              | <for statement>
              | <while statement>
              | <repeat statement>
```

6.1 Assignments

```
<assignment statement> ::= <id> := <expression>
                         | <id> := NIL
```

The first form of the assignment statement causes the left-hand identifier to refer to the result of evaluating the right-hand expression. The second form uses the keyword `NIL` to cause the identifier to become undefined, to refer to no value. It is important to note that Leda uses *pointer semantics*. This means that when one variable is assigned to another, following the assignment both variables refer to the exact same object. A change in some instance member of either variable would result in a change to both variables.

When assigning values to shared members of a class, the member must be accessed via the class name, not an instance of the class.

The result of the right-hand expression must be of the same or some descendant class of the left-hand identifier, or be coercible to such a class as explained below. In the second form of the assignment the left-hand identifier may be of any class.

```
type
  Foo := class
    i : integer;
  shared
    j : integer;
  end;

var
  x, y : Foo;

begin
  x := Foo.new();  // two distinct objects x,y of class Foo are created
  y := new(Foo);   // alternative style of an invocation - see below

  x.i := 1;  // objects 1 and 2 assigned respectively to
  y.i := 2;  //   the distinct members i of x and y

  x := y;    // x and y now refer to the exact same object
  x.i := 99; // x.i and y.i both now have the value 99

  Foo.j := 7;  // legal assignment to shared member j

  x.j := 11;   // ILLEGAL: must use class name when assigning to
               //   shared members
end;
```

6.1.1 Coercion

Leda has a fairly passive type system, which is to say that Leda has a low tendency to coerce types in an attempt to pacify the type checker. There are no user-definable coercions possible. Integers will be coerced to reals (but not vice-versa) in the obvious way. In addition, function and method types may be coerced to each other. A function is coerced into a method by removing the first formal parameter and considering it to be the receiver class. When a method is assigned a function, the class in which the method is defined must match the class of the first formal parameter of the function, which must have the default mode of pass by value. References to the first formal parameter become references to the receiver. Conversely, a method is coerced into a function by inserting a new first formal parameter of the method's receiver class. References to the receiver become references to the first formal parameter instead.

```
type
  Foo := class
    r : real;
  shared
    m : method(real)->integer;
  end;

var
  x : Foo;

  function f(tmp : Foo, y : real)->integer;
  begin
    ... // code for function f
  end;

  function g(z : real)->integer;
  begin
```

```
    ... // code for function g
  end;

begin
  x := Foo(2.0);  // Foo created, with r = 2.0
  x.r := 3;       // value of r changed, 3 coerced to 3.0

  x.m := f;       // f coerced to method(real)->integer
                  //   f has receiver class (first argument) Foo, so OK
  f := x.m;       // reverse situation, also OK

  x.m := g;       // ILLEGAL: g coerces to method()->integer
                  //   with receiver class (first argument) real, so no good
  g := x.m;       // ILLEGAL: x.m coerces to function(Foo, real)->integer
end;
```

6.2 Invoking Subroutines

<invocation> ::= <expression><parameters>
<parameters> ::= **()** | **(**<parameter list>**)** | **:(**<type parameter list>**)(**<parameter list>**)**

<parameter list> := <parameter>
                  | <parameter list> , <parameter>

<parameter> ::= <expression>

The expression in the invocation must refer to or evaluate to either a function, a method or a class. If the expression is a function or method, the parameter expressions are evaluated from left to right, and passed according to the mode specified in its declared type. When `NIL` is used for an actual parameter, the corresponding formal parameter will be undefined, referring to no value. Any reference to class members within the code section of the method will access the corresponding members of the receiver. The receiver may alternately be inserted as the first actual parameter. The entire receiver must be used. Care must be taken that some declaration for the name of the member method does not also exist for a non-member variable, otherwise the form of method invocation in which the receiver is the first parameter of the argument list will not be possible. Control is passed to the function or method. The program will continue at the point following the invocation upon termination of the subprogram. Each actual parameter must be the same as, or some descendent class of, the declared type of the formal parameter. A `NIL` parameter matches any type.

