Multiparadigm Data Structures in Leda

Timothy A. Budd
Department of Computer Science
Oregon State University
Corvallis, Oregon
97331
budd@cs.orst.edu

September 20, 1994

Abstract

Multiparadigm programming is a term used to describe a style of software development that makes use of facilities originally designed in support of a number of different programming language paradigms. In this paper we illustrate our conception of multiparadigm programming, by describing how various data structures can be implemented in the programming language Leda. Leda is a strongly-typed compiled multiparadigm programming language that we have been developing over the past several years. Our exposition serves both to illustrate the idea of multiparadigm programming, and to describe the features of the language Leda.

1 Introduction

Programming languages are often divided into different schools, or paradigms. The term "paradigm", meaning model or world view, is intended to indicate the difference is one of outlook or perception, rather than a fundamental incomparability. From a simple theoretical point of view, all programming languages which have sufficient power to simulate a Turing machine, and this includes almost all languages of interest, are equally powerful. Thus arguments about the "power" of programming languages, if by power one means the ability to perform a given set of tasks, are most often meaningless. Such arguments were called by the late Alan Perlis "Turing Tarpit" arguments, because they are so difficult to disentangle oneself from, and are so fundamentally useless.

Different programming language paradigms are important not because they provide new capabilities that are not available to languages and programmers working in other models, but because they point the way towards thinking about problems in different fashions. Often these differences can be seen to be simply the organization of the same information in contrasting styles. Similar observations have often been made about natural languages [Who56]. An understanding of these differences, and of the problem areas for which different outlooks are most appropriate, is a help to programmers attempting to select the most suitable approach for solving a given task.

Our intent in this paper is to illustrate a style of program development that draws upon features from a number of different programming language paradigms. In particular, we will use aspects of imperative programming, object-oriented programming, functional programming, and logic (or relational) programming. We will show how these various language paradigms can be integrated in a single problem domain, and how each can derive benefit from the others. The problem we will consider is the creation of a few simple data structures for lists and two-dimensional tables. In addition, we will use the development of this example as a means to introduce the multiparadigm programming language Leda.

2 Objects and Classes

We start our discussion by describing some of the object-oriented features of Leda. Our list data structure will be implemented using two classes (Figure 1). The class List is the visible class accessed by users of the list abstraction. This class merely defines the head of a chain of linked-list values. The class ListElement describes an individual link in this linked list. Normally users of the list abstraction would never be concerned with this latter class.[1]

The structure of a class declaration in Leda is slightly different from class declarations in other object-oriented languages, such as C++ or Object Pascal. In Leda a class description is divided into two components. The initial portion indicates those fields that are unique to each instance of the class. The items following the optional shared keyword indicate fields that are shared in common by all instances. The motivation for this design and some of the implications of this decision are explored in more detail in [Budd 91c].

Both the classes List and ListElement are parameterized classes. That is, they are qualified by an argument (X in both classes) that is unbound at the time the class is defined. To declare an instance of such a class it is necessary for the user to supply a binding for this argument. For example, the user can declare a variable as maintaining a list of integers, or a list of reals. Examples of such declarations will be provided shortly.

Even without knowing the binding for the qualifier of a class, it is possible to use the argument name as a type in certain situations. For example, it is possible to declare a field in the unshared portion of a class. An example is the datum field in the class ListElement, which holds the actual value being maintained by the class. It is not possible to declare such a field in the shared portion, although other, bound, types can be used to declare data fields in the shared portion of a class. Notice the qualifier being used to bind the argument in a field representing an instance of a further parameterized type. This occurs in both the next field in the class ListElement and the data field in the class List.

The qualifier for a parameterized class can be used as an argument type in methods associated with the class. We see this in almost all the methods associated with these two classes.[2] A method

[1] In designing Leda there were a number of topics that, while important in a wider sense, seemed irrelevant to the issues we were interested in exploring. Among these are the possibilities of separate compilation, a Leda programming environment, and the purposeful hiding of information. As an example of the latter, it has been suggested that it would be desirable to hide the existence of the class ListElement from users of the List abstraction. While we see some merit in this position, the introduction of such protection mechanisms would have added additional complexity to the language.

