Chapter 3

Combining Paradigms – Graphs

In the previous chapter we illustrated how Leda could be used to write programs in the style of each of the four paradigms described in Chapter 1. In this chapter we will illustrate the concept of *multi*-paradigm programming in Leda, by showing how each of these different programming styles can be made to work in combination, and can be productively employed in the creation of a useful, general purpose data abstraction. We will illustrate how multi-paradigm techniques can be applied not only at the highest levels of abstraction (dividing a problem into a functional part, an imperative part, an object-oriented part, and so on), but can usefully be employed at every level of abstraction. Our vehicle for this exposition will be a data structure for the representation and manipulation of graphs.

3.1 Graphs

The following definition of a graph is provided by Aho, Hopcroft and Ullman [Aho 74]:

> **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.*

Since a graph maintains state (that is, possesses internal data values, namely the set of vertices and the set of edges), we choose to represent a graph using a class structure. The simplified class definition shown in Figure 3.1 closely mirrors the formal definition given above. We will use integer values to represent vertices. The two data fields for a graph represent a set of vertices, and a set of edges. The class `Set`, used to maintain these two data values, is a parameterized class, the first we have seen. Like the parameterized functions introduced in the previous chapter, a parameterized class permits a more general structure to be developed, since we need not know at the time the class `Set` is created the type of values that it will contain.

The class type `Set` represents an abstraction of the mathematical concept of a set (an unordered collection of distinct values). Notice that graphs maintain two types of sets. The first data field in a graph is responsible for maintaining the vertex values, represented by a set of integers. The second data field is again a set, this time maintaining values of type `Edge`. Because the `Set` class is parameterized, a single class can be used to define both types of data structures. Since the various graph operations are all implemented in terms of operations on these underlying sets, before we can describe the implementation of the graph methods we must first illustrate how sets are implemented.

3.2 Simple Sets

In a later chapter we will explore in detail some of the various possibilities in implementing a set. Our interest here will simply be to construct a trivial data structure that provides the minimal functionality we require for the implementation of graphs. The `Set` data type (Figure 3.2) is similar to the linked list data type we constructed in the last chapter. However, unlike the `intList` data type, the class data structure for sets is parameterized. This permits sets to hold arbitrary types of objects, not just integers as in the list class.

The type parameter `X` is restricted to be a type that is a subclass of the class `equality`. We subsequently have much more to say about the class–subclass relationship. For the moment, it is sufficient to simply say that this implies that instances of class `X` must know how to compare against each other for equality.

Within the class definition for `Set`, and within any method defined in association with this class, the variable `X` can be used to represent a type. Variables or parameter values can be declared using this type, as is done with the `value` field, or with the parameters to each of the various methods. The principle way in which a type argument differs from a true type (or class) identifier is that it is not allowed to dynamically create new instances of values using a type argument.

An empty set is represented by an undefined object, just as an empty list was represented in the previous chapter. To add an item to a set, as performed by the function `addSet` (Figure 3.3), requires checking a number of conditions. If the set is empty, then the constructor can simply be invoked to create the new structure. If, however, the set is not empty then it is only if the set does not already include the new element that it is added to the set. The test for inclusion is performed by the method `includes`, to be described shortly. If the element is already part of the set, then the current set is simply returned.

Because the class `Set` is parameterized, the constructor function that is automatically generated for each class must be similarly parameterized. We see this in the call on the

```
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;
```

Figure 3.1: Class Definition for Graph

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

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

    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;

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

Figure 3.2: Class Definition for Sets

```
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;
```

Figure 3.3: Implementation of the Addition function in class Set

constructor in the function `addSet` (which is itself a type-parameterized function).

The inclusion test is performed by a method, (named `includes` in Figure 3.2) and can therefore be performed only for sets that possess at least one element. The function used to implement the inclusion test is recursive. It tests the current link, and if it matches the argument value then the element is known to be in the set. Otherwise, if there are further elements in the set then a recursive call is performed. If there are no further links and if the value does not match the current link, then the element is known not to be in the set and a value of `false` can be returned.

The method to remove an element from a set (named `remove` in Figure 3.2) is similarly implemented using a recursive function. If the element being deleted matches the current value, then the next link in the sequence is returned. Otherwise, if there is a next link then a new set is generated with the current value, and the set formed by removing the argument value from the remainder of the set. Finally, if the value does not match the argument value and there is no next value then the current link can be returned. To do this, the pseudo-variable `self` is used. This variable can be used inside of any method, and always refers to the "current" object; the object on which operations are being performed.