```
  f(x,y);         // invoke the function f with parameters x and y
  x.a.m(i,j);     // assuming m is a method defined within the class of x.a,
                  //   invoke the method x.a.m with receiver x.a with args i and j
  m(x.a, i, j);   // alternative call syntax equivalent to above, assuming
                  //   m is not otherwise defined
  a.m(x, i, j);   // ILLEGAL: entire receiver must be used as first parameter
                  //   for alternative call syntax
```

6.2.1 Constructors

If the expression in the invocation is a class name then a *constructor* is being invoked. This is a function that creates and returns a new instance of the class. The constructor must be provided with a parameter list containing one parameter for each instance member of the class, including inherited instance members. The new object will be initialized by assigning from left to right, each actual parameter to the instance members in the order declared in the class definitions, from the top of the hierarchy downward. (It makes little sense to use a constructor as a statement, since the returned object is not used, so the following example shows the constructor as an expression; the syntax is the same.)

```
type
  Foo := class
    i : integer;
    r : real;
  shared
    m : method();
  end;

  Bar := class of Foo
    v : boolean;
  end;

  List := class
    first : integer;
    rest : List;
  end;

var
  f, g : Foo;
  b : Bar;
  l : List;

begin
  f := Foo(1, 3.14);         // creates a Foo and initializes i and r
  g := Foo();                // creates a Foo, i and r are uninitialized
  b := Bar(1, 3.14, true);  // creates a Bar and initializes i, r, and b
  l := List(100, NIL);       // inits first to 100 and leaves rest undefined
end;
```

The default constructor can be overridden to provide additional instance creation-time semantics. The method **new** is used to refer to the definition of user-specified constructors:

```
type
  Int := class
```

```
    value : integer;
  shared
    new : method (integer);       // constructor
  end;

// definition of user-specified (non-default) constructor

method Int.new (i : integer);
begin
  value := i;
  "created instance of class Int".print();
end;

var x, y : Int;

begin
 x := Int(1);                     // overridden constructor invoked
  :
  :
end;
```

Leda employs a mark-and-sweep storage reclamation scheme for garbage collection, and does not provide for any explicit destructors to specify deallocation-time semantics for allocated memory:

```
begin
 x := Int(1);
  :
  :
 x := NIL;   // memory referenced by x will be reclaimed
  :
  :
end;
```

6.2.2 Lazy-Evaluated Functions

The keyword lazy delays an evaluation of a parameter until that parameter is actually used by called function. It can be thought of as a shorthand notation for passing a closure as a parameter. For example, in the following program:

```
  type Complex := class
      re, im : real;
    shared
      times : method(lazy Complex)->Complex;
    end;
```

```
method Complex.times(x : lazy Complex)->Complex;
var y : Complex;
begin
  if re = 0 & im = 0 then
    return self;
  y := x;
  return Complex(re * y.re - im * y.im,
                 re * y.im + im * y.re);
end;

var a, b, c : Complex;

begin
  a := Complex(0,0);  b := Complex(2,2);
  c := a * (b * b);
end;
```

the evaluation of the argument to `Complex.times` is delayed until `self` turns out to be non-zero. In this example the expression `(b*b)` is not evaluated because `a` is zero. This program is equivalent to the following program:

```
type Complex := class
    re, im : real;
  shared
    times : method(function()->Complex)->Complex;
  end;

method Complex.times(x : function()->Complex)->Complex;
var y : Complex;
begin
  if re = 0 & im = 0 then
    return self;
  y := x();
  return Complex(re * y.re - im * y.im,
                 re * y.im + im * y.re);
end;

var a, b, c : Complex;

begin
  a := Complex(0,0);  b := Complex(2,2);
  c := a * function()->Complex; begin
            return b * function()->Complex; begin
                          return b;
                        end;
            end;
  end;
```

The keyword `lazy` saves in terms of program length. Also, it provides good abstraction. However, because it is just a shorthand notation for passing closures, the closure, which is implicitly given, is evaluated every time the corresponding argument is referenced. In the above example, we avoid recomputation by using the temporary variable `y`. This attribute limits the usability of lazy evaluation, so it is likely that the semantics of the keyword `lazy` will be changed to avoid recomputation in the near future.

6.3 Conditionals

<conditional statement> ::= **if** <expression> **then** <statement> **else** <statement>
| **if** <expression> **then** <statement>

Both the single and two-way conditionals are provided. The expression must evaluate to an object of the predefined `boolean` class or some alias. This class is an enumerated type with enumerated constant values `true` and `false`.