[2] The unbound variable Z in the method reduce is more complex, and will be discussed in detail in Section 4.

```
type
   ListElement := class(X)
         datum : X;
         next : ListElement(X);
      shared
         add : method(X);
         remove : method(X) → ListElement(X);
         items : relation(var X);
         onEach : method(function(X));
      end;

   List := class(X)
         data : ListElement(X);
      shared
         add : method(X);
         addToEnd : method(X);
         addNew : method(X) → List(X);
         includes : method(X) → boolean;
         remove : method(X);
         reduce : method(Z, function(Z, X)→ Z) → Z;
         count : method(function(X)→boolean) → integer;
         items : relation(var X);
      end;
```

Figure 1: The classes List and ListElement

```
method ListElement.add(value : X);
begin
    if defined(next) then
        next.add(value)
    else
        next := ListElement(value, next);
end;

method List.add(value : X);
begin
    data := ListElement(value, data);
end;

method List.addToEnd(value : X);
begin
    if defined(data) then
        data.add(X)
    else
        add(X);
end;

method List.addNew(value : X) → List(X);
begin
    return List(ListElement(value, data));
end;
```

Figure 2: Methods used to add elements to a list

is one of three subprogram abstractions found in Leda, the other two being functions and relations. A method differs from a function in that a method is always associated with a specific class, and thus the invocation of a method by message passing involves an implicit receiver for the message. Nevertheless, a method is very close to a function and it is relatively easy to both convert a method into a function and conversely convert a function into a method. An example of the former will be given in subsequent discussion.

The methods which implement adding elements to a list are shown in Figure 2. Note that these methods can be developed, and code can even be generated for them, without any information regarding the nature of the unbound type X.[3]

All variables in Leda are either defined or undefined. Variables are initially undefined. Furthermore, unlike most programming languages, it is possible to query a variable to determine whether

[3]This property actually depends upon the fact that internally all values have the same representation; namely a pointer. These pointers are never explicitly manipulated by the programmer, and the language itself does not support a pointer type.

```
method ListElement.remove(value : X) → ListElement(X);
begin
   if datum = value then
      return next
   else if defined(next) then
         next := next.remove(value);
   return self;
end;

method List.remove(value : X);
begin
   if defined(data) then
      data := data.remove(value);
end;
```

Figure 3: Methods used to remove an element from a list

it has yet been defined. We see an example of this in the method ListElement.add, which adds an element to the end of a linked list. A test is first made to see if the link field is defined, that is, whether there are more elements in the list. If so, then the value to be added is passed down the linked list. If not, then a new link is formed. This is accomplished using the *constructor* for the class LinkedList.

A constructor is a function used to construct a new instance of a class. Constructors are generated automatically by the Leda compiler for each new class declaration. There are two forms of constructor. A constructor with no arguments creates a new instance of a class where all fields are initially undefined. When arguments are passed to the constructor, as they are here, they must match in type and number the fields in the unshared portion of the class. The values of the arguments are then used to initialize the fields in the newly created object. Thus in this case a new link is created with the argument value used to initialize the datum field, and the currently undefined value of the field next in the current node used to initialize the next field in the newly constructed node. The Leda system tracks references to dynamically created data, and releases such storage when it is no longer accessible.

There are three methods used to add elements to a list structure. The method add places a new element at the front of a list, by building a new link and changing the data field. The method addToEnd adds the element to the end of the list, using the method defined in class ListElement. The method addNew does not modify the current list, but instead creates a new list head and adds the given element to this new list. This new list is then returned as the value of the method. The new list and the original list share the other elements in common. (We could have just as easily duplicated the original list, but chose not to for the sake of brevity of presentation.)

