Appendix C

The Standard Library

In this appendix we describe the functions found in the standard Leda run-time library. All of the functionality provided by this library is written in Leda itself, and is thus available for modification by the programmer (although care must be exercised when manipulating some of the more basic classes, such as `object` or `Class`).

C.1 The class `object`

The class `object` is the root class of all Leda entities, including functions. It defines two data fields, one of type `Class`, and the second of type `object`. While used internally as the static link, the second field is currently not used by any user-accessible functions.

Three operations are implemented in class `object`. The first is the function `asString`, used to convert a value into a string. This is used, for example, by the `print` function (Section C.10). By default objects print simply by printing their class name. This function is overridden in a number of classes, such as `integer`, `string`, and `real`.

The second and third functions define the meaning for the operators `==` and `~=`. The `==` operator is implemented so as to test object identity between the two argument values; that is, the value returned is true if the two arguments are exactly the same entity. This operation is occasionally overridden in subclasses (for example, it is overridden by integers). Testing for same identity should not be confused with testing for equality (next section). The `~=` operator is defined to be simply the inverse of the `==` operation. Thus, `~=` will seldom need to be redefined even if the operation `==` is overridden.

```
class object;
var
    classPtr : Class;      { pointer describing object }
    context : object;      { pointer to context - not used }
```

```
    function asString ()->string;
    begin         { return name of our class }
        return cfunction
            Leda_object_at(classPtr, 2)->string;
    end;

    function sameAs (arg : object)->boolean;
    begin         { true if self and arg are same object }
        return cfunction Leda_object_equals(self, arg)->boolean;
    end;

    function notSameAs (arg : object)->boolean;
    begin         { inverse of sameAs }
        if self == arg then
            return false;
        return true;
    end;
end;
```

C.2 The classes `equality` and `ordered`

The classes `equality` and `ordered` define protocol for objects which know how to compare against similar objects for equality, and comparison using relational operators. Both classes are parameterized by a type value which is the type associated with the right argument in boolean operators. When subclassing from these classes the class type for the subclass is normally used as the type argument as well, so that both left and right arguments in boolean operations have the same type.

The class `equality` simply provides the meaning for the two operators `=` and `<>`. The former is normally redefined in subclasses, although in certain situations the default meaning, which is identity of the two arguments, is sufficient. The latter operation is defined in terms of the former, so it will seldom need to be redefined in subclasses.

```
class equality [T : equality];

    function equals (arg : T)->boolean;
    begin
            {  default is simply pointer equality }
        return self == arg;
    end;
```

```
    function notEquals (arg : T)->boolean;
    begin
            { use = which can be redefined }
        if self = arg then
            return false;
        return true;
    end;
end;
```

The class `ordered` provides meaning for the remaining four relational operators (<, <=, >=, and >). These are defined in terms of each other, so a subclass need only redefine the meaning of the less-than operator to gain access to the remaining three. An additional function `between` is defined which can be used to determine if a value is between two limits.

```
class ordered [T : ordered] of equality[T];

    function less (arg : T)->boolean;
    begin
        return false;
    end;

    function lessEqual (arg : T)->boolean;
    begin
        return self < arg | self = arg;
    end;

    function greater (arg : T)->boolean;
    begin
        return ~ (self <= arg);
    end;

    function greaterEqual (arg : T)->boolean;
    begin
        return self > arg | self = arg;
    end;

    function between (low, high : T)->boolean;
    begin
        return low <= self & self <= high;
    end;
end;
```

C.3 The class `Class` and the `typeTest` function

The class `Class` (the initial upper case letter being used to avoid conflict with the keyword) is used to define protocol for the object which is the representative of each class. Classes maintain data fields containing the class name, size, and parent class.

Class `Class` is a subclass of `ordered`, the relational operators being used to describe the class/subclass relationship. That is, the value of $x < y$ is true if x and y are class values and x is a (proper) subclass of y. The value of $x <= y$ is true if either x is the same class as y or $x < y$.