```
  if i = 7 then
    k := 0
  else
    k := 1;
```

6.4 Iteration

<for statement> ::= **for** <variable> **:=** <expression> **to** <expression> **do**
<statement>
| **for** <variable> **:=** <expression> **downto** <expression> **do**
<statement>

<while statement> ::= **while** <expression> **do** <statement>
<repeat statement> ::= **repeat** <statement> **until** <expression>

The loop variable in the `for` loop can be a real, an integer, a character, an enumerated type (or some alias of one of these types), or any user-defined class with the operations `plus` (for integers), `greaterEqual` and `lessEqual` defined. Likewise, the expressions must evaluate to values of the iteration variable type. The `for` loop assigns the first expression to the loop variable. It then evaluates the second expression, and compares the loop variable to the result. If the value of the loop variable is less than or equal to the result of the second expression, the loop statement is executed, and the loop variable's successor or predecessor (in the case of the reverse loop) is computed and assigned to the loop variable. The second expression is then re-evaluated, the variable compared, and so on, until the variable is greater than the expression and the program continues with the statement following the `for` loop. The loop variable may also be manipulated within the body of the loop:

```
var
  i, j : integer;
begin
  j := 7;
  for i := 1 to j do
    i.print();   // prints 1 through 7

  j := 3;
  for i := 1 to j do
    begin
      i.print();  // prints 1, 3, 5, 7, and 9
      i := i + 1;
      j := 10;
    end;
end;
```

The expression in the `while` and `repeat` loops must evaluate to an object of the predefined class `boolean` or some alias. The `while` loop evaluates the expresssion first and then keeps executing the statement until the expression is `false`. The `repeat` loop always executes its statement once, and then tests the expression. The statement will execute repeatedly until the expression is `true`.

6.5 Returning Values

<return statement> ::= **return** | **return** <expression>

The `return` statement causes termination of the subprogram (or the main program) in which it resides. If an expression is used, the type of the expression must be the the same as, or some descendent class of, the declared return type. The value of the expression becomes the value of the invocation expression or statement which caused the subprogram to be executed.

7 Relational Programming

Relational programming in Leda consists of using the binary operators `&` and `|`, along with a new operator `<-`, which allows backtracking assignment. The operators `&` and `|` allow for the formulation of boolean functions corresponding to Horn clauses.

7.1 Reversible Assignment

<reversible assignment> ::= <id> `<-` <expression>

The reversible assignment (`<-`), coupled with by-reference parameter passing, allows for relational programming in Leda. The reversible assignment operator, like the usual assignment operator `:=`, assigns the value of its right hand side to it left hand side. The reversible assignment stores the old value of its left hand side (on the trail stack) to facilitate backtracking upon failure, however. When backtracking occurs, the current assignment is cancelled and the old value restored. Assignments performed via := do not allow for this feature.

7.2 Facts and Relations

Relational programming in Leda is provided by means of the reversible assignment (<-) statement in conjunction with var parameters. The var parameters allow information to flow in and out of subprograms via unification and backtracking. The information has to be specified in the subprogram definition using var parameters. These parameters are at the core of the logical component in Leda since they are not specifically defined as input or output parameters and can be used in both manners. They generally act like pass-by-reference parameters except when the value is altered via the reversible assignment.

Facts can be created by defining subprograms that contain clauses joined by the boolean connectives & and |. These facts also require some means of performing unification, here provided via the subprogram eq, discussed later. To create a set of clauses representing the genealogy of figures from Greek mythology, one would create a subprogram child:

```
type
  names:=(helen, leda, zeus, hermione, menelaus);

  function child(var name, mother, father : names)->boolean;
  begin
  return eq(name, helen)      & eq(mother, leda)      & eq(father, zeus)     |
         eq(name, hermione)   & eq(mother, helen)     & eq(father, menelaus) |
    // ... more facts
  end;
```

Each line of the subprogram child can be viewed as a clause, asserting a fact. The first line, for example, asserts that Helen is the child of Leda and Zeus.

Inference rules describe how new information can be derived from existing relations.