Figure 3 shows the methods used to remove an element from a list. The method used in the underlying class ListElement illustrates the use of the pseudo-variable self. Methods carry with them an implicit parameter which represents the receiver for the message which invoked the method. This implicit parameter can always be accessed within a message using the name self. The type

```
struct link {
    int value;
    link * next;
    } *p;

struct list {
    struct *link;
    } aList;

sum = 0;
for(p = aList→link; p; p = p→next)
    sum += p→value;
```

Figure 4: Summing the values of a linked list in C

associated with this pseudo variable is the type of the object which received the message, which must necessarily be a descendant of the class in which the method is defined.

3 Relations

The techniques used in the previous section were all object-oriented or imperative in nature. In this section we will start to consider how other paradigms, in particular the relational (or logic-programming) paradigm can be integrated into Leda programs. The relations used in this section are somewhat unusual. More conventional logic programming in Leda is described in [Budd 91a].[4]

The specific problem we wish to address in this section is how the programmer can usefully iterate over elements of a list; for example to print out the list or sum the values in the list. Experience tells us that a user of a data abstraction, such as our lists, will usually not be the same as the developer of the code. Thus software engineering practices, such as Parnas' Principles [Parnas 72], dictate that the user of a data abstraction should require as little information about the internal structure of the data abstraction as possible.

Previous languages have provided various models that could be emulated in addressing this problem. The most common technique is that used in languages such as C or Pascal. In these languages iteration is accomplished using a loop, such as a **for** or **while** statement. This loop performs the intended action, but then must explicitly access the internal structure of the list. For example a method in C to sum the values of a linked list (Figure 4) must have intimate knowledge of the location and name of the link fields in structures with which the linked list is implemented. Such exposure of internal implementation details flies in the face of the principles of abstraction and information hiding.

A second model is provided by languages such as CLU [Liskov 81], which supply a special type of procedure, called an *iterator*. An iterator acts as a procedure, and when invoked yields the

[4]It should be noted that the syntax of Leda has evolved over time. Some of the examples used in the earlier paper are no longer legal Leda programs.

```
List = cluster [ t : type ] is items, add, ...
    rep = oneof[pair : pair, empty : null]
    pair = struct[head : t, tail : List[t]]
    ...
    items = iter(lst : cvt) yields(int)
        tagcase lst
            tag pair(p : pair): yield(p.head)
                for i:t in items(lst.tail) do
                    yield(i)
                end
            tag empty:
            end
        end items
end List

...
sum : int := 0
for val:int in List[int]$items(aList) do
    sum := sum + x
```

Figure 5: An iterator in CLU

first element in a collection. It can later be restarted and will return the next element. It can be restarted repeatedly until all elements have been exhausted. In CLU iterators are controlled by a special form of the for loop (Figure 5). Iterators are generalized to *generators* in languages such as Icon [Griswold 90] and others [Budd 87, Berztiss 90]. The iterator solution has the beneficial advantage of isolating the control details from the user of the abstraction. Nevertheless, it does so only at the cost of introducing a new type of procedural abstraction to the language. One of our major objectives in designing the language Leda was to keep the language as small and simple as possible. Thus a solution formed out of the existing elements of the language would be preferable to the introduction of a new mechanism.

A third potential model is provided by the language Smalltalk [Budd 87]. In Smalltalk iteration is accomplished through the interaction of message passing and the ability to form *blocks*. A block is basically a lexically scoped function which can be passed as argument to a method, and when executed evaluates in the context in which it was defined. Thus a list in Smalltalk, for example, provides a method such as do: (Figure 6). This method takes as argument a block and executes the block once for each element of the list, passing it as argument the value of the element. While we possess the basic mechanisms (namely lexically scoped functions as first class values) to emulate this approach in Leda, the utility of this technique in Smalltalk is facilitated by an unusual semantic rule. Namely a return statement inside a block is treated as a return not from the "block function" but from the function in which the block was created. Supporting this semantics is difficult and error-prone, and thus we rejected this approach. (Although, as we will explore in the next section, passing functions as arguments to other functions is an extremely powerful technique, and quite

```
sum ← 0.
aList do: [ :x | sum ← sum + x ].
```

Figure 6: Iterating over elements of a list in Smalltalk

useful in Leda).

