STUDY OF LEDA PROGRAMMING LANGUAGE

Course Name: Concepts of Programming Languages
Course Code: CS 5314
Faculty: Dr James D. Arthur

Submitted By:
Ravi Gurnani [rgurnani@vt.edu]
Sumeet Gupta [sumeet@vt.edu]
OVERVIEW OF LEDA

The Leda programming language exemplifies what is described as a multi-paradigm language. As such, Leda spans the boundaries of the well known programming language models as the imperative, functional and logic models. Also included in Leda's language arsenal is support of the object oriented programming model.

Leda was developed in the early 1990's by Timothy A. Budd an associate professor at Oregon State University Corvallis. The attempt to define a language that would support multiple programming paradigms was inspired by Budd's excitement about OOP and other innovations occurring in the study of programming languages at the time. The initial attempts at creating the new language started with the development of compilers by students under Budd's direction as well as other researchers at the University of Nice, in France. In 1993 Budd took a sabbatical leave from his University post at Oregon State to teach in Europe and complete research on the new programming language. During this year, he penned the definitive text on the Leda: "Multiparadigm Programming in Leda".

The Leda syntax is influenced by the ALGOL programming language syntax. Leda's syntax is similar to many of the other languages that are direct decedants of the ALGOL-58 and ALGOL-60 programming languages. These include languages such as Pascal, SIMULA, Modula, Oberon and Ada.
BASIC FEATURES OF THE LEDA

Program Structure

The overall program structure of a Leda program is a series of zero or more declarations followed by a single compound statement making up the body of the program. Comments are supported within the program text by use of the curly brackets. The include special statement in Leda allows the attachment of standard libraries to the base program.

Declaration Statements

Declarations exists for constant and variable types, function definitions and class definitions.

Variables and Assignments and Data Types

A sequence of variable declarations is started using the keyword var. Each variable declaration must be associated with a type name. The following basic data types are defined in Leda's standard library: string, integer, real, and array. A simple programming example use of variable assignment is shown below:

```led
include "stdlib.led"
var outputString : String;
begin
    outputString := "hello world";
    print(outputString);
end;
```

Two categories of types exist in Leda besides the basic data types. These include the function types and the class types.
Function Types

Functions are considered first class values in Leda. This means that variables can hold values of type function and that functions can be passed as arguments to other functions. The value returned by a function is indicated at the end of the function declaration using the -> symbol as shown in the example below:

```
type funVar: foo (integer, byRef String)->boolean
```

Class Types

Class types are major data structuring mechanisms used in Leda. They are declared using the keyword `class` and introduce a new identifier scope.

Modular Units:

Functions

Functions provide an abstraction that represents a grouping of instructions. All functions in Leda begin with the keyword `function` followed by a function name, an argument list, an optional return type and a terminating semicolon. This is then followed by zero or more variable declarations and the function body.

```
function min(a,b : integer)->integer
begin
  if a < b then
    return a;
  else
    return b;
end
```

Functions are not required to return a value. When no value is returned by a function, the indication of return type is removed.
Parameter Passing Techniques

Three parameter-passing modes are 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 the convention.
ADVANCED FEATURES

Object oriented programming features such as data hiding, inheritance and polymorphism are supported in Leda through the use of Classes and dynamic binding. The logic programming paradigm is supported through the use of another abstract data type called the relation. The functional programming paradigm is supported in Leda by allowing the programmer the ability to develop fundamental operations such as: reduction, mapping, and filtering.

Dynamic Binding

In addition to static binding, Leda supports the concept of *dynamic dispatch* which is sometimes referred to as *dynamic binding*. Dynamic binding allows the programmer to use a technique called *polymorphism* which is one of the more powerful features of object oriented programming. With static binding, type information for a variable or function is that declared at compile-time from variable declaration. Dynamic binding uses type information at run-time based on the information contained in the object itself. This gives Leda the ability to determine at run time which of several different functions to execute, based on differences in the type of value held by a variable.

Classes

Classes are the major data structuring mechanism used in Leda programs. A class declaration begins with the keyword `class`, 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.

```plaintext
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;
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 in Leda using a special type of function called a constructor.

Relations

Relations are abstract data types that are used extensively in the logic programming paradigm. Budd describes these as a type of Boolean value. They provide specialized
meaning to the logical operations of conjunction and disjunction. The following represents a parental relation:

```plaintext
{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;
```

**Parameterized Types**

Leda has support for parameterized types which is a feature identical to Templates in the C++ programming language. The type parameter facility permits the construction of general purpose data structures which do not overly constrain the nature of the values they will maintain. The type parameters are distinguished syntactically from value parameters by their encasement in square brackets, instead of parentheses as follows:

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

**Functional Programming Support:**

**Reduction**

The *reduction* of a list is characterized by an identifying value and a binary function. Each value of the list is considered in turn. The binary function is applied, using for the first argument either the identity or the result of the reduction so far, and the second argument is the list item. An example of a reduction presented by Budd is the reduce() function used in sorting a list of numbers:
function insert( x : integer, aList : intList)->intList;
begin
  if empty(aList) then
    return intList(x, NIL)
  else
    if x < head(aList) then
      return intList(x, aList)
    else
      return intList(head(aList), inset(x, tail(aList)));
  end;

function reduce (aList : intList, indent : intList, binFun: function(integer, intList)->intList)->intList;
begin
  if empty(aList) then
    return ident
  else
    return binFun(head(aList), reduce(tail(aList), ident, binFun));
end;

function sort (aList:intList)->intList;
begin
  return reduce(aList, NIL, insertion);
end;

reduce(2 5 1 4 7)
insert(2, reduce(5 1 4 7))
insert(2, insert(5, reduce(1 4 7)))
insert(2, insert(5, insert(1, reduce(4 7))))
insert(2, insert(5, insert(1, insert(4, reduce(7))))))
insert(2, insert(5, insert(1, insert(4, insert(7, reduce(NIL))))))
insert(2, insert(5, insert(1, insert(4, insert(7, NIL)))))
insert(2, insert(5, insert(1, insert(4, 7))))
insert(2, insert(5, insert(1, 4, 7))
insert(2, insert(5, 1 4 7))
insert(2, 1 4 5 7)
1 2 4 5 7
Mapping

A mapping of lists is characterized by a function that takes a list as argument and transforms each element into a new value by some means. In Leda the general purpose procedure named `map` can be used to generate a map.

Filtering

A filter of a list is a new list in which elements that fail to match a given specification have been removed. In Leda a programmer can define a function `filter` that will take a list and a predicate as arguments and return the filter of the list relative to evaluation of the predicate on each of the values in the original list.
CONCLUSION

**Leda** is a multiparadigm programming language. The idea of a multiparadigm language is to provide a framework in which programmers can work in a variety of styles, freely intermixing constructs from different paradigms. The techniques supported by Leda include imperative programming, the object-oriented approach, logic programming, and functional programming.