There are three functions implemented in this class. The first simply overrides the printing function inherited from class `object`, so that class values respond with their name. The second provides implementation for the relational less-than operator. Since the default meaning (namely, identity) of the equality testing operator is used, this then implicitly defines all six relational operators. The third function takes as argument a value, and returns true if the value is an instance of the class, or an instance of a subclass of the class.

```
class Class of ordered[Class];
var
    name : string;
    size : integer;
    parent : Class;

    function asString ()->string;
    begin
        return name;
    end;

    function less (arg : Class)->boolean;
    begin
        if self == arg then     { equal, not less }
            return false;
        if parent == arg then
            return true;
        return parent <> self & parent < arg;
    end;

    function isInstance (val : object)->boolean;
    begin
        return val.classPtr <= self;
```

```
    end;
end;
```

The latter operation is used in the implementation of the function `typeTest`, which can be used to not only determine if a value is or is not an instance of a specific class, but to convert the value into a form so that it can be assigned to an instance of a class. (Normally the class in question is a subclass of the declared type for a variable). The `typeTest` function either returns the undefined value `NIL`, which can be assigned to any variable, or it returns the first argument value. The former is used to indicate the value is not an instance of the class in question, while the latter indicates it is. The condition of the result can then be tested using the predicate `defined`.

```
function typeTest [T : object] (val : object, aClass : Class)->T;
begin
    if aClass.isInstance(val) then
        return cfunction Leda_object_cast(val)->T;
    return NIL;
end;
```

An example of the use of `typeTest` is found in the function which overrides the meaning of the `==` operator in the class `integer`. The argument is declared as simply being an instance of class `object`, whereas the function wishes to define an alternative meaning if the actual value is an instance of class `integer`. See section C.6.

C.4 The classes `boolean`, `True` and `False`

The class `boolean` defines the meaning of the unary negation operator `~` and the binary operators `|` and `&`. In the binary operators the argument is passed by-name, and thus evaluation of the right argument value will be delayed until it is actually used in the body of the function.

```
class boolean of equality[boolean];

    function not ()->boolean;
    begin
        if self then
            return false;
        return true;
    end;

    function or (byName arg : boolean)->boolean;
```

```
    begin
            { if either are true, return true }
        if self then
            return true;
        if arg then
            return true;
        return false;
    end;

    function and (byName arg : boolean)->boolean;
    begin
            { if both are true, return true }
        if self then
            if arg then
                return true;
        return false;
    end;
end;
```

In actual practice the implementations defined in class `boolean` will never be executed, as they are all overridden in the pair of classes `True` and `False`, which are used to create the global constants `true` and `false`, respectively. Each class redefines the conversion to string function (inherited from class `object`), and the operations defined in class `boolean`. The effect of these changes is to avoid the evaluation of the right-hand argument to binary operators when the result can be determined solely by the left argument. Such an evaluation scheme is sometimes referred to as *short circuit evaluation*.

```
class True of boolean;

    function asString ()->string;
    begin
        return "true";
    end;

    function not ()->boolean;
    begin
        return false;
    end;

    function or (byName arg : boolean)->boolean;
    begin
        return true;
```

```
    end;

    function and (byName arg : boolean)->boolean;
    begin
        return arg;
    end;
end;

class False of boolean;

    function asString ()->string;
    begin
        return "false";
    end;

    function not ()->boolean;
    begin
        return true;
    end;

    function or (byName arg : boolean)->boolean;
    begin
        return arg;
    end;

    function and (byName arg : boolean)->boolean;
    begin
        return false;
    end;
end;
```