Instead, in Leda we chose to use the existing facilities provided by relations. A relation is a subprogram abstraction, similar to a function or a method. By assigning values to arguments (typically call-by-reference arguments), a relation attempts to "satisfy" some property defined by the body of the relation. In one sense a relation can be viewed as a boolean function, which returns *true* if satisfaction is possible and *false* if it is not. A relation may have multiple means of satisfaction. An important property of relations is that they can be forced to "backtrack" over these multiple solutions until a satisfactory outcome is obtained. In particular, a while statement causes a relation to backtrack over all possible solutions.

Figure 7 shows the relation items for the classes List and ListElement, as well as an example illustrating how a while statement can be used to sum the items in a list. Basically stated, the while loop generates all the values which "satisfy" the relation items. When provided with an unbound variable, every item satisfies the relation. Thus all values will be generated, and in turn each will be added to the sum.

To see how the relation operates, consider first the definition of the relation in class List. This definition can be read as asserting that the relation items holds for the variable value if the field data is defined and if the relation items from the class ListElement is satisfied. The relation in class ListElement is more complex, and is described as the disjunction of two clauses. Thus there are two different ways in which this relation can be satisfied. Each approach is tried in turn in an attempt to satisfy the relation.

The first clause uses the *unification operator*, which appears somewhat like a double sided assignment operator. The unification operator is a boolean valued function that attempts to make the left and right arguments equal. If both are defined and are unequal, it fails. On the other hand, if only one argument is defined (which can be either the right or left argument), then the unbound argument is set equal to the other and the relation succeeds. A subsequent attempt to find a new solution undoes the assignment, and the relation will investigate the next clause. In this particular case the right argument to the unification operator, the variable datum, will always be defined. The left argument will generally be undefined, and thus the effect will be to assign the unbound variable denoted by the call-by-reference parameter value to an element of the list.

The second clause in the relation is a conjunction which asks first if the field next is defined, and if so tries to satisfy the relation by invoking recursively the items relation on the next field. Thus the recursive process will eventually assign to datum all the values in the list.

The important property of this approach is that the user of the relation, represented by the program in Figure 7, needs absolutely no knowledge of how the various alternatives are being generated. Knowledge of the internal structure of lists is not required for procedures to access each element of the list in turn.

Another important property of relations is that they can be used both as generators and as a means to test a property. In Figure 7 the relation is used as a generator. This depends upon the

```
relation List.items(var value : X);
begin
    return defined(data) and data.items(value);
end;

relation ListElement.items(var value : X);
begin
    return
        (value :=: datum) or
        (defined(next) and next.items(value));
end;

var
    aList : List(integer);
    sum, val : integer;
begin
    aList := List;
    aList.add(3);
    aList.add(7);
    ...

    sum := 0;
    while aList.items(val)
        sum := sum + val;
end
```

Figure 7: The relation items and an example of its use

```
method List.includes(value : X) → boolean;
begin
    if defined(data) then
        if data.items(value) then
            return true;
    return false;
end;
```

Figure 8: Testing membership in a list

fact that the argument to the generator is initially undefined, and is bound as part of the generation process. Passing an actual value to a relation yields success if the relation is satisfied with that value. In this case, the relation is satisfied if the value is present in the list. This fact is used by the method includes (Figure 8), which is used to test membership in a list.

We will return to relations in a later section to investigate further ways in which relational programming can be combined with object-oriented and functional styles of program development.

4 Functionals

In this section we investigate in more detail the problem posed by taking a list of integers and computing their sum. This could be easily solved by use of a while loop, such as that used in the earlier Figure 7. Having solved this one problem, hypothesize now that a need arizes for a routine to compute the product of the same list. Later there may be a need to count the elements which satisfy some property. It would be desirable if more than just the outline of the approach to solution could be used in each of these cases. That is, it would be nice if the actual code used in the solution could be reused.

To do this, it is first necessary to generalize the problem. We can characterize both the problems of taking the sum and product of a list by saying in each case we are provided with two objects.

1. An *identity* value which is used to "prime the pump" prior to iterating over the elements of the list. In the case of a sum this identity value is zero, while in the case of a product this identity value is one.[5] This identity may or may not be the same type as the elements in the list. If we continue to use X to represent the type of the elements in the list, let us use Z to represent the (unknown) type of the identity value.

