Appendices

Appendix A

Introduction to Leda

In this appendix we explain the syntax of the Leda programming language. The text, beginning in chapter 2, provides a discussion of how programs are constructed in Leda in a style characteristic of each of the four targeted paradigms, and how these paradigms are then used in the solution of problems. Our mode of exposition here will be purposely informal, being designed to simply give the reader an intuition as to how to read Leda programs. A more formal characterization of the Leda syntax is provided by the grammar given in Appendix B. Readers who feel comfortable learning syntax by examining completed programs may wish to simply skim this appendix, or omit it altogether.

The syntax of any programming language represents only half the tools used in the solution of problems. Equally important are the fundamental functions and data types which are employed in almost all nontrivial programs but which are not, strictly speaking, part of the syntax for the language. Often, as in Leda, these are written in the programming language itself, and are referred to as a standard "library" of abstractions. Portions of the Leda standard library will be introduced as necessary in this chapter. The complete Leda standard library is fully documented in Appendix C.

The syntax of Leda follows the "Algol" school of language design. What this means is that the syntax is similar to that used by a large number of programming languages, all of which trace their ancestry to the languages Algol-58 and Algol-60, designed in the early days of high-level computer languages. Other programming languages in the Algol school include Pascal, Simula, Modula, and Ada.

A.1 Program Structure

The overall structure of any Leda program is a series of zero or more declarations, followed by a single compound statement. This compound statement is the main body, and describes the actions that will be performed when the program is executed. The following is one of the simplest Leda programs, containing no declarations and only a single statement within the main body:

```
begin
        { this is a very simple program }
    print("hello world");
end;
```

Any material between curly-brace characters is interpreted by Leda as a comment. Comments are not used by the Leda system during the process of execution, and are merely provided as a device to make programs easier for humans to read. Comment text usually describes the general purpose of a section of code, or explains features that might not be easily understood by examination of the executable source code itself.

The text `"hello world"` is a literal string, one of several different forms of constant values in Leda. Other common forms of literals include integer and real constants, and array literals. These will be described in Section A.7.

The function `print` is part of the standard library, and is used to display at run-time the value of an expression. The semicolon is a statement terminator in Leda, and is used to indicate the end of both declarations and execution statements.

Although the contents of the standard library has been fixed, implementations of Leda are purposely free to define their own mechanism for the attachment of libraries, such as the standard library, to programs. The current Leda system performs this by means of a special statement called an *include* statement, which must appear in the declaration section of a program. An example is the following:

```
include "stdlib.led";
begin
    print("hello world");
end;
```

For the sake of readability we will henceforth omit include statements from our example Leda programs.

There are five different types of declarations in Leda. These are constant declarations, variable declarations, type definitions, function definitions, and class definitions. The simplest of these, constant declarations, simply associates a symbolic name with a given value. An example is the following:

```
const
    outputString := "hello world";
begin
    print(outputString);
end;
```

The keyword `const` indicates that subsequent expressions represent a series of one or more constant declarations. The value attached to a constant declaration can be any type of expression. If an expression defining a constant value cannot be determined at compile time, then the values are computed at execution time in the order they are given and prior to any subsequent statements.

Constant values can be used within a program, such as here where the value is passed as argument to the function `print`. However, constant names cannot be modified; that is, they cannot be used as the target of an assignment statement.

A.2 Variables and Assignment

A sequence of variable declarations is introduced using the `var` keyword. Each variable must be associated with a type, which indicates the nature of the values that can be assigned to the variable. The type `string`, used in the following example, is one of several basic data types defined in the standard library. Other commonly used types include `integer`, `real`, and `array`. Unlike constants, variables can be redefined by assignment statements so as to hold different values during the course of execution.

```
var
    outputVariable : string;
begin
    outputVariable := "hello world";
    print(outputVariable);
    outputVariable := "hello again";
    print(outputVariable);
end;
```

