Multiparadigm Programming in Leda

Tim Budd

Oregon State University
Department of Computer Science
Corvallis, Oregon
USA

Paradigm – Definition

We start by considering the definition of the term "paradigm":

> **par·a·digm** *n.* **1**. A list of all the inflectional forms of a word taken as an illustrative example of the conjugation or declension to which it belongs. **2.** Any example or model. [Late Latin *paradīgma*, from Greek *paradeigma*, model, from *paradeiknunai*, to compare, exhibit.]

What in the world does this have to do with computer programming languages?

The Power of Metaphor

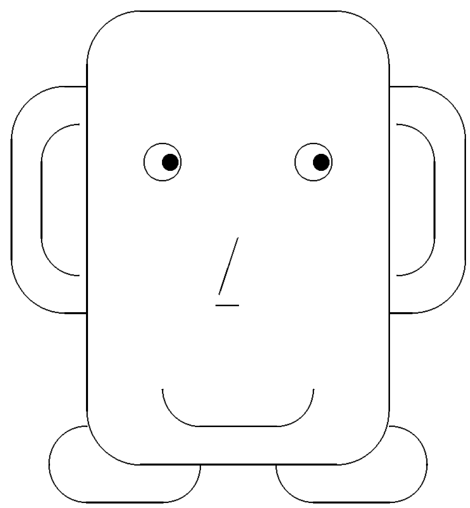
Most people solve on their own the vast majority of problems they experience in everyday life.

People bring with them everyday knowledge when they confront other, artificial problems.

There is a vast power in finding the right *metaphor* that makes a problem easier to understand.

From Newsweek

“Unlike the usual programming method – writing software one line at a time– NeXT’s “object-oriented”’ system offers larger building blocks that developers can quickly assemble the way a kid builds faces on Mr. Potato Head.”



Sapir – Whorf Hypothesis

In linguistics there is a hypothesis that the language in which an idea is thought or expressed colors or directs in a very emphatic manner the nature of the thought.

- Eskimo languages and snow.
- Arabic languages and camels.

What is true of natural languages is even more true of artificial computer languages.

An Example from Computer Languages

Problem : given a very long array of numbers, see if any section of length M is repeated (comes from DNA research).

```
    DO 10 I = 1, N−M
       DO 10 J = 1, N−M
          FOUND = .TRUE.
          DO 20 K = 1, M
20           IF X[I+K−1] .NE. X[J+K−1] THEN FOUND = .FALSE.
          IF FOUND THEN ...
10  CONTINUE
```

A Better Idea – found in APL

An APL programmer was able to solve the problem much more quickly. The question is, why was the programmer led to this solution?

x_1	x_2	...	x_m
x_2	x_3	...	x_{m+1}
...			...
x_{n-m}	...	...	x_{n-1}
$x_{n-(m-1)}$	...	x_{n-1}	x_n

Can Programming be Multiparadigm?

If a paradigm is a way of looking at the world, can a programmer have multiple paradigms? Is multi-paradigm programming possible?

Paradigms provide the tools a programmer uses in the solution of a problem. Certainly a programmer should have a rich set of tools at his or her disposal.

Think of the varieties of tools a mathematician can use to solve a problem. (algebraic, geometric, logical, constructive, and so on).

The Paradigms in Leda

In Leda we have targeted for inclusion four of the more common programming paradigms:

- Imperative Programming (the value-state model).
- Object-Oriented Programming (recursive design)
- Functional Programming (the transformational model)
- Logic Programming (the solution space search model)

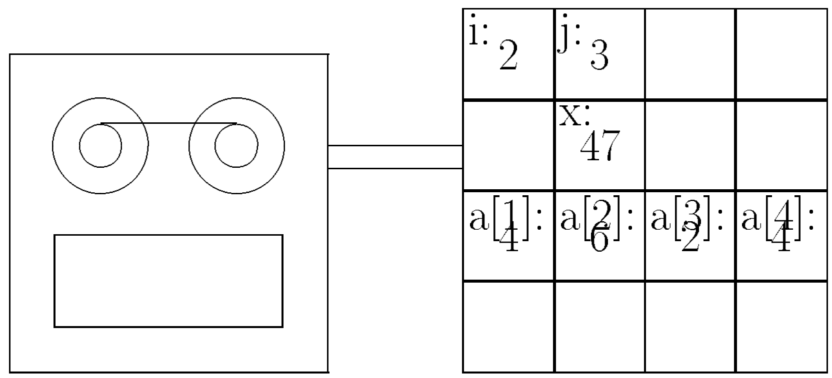
Imperative Programming

Imperative Programming is the "traditional" model of computation.