2. A function which takes two values. The left value is of type Z and is initially the identity value. The right argument is of type X and ranges over the elements of the list. The function must yield a new value of type Z. As each element of the list is examined this function is invoked, yielding the new value to be used with the next list element. When all list values have been exhausted, the final value of this function is returned.

The method reduce, shown in Figure 9, takes these two quantities as arguments and generates the *reduction* of a list to a single element. Notice the body of the method uses the relational technique

[5] To be technically accurate, what we require is a left-identity value.

described in the last section to access each element of the list in turn. A method or function, such as reduce, which takes a function as argument or yields a function as a result, is called a *functional* (sometimes a *higher-order* function). Programs which make extensive use of functionals are often concise and elegant. Such a style of programming is called *functional programming* [Backus 78, Wikstrom 87].[6]

Programming with functionals is facilitated by the ease with which other program structures, such as operators, methods and relations, can be converted to functions. In Figure 9 the built-in operators + and * are converted to functions and passed as arguments to reduce, thereby yielding the sum and product of a list. User defined functions can also be used. Figure 9 shows the definition of a function which takes a count and a list element and returns an updated count which is one greater if the list element is positive. Reducing using this function will indicate the number of positive values in a list.

When a method is used in a context in which a function is required, such as being passed as argument to a function, the method is coverted into a function by adding the implicit receiver as the first argument to the function. Figure 9 shows this behavior, illustrating how the function reduce can be used to append one list to the end of a second. The first list is used as the "identity" element. The function passed as the second argument is the addNew field of the class List. Since the variable aList is of type List(integer), the function aList.addNew is converted into a function which takes as first argument a list of integers, an integer as the second argument, and yields as result a new list of integers. The reader can verify that the result of this reduction will be a new list in which values of the two original lists haven been appended together.

A second functional we will describe is the method count, shown in Figure 10. This method takes as argument a function which yields a boolean result, and returns the count of the number of elements in the list for which the function is satisfied. While we could have implemented count using reduce, the description of an alternative way to implement this method will help to illustrate a few more features of Leda. Instead of using relations, the alternative technique makes use of the method onEach, which is found in the class ListElement. This method takes a procedure (a function which does not yield a result), and executes it on each element of the list. The method count constructs dynamically a nameless function which is passed as argument to the onEach method. This function as a side effect alters the value of the variable sum. Features to note are that functions can be written as expressions, and that nested functions capture the environment in which they are defined.

One of the most important features of Leda is the fact that by supporting multiple programming styles we have a framework in which various approaches to problem solution, such as these two techniques of implementing higher-order functions, can be compared and contrasted for features such as ease of use, efficiency, and understandability.

[6] We are actually being somewhat loose here. Pure functional programming also requires an absence of functions which produce side effects. This requirement is not enforced by the Leda system. In fact, it has been our observation that at the lowest level functions which use side effects to achieve their objective are often more efficient when programming in a functional style. This supports our basic tenet that a multiparadigm language can provide benefits to all styles of programming.

```
method List.reduce(ident : Z, f : function(Z, X) → Z) → Z
var value : X;
begin
    if defined(data) then
        while data.items(value)
            ident := f(ident, value);
    return ident;
end;

function posCount(count : integer, element : integer) → integer
begin
    if element > 0 then
        count := count + 1;
    return count;
end;

var
    aList : List(integer);
    bList : List(integer);
    cList : List(integer);
    sum, prod, positiveCount : integer;
begin
    aList := List;
    aList.add(3);
    aList.add(7);
    bList := List;
    bList.add(6);
    bList.add(13);
    ...

    sum := aList.reduce(0, +);
    prod := aList.reduce(1, ∗);
    positiveCount := aList.reduce(0, posCount);
    ...
    cList := aList.reduce(bList, aList.addNew);
    ...
end
```

Figure 9: The functional reduce and examples of its use

```
method List.count(f : function(X) → boolean) → integer;
var sum : integer;
begin

   sum := 0;
   if defined(data) then
      data.onEach(function(val: X);
            begin
               if f(x) then sum := sum + 1;
            end );
   return sum;
end;
```

Figure 10: A functional implemented without relations

5 Subclasses and Parameterized Classes

Leda supports the specialization of classes through inheritance. Currently only single inheritance is supported. We have found that the majority of problems which are solved using multiple inheritance in other languages can often be addressed more easily using parameterized classes and single inheritance.