A.3 Types

There are two major categories of types in Leda, function types and class types. Functions are first-class values in Leda. This means that variables can hold values of type function, functions can be passed as arguments to other functions, and even be returned as the result of other functions. The following declaration shows a variable being declared as holding a value representing a function that takes two parameters, one passed by reference, and returning a boolean value.

```
type
    funVar : function(integer, byRef string)->boolean;
```

The major data structuring mechanism is the class, to be described in section A.6. Commonly used data types defined by classes in the standard library include `integer`, `real`, `string`, and `array`. The latter is the only true collection data structure defined in the standard library. The declaration of an array provides only the type of elements held by the structure:

```
var
    anArray : array[integer];
```

The extent and size of the array is determined when an array value is generated, using the function `newArray`. The arguments represent the lower and upper bounds of the array index values.

```
    anArray := newArray[integer](72, 83);
```

A.4 Statements

We have seen two statement forms already; namely, the assignment statement and the compound statement used to form the main body of a program.

Conditional statements are formed using the keywords `if` and `then`, and an optional `else` part. The condition expression (between the `if` and `then` keywords) must result in a value either of type boolean or type relation. The latter will be explained in Section A.8. The condition is evaluated, and if true the statement following the `then` keyword is executed, otherwise the statement following the `else` keyword is executed. If no `else` portion is provided then no action is performed in the case when the condition is false.

```
if stateOfMind = "happy" then
    print("hello world")
else
    print("go away world");
```

Condition statements can be nested arbitrarily, with `else` portions being matched to the nearest surrounding `if` statements.

Simple loops are formed using a `while` statement. The loop executes the statement repeatedly (zero or more times) as long as the condition portion is true.

```
while stateOfMind = "happy" then begin
    print("hello world");
    ...
```

```
end;
```

Loops that use an integer variable running through a progression of successive values can use the `for` statement:

```
for i := 1 to 10 do begin
    print("hello ");
    print(i);
end;
```

The `for` statement is more commonly used in conjunction with relations. This will be discussed in Section A.8.

The remaining statement forms are procedure calls and the `return` statement. These will be discussed in the next section.

A.5 Functions

A function declaration provides an abstraction, or encapsulation, of instructions. A function declaration begins with the keyword `function`, followed, in order, by a function name, an argument list, an optional return type, and a semicolon. This is immediately followed by a series of zero or more declarations and a function body.

```
function max (a, b : integer)->integer;
var
    m : integer;
begin
    if a > b then
        m := a
    else
        m := b;
    return m;
end;
```

Functions need not return a value. In situations where no value is returned, the indication of the return type is omitted.

```
function printmax (a, b : integer);
begin
    print("maximum of " + a + " and " + b + " is ");
    print(max(a,b));
end;
```

A function that does not return a value is executed purely for a side effect, such as here where the function prints the values of certain arguments. Such a function is often termed a *procedure*, although Leda (unlike certain other Algol-like languages) does not use the word `procedure` as a keyword. A procedure can only be invoked as a statement, not as an expression in which the return value is needed for a subsequent computation.

Note the overloading of the `+` operator in the preceding function. When the left argument is a string, the `+` operator is used to perform conversion of the right argument into a string and catenation with the left argument. This is often used in `print` statements to produce formatted output.

A `return` statement can be used within the body of a function. If a return type is specified for the function an expression of the same type must be provided with the `return` statement; otherwise, if the function does not provide a return type then no expression should be used in this statement.