```
  function mother(var mom, kid : names)->boolean;
  var
    dad: names;
  begin
    return child(kid, mom, dad);
  end;

  function daughter(var lass, parent : names)->boolean;
  begin
    return female(lass) & mother(parent, lass) |
           female(lass) & father(parent, lass);
  end;
```

7.3 Unification and Backtracking

The predicate eq implements unification to the degree required in the example being discussed here, namely, this version of eq does not support unification of two undefined variables. The predicate eq can be written as follows:

```
function eq(var x, y : names)->boolean;
  begin
    if defined(x) then
      if defined(y) then
        return x = y
      else
        return y <- x
    else
      if defined(y) then
        return x <- y
      else
        return false;
  end;
```

The predicate eq takes two reference parameters, x and y and returns a boolean. If both x and y hold some values, eq compares those values. If one of them is undefined, it gets the value of the other. Otherwise, both parameters are undefined. As this case never occurs in the above example, eq simply returns false.

The predicate eq acts like a bidirectional assignment, thus implementing unification. For assignment, eq uses the reversible assignment operator (<-) instead of the simple assignment operator (:=). This allows for the unbinding of values during backtracking.

Here is the remainder of the program example, illustrating some sample uses of the functions defined above:

```
var
  p1, p2 : names;
  res : boolean;

begin
  p1 := NIL;          // set p1 to 'undefined'
  p2 := helen;        // set p2 to 'helen'
  mother(p1, p2);     // 'leda' gets assigned to p1
  p1.print();         // prints 'leda'

  p1 := helen;
  p2 := NIL;
  mother(p1, p2);     // 'hermione' gets assigned to p2
  p2.print();         // prints 'hermione'
  res:=mother(leda, hermione); // arguments can be expressions
  res.print();                 // prints 'false'
end;
```

From within imperative code a relation is invoked as if it were a boolean function. The returned value may be used or not. Unification causes information to flow in and out across the `var` parameters. Since the relational paradigm is not manifested by its own procedural abstraction but rather embedded in functions and methods, arbitrary imperative style code can appear between the clauses. A call on a relation, with some arguments potentially bound to values, is known as a *query*. If a query has multiple solutions one response will be returned.

7.4 Iterating over Queries

<for statement> ::= **for** <invocation> **do** <statement>

The `for` statement allows <statement> to iterate over all solutions of a query, which is specified as <invocation>. The relation is invoked with the specified arguments and its most recent choice point (pointing to its next alternative) is saved. Then <statement> is executed, and after restoring the saved choice point, backtracking is invoked by failure. The next alternative will then be tried. This scheme will be repeated until the relation finally returns `false` and control is transferred to the statement following the `for` statement.

```
  for brother(p1, p2) do   // p1 and p2 are undefined
    begin                  // prints all pairs of brothers
      p1.print();
      p2.print();
    end;

  p2 := helen;
  for brother(p1, p2) do print(p1);  // prints all brothers of helen
```

The above example shows how the relation `brother` might be used with `for` statements. Note that in order to force the exploration of all pairs of brothers, the actual parameters have to be unbound (i.e. their value is undefined). It is only then that they subsequently get bound to all possible solutions.

8 Constraint Logic Programming

Constraint solvers can be implemented in Leda using the object-oriented features of Leda and the reversible assignment operator (`<-`). There are no built-in constraint solvers in Leda, and even the simple unification function `eq`, discussed earlier in the section on relational programming, must be supplied by the user. The paper by Takikawa[Tak92] gives several examples of constraint solvers written in Leda.

9 Predefined Classes

Certain predefined classes are provided by the Leda compiler. These are `integer`, `real`, `character`, `string`, `array`, `pointer`, the abstract class `enum`, and one of its subclasses `boolean`. These classes make use of non-object members with *primitive* types that cannot be manipulated by the Leda programmer. The supplied methods for these classes are written in a lower level language, and have to be linked with Leda programs as part of the compilation process. They can not be overridden in a Leda program. The methods available for the `integer` and `real` classes are: `plus`, `minus`, `times`, `slash`, `print`, `less`, `lessEqual`, `greater`, `greaterEqual`, `equal`, `notEqual`, `unaryPlus`, `unaryMinus`. In the class `integer` there is also a method `mod` available. All methods take a single parameter of the same class as the receiver, except for `print`, `unaryPlus`, `unaryMinus`, that take no arguments. The arithmetic methods return still the same class while the relational methods return a `boolean` object. Recall that infix notation may be used to invoke these methods by choosing the appropriate operator from the operator/name chart above. The predefined classes `array` and `pointer` have no methods that can be used directly by the user.

All enumerated types defined within a Leda program are defined to be subclasses of the predefined abstract superclass `enum`. From this class they inherit the methods: `print`, `succ`, `pred`, `plus`, `minus`, `less`, `lessEqual`, `greater`, `greaterEqual`, `equal`, `notEqual`. The methods `succ` and `pred` take no arguments and respectively return the successor and predecessor of the receiver. The successor of the last item is the first who's predecessor is again the last. The `boolean` class adds the special[2] methods `and`, `or`, `not`, which all return values of type `boolean`. Programmers may also use the `boolean` constants `false` and `true`. Unlike many languages, when the `print` method is received by an object of an enumerated type, the actual text of the enumerated constant is printed out as a string.

10 Scoping

Leda uses static scoping. Subprograms retain their environment of definition, even when returned from, or passed to, other subroutines. Leda functions compile to C language functions with heap-based allocation as opposed to stack-based allocation, allowing closures to be constructed.

11 Libraries

Leda allows for the notion of libraries via the include statement. Examples include the Leda files `stdio.led` and `string.led`, included in the Leda distribution. The `stdio.led` file includes definitions for file operations while `string.led` defines operations on strings.