Figure 11 shows two instances of classes being defined using subclassing. In the first case the parameterized class List is subclassed in order to provide a binding for the parameter, yielding the class IntegerList. This class can then provide integer-specific operations, such as a method to return the sum of the list.

The second example illustrates the fact that the unbound parameter from a parameterized class must be bound before a class can be subclassed. However, a programmer is free to create a new parameter for the subclass. The parameterized class Set is generated by subclassing from the class List, modifying only the method used to add a new element to the collection.

6 Tables

The *table* data structure, sometimes called a *dictionary*, can be thought of as a generalization of an array. Like an array, a table is a collection of key and value pairs. However, unlike an array, the key values can be arbitrary, and the table need not have a fixed declared size. There are many different techniques that can be used to implement tables. The approach we will use here is designed to illustrate features of the language Leda, and is not necessarily the most efficient approach when measured by access or insertion time.

As was the case with lists, our construction of tables will be built on top of another utility class that is manipulated only by the methods in class Table, and never directly by the user. In this case the utility class is called Association (Figure 12). An association maintains a key-value pair. The class Association has no behavior, only data. Thus the class in this case is being used as a record structure. The table itself will maintain data as a list of associations.

```
type
    IntegerList := class of List(integer)
        shared
            sum : method() → integer;
        end;

    Set := class(Y) of List(Y)
        shared
            add : method(Y);
        end;

method Set.add(value : Y);
begin
    if not includes(Y) then
        List.add(self, value);
end;
```

Figure 11: Subclassing in Leda

The method add is used to add a new value to the table. The relation items is used, as was the case with the similarly-named relation in class List, to access the values of the relation. In fact the implementation of this relation simply invokes the relation from the list class, then uses the unification operator to break apart the elements yielded by the list.

The use of relations for accessing elements within a table implies that retrival is not restricted to be by key alone. If the variable tabl contains a table and x is a currently undefined value, the expression tabl.items(3, x) is a boolean expression which returns true and, as a side effect, sets the value of x to the entry (or first entry) associated with the key 3. It will return false if there is no such entry. Alternatively, the expression tabl.items(x, 4) will generate the key value associated with the value 4. Thus, tables can be used as *associative arrays* indexing by either the first or second field.

7 Relational Objects

In previous sections we have outlined how relational programming techniques can be used in support of problems solved in a more-or-less object-oriented style. In this section we return to problems formulated in a relational style, and consider how some of the object-oriented data structures we have developed can be of benefit to relational programs.

A feature of the logical programming language Prolog that is not found in Leda is the ability to dynamically assert a relation at run-time. For example, suppose that parent is a database of genealogical information, as in Figure 13. During execution a different relation, such as the new-Daughter relation shown in Figure 13, can assert an additional fact about the relation parent. This new fact is added to the database, and is thereafter used just as if it were asserted in the original database.

```
type
    Association := class(X, Y)
            key : X;
            value : Y;
        end;

    Table := class(X, Y)
            data : List(Association(X, Y));
        shared
            add : method(X, Y);
            items : relation(var X, var Y);
        end;

method Table.add(key : X, value : Y);
begin
    data.add(Association(key, value));
end;

relation Table.items(var key : X, var value : Y);
var item : Association(X, Y);
begin
    return data.items(item) and
        (item.key :=: key) and (item.value :=: value);
end;
```

Figure 12: The implementation of the table data structure

```
parent(leda, helen).
parent(zeus, helen).
parent(leda, castor).
parent(zeus, pollux).
...
male(zeus).
female(leda).
female(helen).
...
mother(X, Y) :- parent(X, Y), female(X).
newDaughter(X, Y) :- assert(parent(X, Y)), assert(female(Y)).
```