- State
- Variables
- Assignment
- Loops

A processing unit is separate from memory, and "acts" upon memory.

Visualization of Imperative Programming



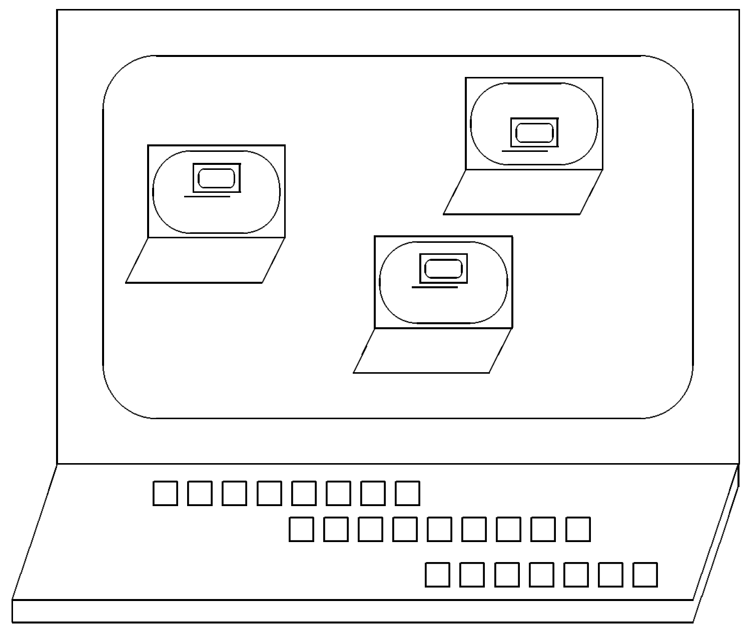
Sometimes called the “pigeon-hole” model of computation.

Object-Oriented Programming

Object-oriented programming is based on the principle of recursive design.

1. Everything is an object.
2. Objects perform computation by making requests of each other through the medium of messages.
3. Every object has it's own memory, which consists of other objects.
4. Every object is an instance of a class. A class groups similar objects.
5. The class is the repository for behavior associated with an object.
6. Classes are organized into a singly-rooted tree structure, called an inheritance hierarchy.

Recursive Design



The structure of the part mirrors the structure of the larger unit.

Service View of Computation

An object is a component charged with performing a specific service.

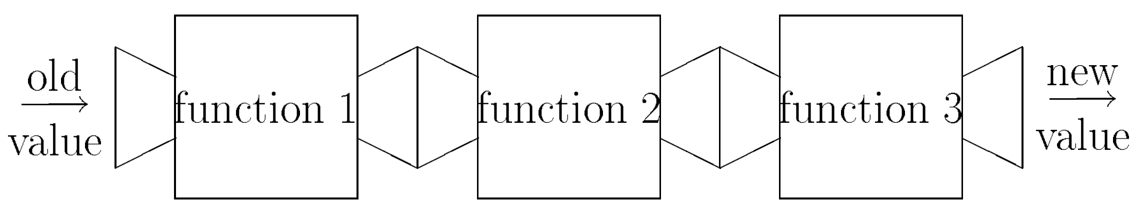
It is not necessary to know *how* this service is performed in order to make use of the service.

"Ask not what you can do *to* your data structures, but ask what your data structures can do *for* you."

Functional Programming

Functional Programming can be characterized as follows:

- Values are treated as single entities, not "boxes with slots". Values can be created, but once created, are never modified.
- Computation is performed by applying functions to values, and recursively by applying functions to the results.
- Functions are first-class data values, and can be assigned to identifiers, passed as arguments, or returned as the result of executing other functions.