```
include "stdio.led";            // include the stdio library
include "string.led";           // include the string library
```

[2]these methods are *special* in the sense that in the expression `if a and b(x)....` the function `b(x)` is passed unevaluated to the method `and`

11.1 The `stdio.led` Library

`Stdio.led` provides the class `file`, and associated methods `open`, `close`, `write`, `readChar`, `readInt`, and `readReal`. Also provided is a routine `initStdio` to initialize the C language based `stdin`, `stdout`, and `stderr`.

The `leftShift` and `rightShift` methods are overridden by class `file` to allow the C++ style of input and output:

```
include "stdio.led";

var
  i : integer;

begin
  initStdio();
  stdout << " Enter an integer ";
  stdin >> &i;
  stdout << " The number entered was " << i << '\n';
end;
```

11.2 The `string.led` Library

Class library `string.led` provides a class definition for `String`, a transfer function `str` to convert from type `string` to an instance of class `String`, and various methods for creating instances of `String` and determining equality among instances of class `String`.

12 Compiler Usage

Leda is invoked with the command `leda <flags> <filename>`. There are various compiler flags that can be used in conjunction with the compiler. They are:

-debugging
-separate
-w#
-w

The -debugging option is not for end-user purposes. The -separate option is for planned use when separate compilation is implemented. The -w# option takes a single digit integer (i.e. # is a number between 0 and 9) to indicate the level of warnings for the compiler to return. Level 0 suppresses all warning messages. Level 1 is the default, while level 2 is very verbose. Levels 3-9 are the same as level 2. The -w option, with no level specified, uses the default of level 1.

A Reserved Words

NIL	**downto**	**lazy**	**to**
array	**else**	**method**	**true**
begin	**end**	**module**	**type**
cfunction	**false**	**of**	**until**
class	**for**	**repeat**	**var**
const	**function**	**return**	**while**
defined	**if**	**shared**	
do	**include**	**then**	

B Compile-Time Error Messages

<string> too long The length of symbol, string, or number exceeds the length of internal buffer (256 characters).

Parent class is undefined An undefined class is used as the parent (or superclass) of a class definition.

Parent must be a class The parent (or superclass) of a class definition must be a user-defined class (not, say, integer).

boolean expression is expected The condition of "if" or "while" statement must be a boolean expression.

illegal character A control code that is not defined in Leda is used.

illegal kanji code An input code breaks the rules of Kanji code format.

lvalue is expected Use of a left-value such as variable, array reference, or pointer dereference is required.

no closing double quote A string has been left unterminated. There is no corresponding double quote ending the string.

no closing single quote A character has been left unterminated. There is no corresponding single quote ending the character.

no return value Function that has a declared return value must return some such value.

nonsense subscript The lower bound of an array exceeds the upper bound.

numeric expression is expected Arithmetic operators need numeric expressions as their arguments.

overriding type mismatch The overriding type constraints are not satisfied i.e a child class overrides a field of the parent and types of the two fields are different.

period used with non object Period (field selection expression) is used with a non-structured object (such as a pointer).

period used with undefined class Period (field selection expression) is used with an object of an undefined class.

pointer is expected Pointer dereference expression requires a pointer as its argument.

procedure cannot return a value Function that does not have a declared return value returns a value.

subscripts on non array Subscripts are used with a non-array object.

term does not evaluate to a function A function-call expression needs a function as its argument.

too few arguments The number of actual parameters is less than the number of formal parameters.