Figure 13: A dynamically changing database in Prolog

In Leda a dynamically changing relation can be modelled using a data structure, such as the table data structure we defined in the last section. Figure 14 illustrates this technique. Here `names` is an enumerated datatype, the most commonly used substitute for Prolog symbols. (Although, unlike Prolog, relations in Leda can manipulate any type of value). Parent is declared as a table of names, and the initial database is constructed by `add`ing elements to the table. Subsequent additions, such as that used in the `newDaughter` function, as also performed using the `add` method.

8 Concluding Remarks

Our intent in this paper has been two-fold. First, our desire has been to demonstrate by example exactly what we mean by the frequently-used but seldom-defined term *multiparadigm programming*, and demonstrate that such an idea is not only possible but powerful and useful. Second, by means of this example we wish to illustrate the features of the programming language Leda which we are developing.

Our basic tenet is that imperative programming, object-oriented programming, functional programming, and logic programming are all useful abstraction techniques, and each shines in different problem domains. More importantly, each can benefit by the use of facilities provided by the others. By providing a simple integrated framework in which each is supported, we supply the programmer with the broadest possible assortment of tools with which to address any given problem.

A danger in this approach is the "kitchen-sink" syndrome. This is where a language attempts to provide all features to all programmers, and ends up being so large that no programmer can ever hope to master the entire language. Instead, Leda is a surprisingly simple language, at least when measured by the complexity of its grammar. Our objective has been to produce a language of approximately the same complexity as Pascal.

The list and table data structures we have developed in this paper use features of object-oriented programming for their basic structure, but important elements are provided using relational and functional techniques. While this is certainly not the only way in which these data structures can be implemented (even in Leda), the design here illustrates how a programmer can easily integrate programming techniques from a number of different language paradigms.

References

[Backus 78] John Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", *Communications of the ACM*, Vol 21(8): 613-641, (August 1978).

[Berztiss 90] Alfs Berztiss, *Programming with Generators*, Ellis Horwood, 1990.

[Budd 87] Timothy A. Budd, *A Little Smalltalk*, Addison-Wesley, 1987.

[Budd 89] Timothy A. Budd, Functional Programming in an Object Oriented Language, Oregon State University, Technical Report 89-60-16, August 1989. *submitted for publication.*

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

relation mother(mom : names, kid : names);
begin
    return parent.items(mom, kid) and female.items(mom);
end;

function newDaughter(prnt : names, kid : names);
begin
    parent.add(prnt, kid);
    female.add(kid);
end;

var
    parent : Table(names, names);
    female : List(names);
    male : List(names);

begin
    parent := Table;
    male := List;
    female := List;
    ...
    parent.add(leda, helen);
    parent.add(zeus, helen);
    parent.add(leda, castor);
    parent.add(zeus, pollux);
    ...
    male.add(zeus);
    female.add(leda); female.add(helen);
    ...
    newDaughter(helen, hermione);
end;
```

Figure 14: A dynamically changing database in Leda

[Budd 91a] Timothy A. Budd, "Blending Imperative and Relational Programming", *IEEE Software*, Vol 8(1): 58-65, January 1991.

[Budd 91b] Timothy A. Budd, *An Introduction to Object-Oriented Programming*, Addison-Wesley, 1991.

[Budd 91c] Timothy A. Budd, "Sharing and First Class Functions in Object-Oriented Languages", submitted for publication.

[Griswold 90] Ralph E. Griswold and Madge T. Griswold, *The Icon Programming Language*, Prentice-Hall, 1990.

[Liskov 81] Barbara Liskov, Russel Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*, Springer-Verlag, 1981.

[Parnas 72] David L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules", *Communications of the ACM*, 15(12): 1059-1062, 1972.

[Wikstrom 87] Åke Wikström, *Functional Programming Using Standard ML*, Prentice-Hall International, 1987.

[Who56] Benjamin Lee Whorf, *Language, Thought & Reality*, MIT Press, Cambridge, 1956.