Logic Programming

Logic Programming views computation as a search of a space of potential solutions for a single desired answer.

Grew out of work on automatic theorem proving. In theorem proving there are three parts:

- facts, or assumptions.
- rules of inference.
- questions, or queries.

A typical theorem

Here is the use of propositional calculus in the proving of a typical theorem.

Question: Did the Beavers beat the Huskies?

if the Beavers beat the Ducks **or** the Beavers beat the Huskies **then** the Beavers will be conference champions	*assumption*
not(The Beavers are conference champions)	*assumption*
not(Beavers beat the Ducks **or** Beavers beat the Huskies)	*modus tollens* P $\Rightarrow$ Q and not Q then not P
not(Beavers beat the Ducks) **and** **not**(Beavers beat the Huskies)	*DeMorgans Rule* not (A or B) $\equiv$ (not A) and (not B)
not Beavers beat the Huskies	*simplification*

Declarative Programming

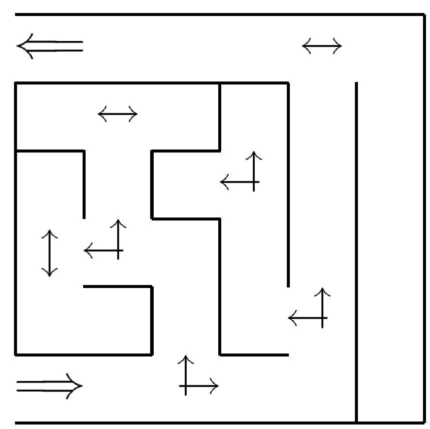
What is most important about logic programming is that it is *declarative*.

The programmer specifies (declares) only facts and rules of inference. The search for a solution, which may be quite involved, is performed behind the scene by a "search engine".

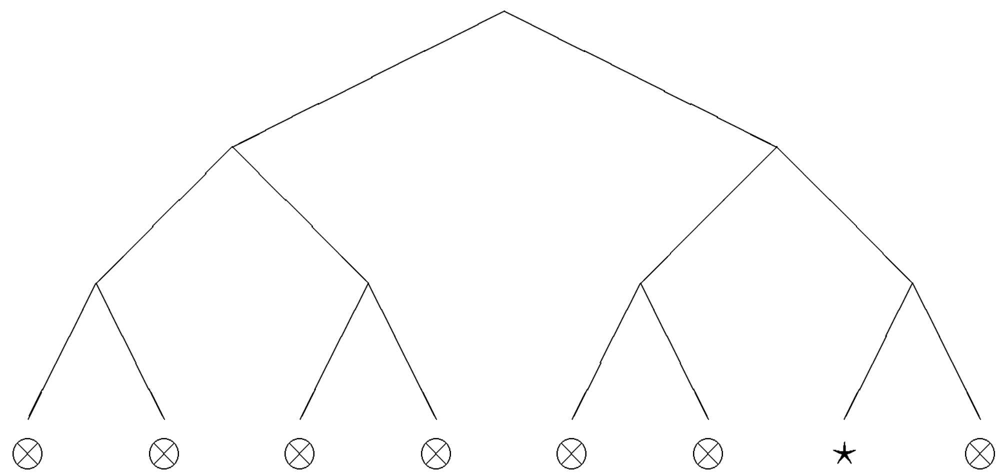
As with most logic programming languages since Prolog, Leda uses a depth-first search procedure to perform backtracking.

A Problem Requiring Backtracking

A maze is a typical problem that involves backtracking in the discovery of a solution.



Visualization of Backtracking



We will use the backtracking feature of Leda whenever a problem can have multiple solutions.