too many arguments The number of actual parameters is more than the number of formal parameters.

type mismatch Types must meet when assigning a value to a variable, or when passing a value as an argument of a function.

type parameter is used for a shared member Cannot use type-parameter in shared portion of a class definition, except in method definitions.

type parameters are needed When defining a variable of a type-parameterized class, actual type parameters for the class must be provided.

type parameters for non parameterized class Actual type parameters are used with non type-parameterized class.

type parameters for non parameterized function Actual type parameters are used with non type-parameterized function.

type parameters mismatch The number of actual and formal type-parameters are different.

type redefinition The same class definition appears twice.

type restriction is broken Actual type parameters must satisfy type-parameter constraints.

undefined class is used as constructor Undefined class is used in a constructor expression.

undefined field name Field name for a class is not found in the class definition.

undefined variable Undefined variable is used.

unexpected end of file The compiler exhausts the input while processing a structure such as a compound statement or a class definition.

unknown operator on enum The operator used is not defined on enumerated types.

unknown operator on pointer The operator used is not defined on pointers.

variable redefinition The same variable definition appears twice.

C Run Time Error Messages

GC stack overflow The stack for garbage collection overflows.

cannot access special field of special object

cannot change value of field of special object

cannot take address of field of special object

Special field: myClass, parent; Special object: integer, real, pointer, etc; Field of special object: integer value, real value, etc.

chkEnv `Leda` has not been correctly compiled on the system.

illegal pointer addition Only pointers that actually point to an array member are allowed in pointer-addition expressions.

illegal pointer comparison Only pointers that actually point into the same array are allowed in pointer-comparison expression.

insufficient memory Cannot allocate initial memory.

out of memory Heap is exhausted.

subscript out of range A pointer that does not point within the boundaries of an array is dereferenced.

trail stack overflow Trail stack is exhausted.

undefined value is used `NIL` is used in an arithmetic or sub-field expression.

D Bibliography

References

[Bud89a] Budd, T. A., "Low Cost First Class Functions", Oregon State University, Technical Report 89-60-12, June 1989.

[Bud89b] Budd, T. A., "Data Structures in LEDA", Oregon State University, Technical Report 89-60-17, August 1989.

[Bud89c] Budd, T. A., "The Multi-Paradigm Programming Language LEDA", Oregon State University, Working Document, September 1989.

[Bud91a] Budd, T. A., "Blending Imperative and Relational Programming", *IEEE Software*, pp. 58–65, January 1991.

[Bud91b] Budd, T. A., "Sharing and First Class Functions in Object-Oriented Languages", Working Document, Oregon State University, March 1991.

[Bud91c] Budd, T. A., "Avoiding Backtracking by Capturing the Future", Working Document, Oregon State University, March 1991.

[Bud92] Budd, T. A.,"Multiparadigm Data Structures in Leda", *Proceedings of the 1992 International Conference on Computer Languages*, Oakland, CA, p. 165–173, April 1992.

[Che91] Cherian, V., "Implementation Of First Class Functions And Type Checking For A Multi-Paradigm Language", *Master's Project*, Oregon State University, May 1991.

[Pes91] Pesch, W., "Implementing Logic In Leda", Oregon State University, Technical Report 91-60-10, September 1991.

[PeS91] Pesch, W., and Shur, J., "A Leda Language Definition," Oregon State University, Technical Report 91-60-09, September 1991.

[Shu91] Shur, J., "Implementing Leda: Objects And Classes", Oregon State University, Technical Report 91-60-11, August 1991.

[Tak92] Takikawa, M., "CLEDA–LEDA with Constraint Logic Programming," *Master's Project*, Oregon State University, October 1992.