C.5 The class `real`

The class `real` defines the protocol for real numbers. Reals are a subclass of class `ordered`, and thus can be used with relational operators. The inherited functions `asString`, `equals` and `less` are all overridden with appropriate definitions. Reals define the arithmetic operators $+$, $-$, $*$ and $/$, but not the remainder operator `%`. The remaining two functions implement unary negation and the conversion of real values to integer via truncation.

```
class real of ordered[real];
```

```
function asString ()->string;
begin
    return cfunction Leda_real_asString(self)->string;
end;

function equals (arg : real)->boolean;
begin
    return cfunction
        Leda_real_equals(self, arg)->boolean;
end;

function less (arg : real)->boolean;
begin
    return cfunction
        Leda_real_less(self, arg)->boolean;
end;

function plus (arg : real)->real;
begin
    return cfunction
        Leda_real_plus(self, arg)->real;
end;

function minus (arg : real)->real;
begin
    return cfunction
        Leda_real_minus(self, arg)->real;
end;

function times (arg : real)->real;
begin
    return cfunction
        Leda_real_times(self, arg)->real;
end;

function divide (arg : real)->real;
begin
    return cfunction
        Leda_real_divide(self, arg)->real;
end;
```

```
    function negation ()->real;
    begin
        return 0.0 - self;
    end;

    function asInteger()->integer;
    begin
        return cfunction
            Leda_real_asInteger(self)->integer;
    end;
end;
```

There are (almost) no automatic coercions or conversions performed by Leda.[1] Mixed mode arithmetic is implemented by overloaded versions of the arithmetic operators, defined in the global scope:

```
function plus (left : integer, right : real)->real;
begin
    return left.asReal() + right;
end;

function plus (left : real, right : integer)->real;
begin
    return left + right.asReal();
end;
```

C.6 The class integer

The class `integer` defines protocol for the manipulation of integer objects. The underlying computer architecture may place limitations on the size of integers that can be held in an instance of this class. (The creation of infinite precision integers is, however, an interesting and useful programming exercise. Since the standard library is written in Leda itself such values can even be easily integrated into the rest of the system).

The class `integer` inherits from class `ordered`, and redefines the relational operators, as well as the conversion to string function inherited from class `object`. The identity testing operator `==` is redefined in class `integer` so that two integer values with the same magnitude will test equal. Doing so requires testing the right argument, which is declared simply as an instance of class `object`.

[1]The one exception being the conversion of relation values into booleans in the expression portion of an `if` or `while` statement.

The binary operators `|` and `&` are defined so as to mean bit-wise boolean operations, as in the unary operator `~`, which produces a result by inverting all bits in the argument.

The arithmetic operators $+$, $-$, $*$, $/$ and `%` are given their expected meaning, the latter being defined as the remainder which remains after the left argument is divided by the right.

The final two operations implemented in the class perform unary negation and the conversion of integer values into real numbers.

```
class integer of ordered[integer];

    function asString ()->string;
    begin
        return cfunction Leda_integer_asString(self)->string;
    end;

    function sameAs (arg : object)->boolean;
    var
        argInt : integer;
    begin
            { if arg is an integer, use int compare }
        argInt := typeTest[integer](arg, integer);
        if defined(argInt) then
            return self = argInt;
            { not equal to anything but integer }
        return false;
    end;

    function equals (arg : integer)->boolean;
    begin
        return cfunction
            Leda_integer_equals(self, arg)->boolean;
    end;

    function less (arg : integer)->boolean;
    begin
        return cfunction
            Leda_integer_less(self, arg)->boolean;
    end;

    function or (arg : integer)->integer;
    begin
        return cfunction
            Leda_integer_or(self, arg)->integer;
```

```
end;

function and (arg : integer)->integer;
begin
    return cfunction
        Leda_integer_and(self, arg)->integer;
end;

function not ()->integer;
begin
    return cfunction Leda_integer_not(self)->integer;
end;

function plus (arg : integer)->integer;
begin
    return cfunction
        Leda_integer_plus(self, arg)->integer;
end;

function minus (arg : integer)->integer;
begin
    return cfunction
        Leda_integer_minus(self, arg)->integer;
end;

function times (arg : integer)->integer;
begin
    return cfunction
        Leda_integer_times(self, arg)->integer;
end;

function divide (arg : integer)->integer;
begin
    return cfunction
        Leda_integer_divide(self, arg)->integer;
end;

function remainder (arg : integer)->integer;
var
    result : integer;
begin
    result := self - (self / arg) * arg;
```

```
        if result < 0 then
            result := result + arg;
        return result;
    end;

    function asReal ()->real;
    begin
        return cfunction
            Leda_integer_asReal(self)->real;
    end;

    function negation ()->integer;
    begin
        return 0 - self;
    end;
end;
```