Functions introduce a new identifier name scope. Names declared within a function can be used only within that function, and have no meaning outside the function body. Names can match identifiers declared outside the function, in which case the new local name hides the use of the identifier in the surrounding context.

```
var
    x : string;     { global declaration }

function demonstrateHide();
var
    x : string;     { hides global declaration }
begin
    x := "inner value";
    print(x);     { prints inner value }
end;

begin
    x := "hello world";
    demonstrateHide();
    print(x);     { prints global variable – hello world }
end;
```

There are three parameter passing modes permitted in Leda. If no parameter passing mode is specified in the argument list, then parameters are passed by value. If the `byRef` keyword appears then parameters are passed by reference. In this case an assignment to the parameter identifier made within a function will be reflected in a change to the actual argument value. Finally, if the `byName` keyword appears then parameters are passed by name. This indicates that the evaluation of the argument value takes place at each point the argument value is used within the function, rather than at the point the function is invoked, as is conventional. Usually this feature is used to delay or avoid the evaluation of parameters which may not be used in certain circumstances. For example, the following function mimics the behavior of the logical *and* operator, only evaluating the second argument if the first one is true.

```
function logicalAnd (a : boolean, byName b : boolean)->boolean;
begin
    if a then
        return b { b evaluated here }
    else
        return false; { b not evaluated in this case }
end;
```

A call on a function that does not yield a return value cannot be made from within an expression in which a value is expected. Conversely, functions which yield values cannot be invoked as procedures at the statement level.

A.6 Classes

Classes are the major data structuring mechanism used in Leda programs. A class declaration begins with the keyword `class`. This is followed by a class name. Like functions, classes introduce a new identifier scope. Declarations appearing within this name scope define the data fields and operations associated with instances of a class. Classes are terminated by the `end` keyword.

```
class intList;
var
    value : integer;
    nextElement : intList;

    function head ()->integer;
    begin
        return value;
    end;

    function includes (testValue : integer)->boolean;
    begin
        if testValue = value then
            return true;
        if defined(nextElement) then
```

```
            return nextElement.includes(testValue);
        return false;
    end;

    function onEach (f : function(integer));
    begin
        f(value);
        if defined(nextElement) then
            nextElement.onEach(f);
    end;
end;
```

The keyword `of` can appear in a class heading to indicate one class inherits data fields and operations from another class. All fields and operations associated with the parent class (the class named after the `of` keyword) are immediately available without redefinition in instances of the new class.

```
class intSet of intList;

        { add only if not already in collection }
    function addToSet(x : integer);
    begin
        if ~ includes(x) then
            add(x);
    end;

end;
```

Instances of classes are created using a special type of function called a *constructor*. A constructor function is automatically created for each class declaration. The name of the constructor function is the same as the class name. Constructors can be invoked with no arguments, in which case all data fields are initialized to the value `NIL`, or with arguments that exactly match the order and number of data fields in the class, in which case the data fields are initialized with the argument values.

```
var
    listOne : intList;
    listTwo : intList;
begin
    listOne := intList(3, NIL);
    listTwo := intList();
end;
```

A.7 Expressions

The simplest form of expression are literal values. Leda recognizes integer, string and real number constants. Arrays of constant values can be formed by surrounding with square brackets a list of similarly-typed values:

```
const
    colors := [ "red", "blue", "yellow" ];
```

Identifiers (variables or constants) can be used as expressions when they are within the scope in which they are declared. The identifier `NIL` is associated with the value used to initialize all variables. Whether or not a variable is holding the value `NIL` can be determined by the standard library function named `defined`. An example occurs in the two functions `includes` and `onEach`, in the `intList` class description shown earlier.

An expression which results in a class value can be followed by a period and an identifier representing one of the local variables held by the class, or a function defined in the class scope. In the former case the result is the value of the local variable, while in the latter case the result is a value of type `function`, which is most commonly then provided with arguments and invoked. An example occurs in the function `onEach` shown earlier in the `intList` class description.

While a constructor is used to form an instance of a class by defining values for each data field, the opposite action is performed by a pattern matching expression. The pattern matching operator is the keyword `is`. This is preceded by a value, and followed by a class name, optionally followed by a list of identifiers. If the value to the left of the keyword is an instance of the named class the expression values as `true`, and results in `false` otherwise. If a list of identifiers is given and if the pattern match is successful, then the values of the data fields being maintained by the class instance are copied into the named variables. The variables must match in type and number the data fields declared in the class description.

```
    if aList is intList(val, next) then ...
```

Leda supports the usual range of arithmetic and relational operators for numeric and string values. A table of these can be found in Appendix B.