Remainder of Talk

- Illustrate true multiparadigm programming in Leda
- Show how each of the paradigms contributes to the solution of a problem
- We will use classes for graph problems as our illustration

Graphs – Definition

Definition. A *graph* $G = (V, E)$ consists of a finite, nonempty set of *vertices* V and a set of *edges* E. If the edges are ordered pairs (v, w) of vertices, then the graph is said to be *directed;* v is called the *tail* and w the *head* of the edge (v, w). If the edges are unordered pairs (sets) of distinct vertices, also denoted by (v, w), then the graph is said to be *undirected.*

Class Definition for Graphs

```
class Graph;
var
    vertexSet : Set[integer];
    edgeSet : Set[Edge];

    function addVertex(newVal : integer)→Graph;
        ...
    function includesVertex(val : integer)→boolean;
        ...
    function includesEdge(t : integer, h : integer)→boolean;
        ...
    function addEdge(newStart : integer, newEnd : integer)→Graph;
        ...
    function removeVertex(val : integer)→Graph;
        ...
    function removeEdge(tail : integer, head : integer)→Graph;
```

```
    ...
  function vertex(byRef val : integer)→relation;
    ...
  function edge(byRef tail : integer, byRef head : integer)→relation;
    ...
  function inDegree(aVertex : integer)→integer;
    ...
  function outDegree(aVertex : integer)→integer;
    ...
  function degree(aVertex : integer)→integer;
    ...
  function path(byRef i : integer, byRef j : integer)→relation;
    ...
  function display();
    ...
end;
```

Class Definition of Sets

```
class Set [X : equality];
var
    value : X;
    next : Set[X];

    function includes(val : X)→boolean;
        ...

    function remove(val : X)→Set[X];
        ...

    function items(byRef val : X)→relation;
        ...
end;
```

Adding an Element to a Set

```
function addSet [X : equality] (aSet : Set[X], newVal : X)→Set[X];
begin
    if defined(aSet) & aSet.includes(newVal) then
        return aSet;
    return Set[X](newVal, aSet);
end;
```

Inclusion Test

```
class Set [X : equality];
    ...

    function includes(val : X)→boolean;
    begin
        if value = val then
            return true;
        if defined(next) then
            return next.includes(val);
        return false;
    end;
```

Removal from a Set

```
class Set [X : equality];
    ...

    function remove(val : X)→Set[X];
    begin
        if val = value then
            return next;
        if defined(next) then
            return Set[X](value, next.remove(val))
        else
            return self;
    end;
```

Set Iteration – using Relations

```
class Set [X : equality];

    ...

    function items(byRef val : X)→relation;
    begin
        return unify[X] (val, value)
            | defined(next) & next.items(val);
    end;

end;
```

Unification Function

The unification function attempts to make the two arguments the same:

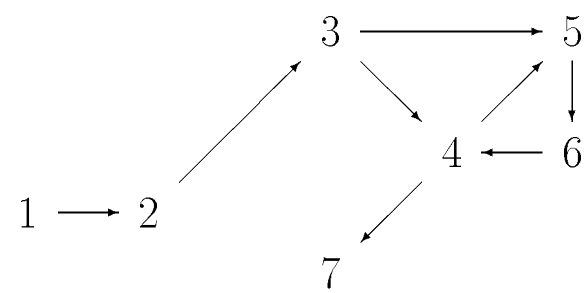
```
function unify [T : equality] (byRef left : T, right : T)→relation;
begin
    if defined(left) then
        return left = right
    else
        return left ← right;
end;
```

Relations Used for Loops

Relations can be used in for statements to form loops

```
for aSet.items(val) do
    ... some statement
```

An Example Graph



Commands to make Example Graph

```
var
    aGraph : Graph;
    i : integer;
begin
    aGraph := Graph();
    for i := 1 to 7 do
        aGraph := aGraph.addVertex(i);
    aGraph := aGraph.addEdge(Edge(1, 2));
    aGraph := aGraph.addEdge(Edge(2, 3));
    aGraph := aGraph.addEdge(Edge(3, 4));
    aGraph := aGraph.addEdge(Edge(4, 5));
    aGraph := aGraph.addEdge(Edge(3, 5));
    aGraph := aGraph.addEdge(Edge(5, 6));
    aGraph := aGraph.addEdge(Edge(6, 4));
    aGraph := aGraph.addEdge(Edge(4, 7));
```

The Class Edge

```
class Edge of equality[Edge];
var
    tail : integer;
    head : integer;

    function equals (arg : Edge)→boolean;
    begin
        return tail = arg.tail & head = arg.head;
    end;
end;
```