C.7 The class `string`

The class `string` implements operations used in the manipulation of text values. Strings are a subclass of ordered, and define the ordering operations as lexicographic (dictionary-style) sequencing. The plus operation is used to concatenate two string values together – the second argument is converted into a string using the function `asString` if it is not already a string. The function `length` is used to determine the number of characters in a string. The function `subString` is used to extract a subportion of a string; the two arguments represent the starting location and length of the subtext. If the first argument is negative it is interpreted as an index from the right side. The function `index` takes a second string as argument and returns the position in the receiver at which the argument string first appears, returning `NIL` if the argument does not appear in the string at all.

```
class string of ordered[string];

    function asString ()->string;
    begin
        return self;
    end;

    function equals (arg : string)->boolean;
    begin
        return 0 = cfunction
```

```
        Leda_string_compare(self, arg)->integer;
end;

function less (arg : string)->boolean;
begin
    return 0 > cfunction
        Leda_string_compare(self, arg)->integer;
end;

function plus (arg : object)->string;
begin
    return cfunction
        Leda_string_concat(self, arg.asString())->string;
end;

function print ();
begin
    cfunction Leda_string_print(self);
end;

function length ()->integer;
begin
    return cfunction Leda_string_length(self)->integer;
end;

function subString (start, len : integer)->string;
const
    selfLength := length();
begin
    start := start % selfLength;
    if len < 0 then
        len := 0;
    if start + len > selfLength then
        len := selfLength - start;
    return cfunction
        Leda_string_substring(self, start, len)->string;
end;

function index (pattern : string)->integer;
const
    patternLength := pattern.length();
var
```

```
        position : integer;
    begin
        for position := 0 to length() do
            if pattern = subString(position, patternLength) then
                return position;
        return NIL;
    end;

end;
```

C.8 The class array

The array data structure is the only collection data type defined in the standard library. An array is a fixed-length indexed collection of elements having the same type. The function size returns the number of elements held in the array. Elements are placed into an array using the function atPut, and extracted from the array using the function at. The relation items is used to iterate over the values held in an array.

New arrays can be generated using the function newArray. The user must supply both the lower and upper index values for the array. A literal array is generated by surrounding a list of similarly-typed values with square brackets. In literal arrays index values always begin at zero.

```
class array [T : object] of equality[array];
var
    lowerBound : integer;
    higherBound : integer;
    data : object;

    function size ()->integer;
    begin        { return number of elements in collection }
        return 1 + (higherBound - lowerBound);
    end;

    function at (index : integer)->T;
    begin
        if index.between(lowerBound, higherBound) then
            return cfunction
                Leda_object_at(data, index - lowerBound)->T
        else
            return NIL;
```

```
    end;

    function atPut (index : integer, newVal : T);
    begin
        if index.between(lowerBound, higherBound) then
            cfunction Leda_object_atPut(data, index - lowerBound, newVal);
    end;

    function items (byRef val : T)->relation;
    var
        index : integer;
    begin
        return (lowerBound <= higherBound) &
            integerRange(lowerBound, higherBound, 1, index) &
            val <- at(index);
    end;

    function asString ()->string;
    var
        result : string;
        v : T;
    begin
        result := "[";
        for items(v) do
            result := result + v + " ";
        return result + "]";
    end;
end;

function newArray [T : object] (lb, hb : integer)->array[T];
begin
    if (hb > lb) then
        return array[T](lb, hb,
            cfunction Leda_object_allocate((hb-lb)+1)->object)
    else
        return NIL;
end;
```