Values of type function can be created as an expression. The syntax is exactly the same as for function declarations, but without the associated function name.

```
        { print out the values held in the list }
    aList.onEach(function(val : integer);
            begin
```

```
                print(val);
                print(" ");
            end);
```

The keyword `cfunction` introduces a run-time system call. This invokes a function that is provided as part of the run-time library, used to provide low level operations that cannot be described in Leda. Usually the argument or return types of such calls are not checked for validity.

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

Calls on cfunctions that do not yield result values can be invoked as procedures, in which case the description of the return type is omitted. Implementations are free to impose further restrictions on the use of this feature.

A.8 Relations

One of the more unusual data types in Leda is the `relation`, which is used extensively in the logic programming paradigm. A relation can intuitively be imagined as a type of boolean value. However, relations also implement the backtracking mechanism used by logic programs. This is accomplished by providing specialized meaning to the logical operations of conjunction and disjunction ("and" and "or") when used with relations.

The logic programming paradigm makes extensive use of functions and relations. For example, the following describes a relation to determine if two names represent a parent-child pair, building on top of relations that encode father-child and mother-child pairs:

```
    { parentOf relation – true if parent is parent to child }
function parentOf (byRef parent, offspring : string)->relation;
begin
    return fatherOf(parent, offspring)
        | motherOf(parent, offspring);
end;
```

By combining relations with the by-reference parameter passing mechanism, as shown here, relations can be used to set, as a side-effect, identifiers passed as argument values.

The most primitive form of relation is formed using the relational assignment operator, $<-$. Like an assignment, this operator has the effect of changing the value of the identifier written on the left so that it maintains the value of the expression written on the right. The value of the operator is the boolean value `true`. However, if the relational backtracking mechanism forces execution to return to the operator (in search of an alternative way of satisfying the relation), then the identifier is restored to its original value.

A relation is ultimately invoked either in the condition portion of an `if` statement or within a `for` statement. The `if` statement form is used to simply determine whether or not the relation can be satisfied at all. For example, the following example, which uses the genealogical database from Chapter 19, evaluates the true portion of the `if` statement if the database contains an individual known to be a parent of Leda, and executes the false portion if no such individual is known:

```
if parentOf (name, "Leda") then
    print("Leda is child of " + name)
else
    print("no parent for Leda in database ");
```

Relations can also be invoked using a `for` statement. The `for` statement form runs through all possible ways of satisfying a relation, executing the associated statement portion of each. For example, the following prints the names of all parents of Leda (that is, both the mother and father).

```
for parentOf (name, "Leda") do
    print("Leda is a child of " + name);
```

A.9 Parameterized Types

A powerful feature of Leda is the ability to provide type parameters to classes and to functions. The type parameter facility permits general purpose data structures to be constructed, which do not overly constrain the nature of the values they will maintain. For example, list or set data structures can be generated that will hold values of various different types. Type parameters are distinguished syntactically from value parameters by being surrounded by square brackets, instead of the parenthesis used in value parameters. For example, the following class declaration might be used in a general list structure:

```
class list [ X : object ];
var
    value : X;
    next : list[X];
    ...
end;
```

Within the class declaration the identifier `X` can be used to represent a type, however the type is unknown at the time the data structure is developed. Before an instance of the class can be constructed this type parameter must be filled, as in the following examples:

```
var
    listOne : list[integer];
    listTwo : list[real];
```

The fact that `X` is declared as an instance of `object` indicates that `X` must be a subclass of class `object`. This is no real restriction, since in Leda everything is a subclas of class `object`. However, constraining values to be subclasses of `equality` or `ordered` is more common, and ensures that the type-arguments provide certain minimal functionality.

Functions can also be written with type parameters, as in the following example:

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

When such a function is invoked the value associated with the type parameter must be specified:

```
    unify[integer] (head, alist.value)
```