Adding a Vertex / Adding an Edge

```
function addVertex(newVertex : integer)→Graph;
begin
    return Graph(addSet[integer](vertexSet, newVertex), edgeSet);
end;

function addEdge(newEdge : Edge)→Graph;
begin
    if includesVertex(newEdge.tail) & includesVertex(newEdge.head)
    then
        return Graph(vertexSet, addSet[Edge](edgeSet, newEdge))
    else
        return self;
end;
```

Inclusion test for Vertex / Edge

```
class Graph;
    ...

    function includesVertex(val : integer)→boolean;
    begin
        return defined(vertexSet) & vertexSet.includes(val);
    end;

    function includesEdge(anEdge : Edge)→boolean;
    begin
        return defined(edgeSet) & edgeSet.includes(anEdge);
    end;
    ...
end;
```

Remove Vertex

```
function removeVertex(val : integer)→Graph;
var
    newGraph : Graph;
    anEdge : Edge;
begin
        { make a new graph with no edges and vertex removed}
    newGraph := Graph(vertexSet.remove(val), NIL);

        { add edges, only those with new vertices will be added}
    if defined(edgeSet) then
        for edgeSet.items(anEdge) do
            newGraph := newGraph.addEdge(anEdge);
    return newGraph;
end;
```

Remove an Edge

```
function removeEdge(anEdge : Edge)→Graph;
begin
    if defined(edgeSet) then
        return Graph(vertexSet, edgeSet.remove(anEdge))
    else
        return self;
end;
```

A Relation for Vertices

```
function vertex(byRef val : integer)→relation;
begin
    return defined(vertexSet) & vertexSet.items(val);
end;
```

Can be used for both testing and for iteration

A Relation for Edges

```
function edge(byRef tail : integer, byRef head : integer)→relation;
var
    anEdge : Edge;
begin
    return
        defined(edgeSet) & edgeSet.items(anEdge)
            & unify[integer](tail, anEdge.tail)
            & unify[integer](head, anEdge.head);
end;
```

Finding in-edges

The following finds the vertices which have in-edges leading to vertex 4.

```
x := NIL;
for aGraph.edge(x, 4) do
    print(x);
```

A Procedure for in-degree

```
function inDegree(aVertex : integer)→integer;
var
    head : integer;
    count : integer;
begin
    count := 0;
    head := aVertex;

    for edge(NIL, head) do
        count := count + 1;

    return count;
end;
```

A Relation for Paths

```
class Graph;
    ...

    function path(byRef i : integer, byRef j : integer)→relation;
    var
        k : integer;
    begin
        return vertex(i) & vertex(j) &
            (((i = j)
            | edge(i, j)
            | (edge(i, k) & removeVertex(i).path(k, j))));
    end;

    ...
end;
```

Finding Paths

```
i := NIL;
for aGraph.path(3, i) do begin
    print("vertex");
    print(i);
    print("can be reached from vertex 3\n");
end;

for aGraph.path(i, 5) do begin
    print("vertex");
    print(i);
    print("can reach vertex 5\n");
end;
```

Finding Cycles

```
function cycle(aGraph : Graph, byRef v : integer)→relation;
var
   w : integer;
begin
      { cycle if edge from v to w, and }
      { path back from w to v }
   return aGraph.edge(v, w) & aGraph.path(w, v);
end;
```

Finding a Hamilton Path

```
function hamiltonPath(g : Graph,
    byRef start : integer, byRef path : intList)→relation;
var
    tail : integer;
    newGraph : Graph;
begin
    return
        g.vertex(start) & newGraph ← g.removeVertex(start) &
            (( ~ defined(newGraph.vertexSet)
                & path ← intList(start, NIL))
            | (g.edge(start, tail) &
                hamiltonPath(newGraph, tail, path) &
                path ← intList(start, path)));
end;
```

Conclusions

- Overall structure is object-oriented
- Operations which make changes to graph are functional
- Searching and iteration provided using relations
- All four paradigms are useful in providing short and clear solutions to problems
- The seamless integration provides benefits that could not be realized by any paradigm by itself
- OOP has become “part of the air we breathe”. It is now time to elevate the other paradigms to similar status.