C.9 Testing for Definition

The function `defined` can be used to test whether a value is defined. It does this by simply comparing against the value `NIL`.

```
function defined (arg : object)->boolean;
begin
    return ~ cfunction Leda_object_equals(NIL, arg)->boolean;
end;
```

For efficiency reasons some implementations may elect to implement this function as a built-in operation rather than as Leda code in the standard library. The only noticable difference (other than efficiency) would be that `defined` then becomes a keyword, and can not be used in any other context.

C.10 Input and Output

Output to the standard output device is performed using the function `print`. The argument to this function is declared as object, and thus can be any value. It is converted into a string using the function `asString` (see Section C.1), and the resulting string is then displayed. If the argument is undefined a literal value is displayed.

```
function print (arg : object);
begin
    if defined(arg) then
            { convert to string, then print }
        arg.asString().print()
    else
        "(undefined)".print();
end;
```

Input is performed line by line. The function `readLine` reads a single line from the standard input, assigning it to a by-reference parameter. A boolean true value is returned if input is successful, while a boolean false value is returned on end of input.

```
function readLine (byRef line : string)->boolean;
begin
    line := cfunction Leda_stdin_read()->string;
    return defined(line);
end;
```

C.11 Relations

From the programmers point of view, relations are defined using the relational assignment operator (<-), conjunctions and disjunctions, and boolean functions. The relational assignment operator is implemented using the following function:

```
function Leda_arrow (byRef left : object, right : object)->relation;
begin
    return function(future : relation)->boolean;
        var
            save : object;
    begin
            { save the current value of the left argument }
        save := left;
            { then change it to the right }
        left := right;
            { try the task to be done }
        if future(trueRelation) then
                { worked, report success }
            return true;
            { didn't work, restore the old value }
        left := save;
        return false;
    end;
end;
```

Conjunction of two relations is implemented by the following:

```
function and (left : relation, byName right : relation)->relation;
begin
        { conjunction of two relations – do one after the other }
    return function(future : relation)->boolean;
        begin
            return left(function(f : relation)->boolean;
                begin
                    return right(future);
                end);
        end;
end;
```

The backtracking mechanism is found in the implementation of the disjunction operator:

```
function or (left : relation, byName right : relation)->relation;
begin
        { disjunction of two relations - do backtracking }
    return function(future : relation)->boolean;
        begin
            if left(future) then return true;
            return right(future);
        end;
end;
```

Conversions between relations and booleans are performed using a pair of functions:

```
function booleanAsRelation (byName x : boolean)->relation;
begin
        { convert boolean into a relation }
    return function(future : relation)->boolean;
        begin
            return x & future(trueRelation);
        end;
end;

function relationAsBoolean (future : relation)->boolean;
begin
        { convert relation into a boolean }
    return future(trueRelation);
end;
```

The function `unify` is used as a simple approximation to Prolog-style unification (more complex unification techniques can also be written in Leda, but are generally not necessary). It attempts to make the left argument the same as the right, either because they are both defined and equal, or by assigning one the value of the other.

```
function unify [T : equality] (byRef left : T, right : T)->relation;
begin
    if defined(left) then
        return left = right
    else
        return left <- right;
end;
```

Finally, the function `integerRange` defines a simple relation which will generate values from a range of integers. The starting and ending values and the step amount must be specified. The ending value must be reached exactly, or an infinite regression will result.

```
function integerRange
    (low, high, step : integer, byRef ident : integer)->relation;
begin
    return ident <- low
        | (low <> high) & integerRange(low + step, high, step, ident);
end;
```