Notice that both the addition function and the removal method are functional in nature, in that they construct a new structure that possesses the desired characteristics (adding or deleting elements), without altering the original structure. This property will turn out to be crucial in the use of sets for the implementation of graphs.

3.2.1 Set Iteration

The final set operation named in the class description near the bottom of Figure 3.2 is slightly more complex, and illustrates how relational operations can be usefully combined with the other paradigms.

The function `items` defines a relation, which takes as parameter a single argument passed by reference. If this value is defined, then the relation simply returns a true or false value indicating whether the element is in the set or not, and thus duplicates the effect of the function `includes` which we have already seen.

A more useful effect occurs when the argument value is not defined. In this case, the parameter value is then bound (assigned) to the first element in the set. This binding is accomplished in the function `unify`, which attempts to ensure the two arguments which are passed to the function have the same value (that is, it unifies the two values). If both arguments are defined they they are the same only if they are equal. If the first argument is not defined, however, they can be made equal by assigning the left argument the value of the right:

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

```
    else
        return left <- right;
end;
```

This assignment is performed using the relational assignment operator, written as `<-`. An important property of relations that we did not explore in the logic program presented in the last chapter is the relational search facility, provided as part of the `for` statement. Used within a `for` statement, a relation will find in sequence *every* binding for variables. Thus, if `aSet` represents a set of values, and `val` is a previously undefined variable, then the following statement will iterate over every value in the set, assigning each to the variable `val` in turn:

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

The mechanism used by the backtracking facility operates through a combination of the `for` statement, the `relation` datatype, and the *or* operator. When used with relations, the or operator suspends control when the first clause is true, and therefore sufficient to determine that the result should be true. The `for` statement continually resumes execution as long as there is a valid suspension, which has the effect of then executing the second clause in the or expression. The failure of a subsequent boolean expression can also cause the reevaluation of a suspended expression, as the backtracking mechanism tries to discover a set of values which will make an expression evaluate true. The details of this implementation will be explained more fully in Chapter 22.

Note that this mechanism can be used to iterate over the elements in a set, yet does not require the programmer using the set to have knowledge of the representation of the set (as would be required if, for example, the programmer explicitly wrote a loop to traverse the link fields).

3.3 Basic Graph Operations

Having described the functionality provided for us by the `Set` data type, we are now in a position to return to our `Graph` data structure and illustrate the implementation of each of the various basic operations. As with the `Set` data structure, these operations tend to be functional in nature. That is, rather than making *changes* to an existing graph, they generate a new graph with the requested properties.

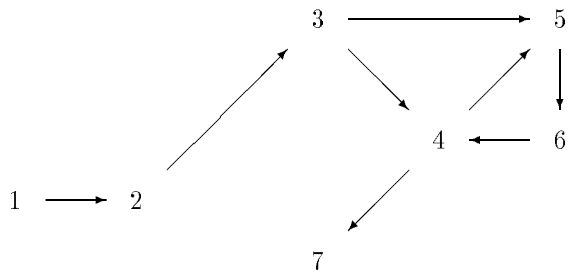
The class definition for edges (Figure 3.4) defines only one function; to test equality between two edges. Two edges are judged to be equal if their head and tail fields are the same. In Leda all operators have an associated textual name, and new meanings are provided for operator symbols by defining functions which use this textual name. A table of operators and their textual names is given in Appendix B.

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

    function equals (arg : Edge)->boolean;
    begin        { two edges are equal if their fields are equal }
        return tail = arg.tail & head = arg.head;
    end;
end;

class Graph;
    ...

    function addVertex (newVertex : integer)->Graph;
    begin        { add a new vertex to a graph }
        return Graph(addSet[integer](vertexSet, newVertex), edgeSet);
    end;

    function addEdge (newEdge : Edge)->Graph;
    begin        { add new edge only if both vertices are in graph }
        if includesVertex(newEdge.tail) & includesVertex(newEdge.head)
        then
            return Graph(vertexSet, addSet[Edge](edgeSet, newEdge))
        else
            return self;
    end;

    ...
end;
```

Figure 3.4: Implementation of Graph Addition Operations

The implementation of the functions to add new fields to a graph is simplified by the use of the functionality provided by the underlying sets. To add a vertex to a graph it suffices to simply form a new graph with a vertex set including the new element, and the existing edge set. This is shown in the method named `addVertex` in Figure 3.4. A new edge is added, (function `addEdge`), only if both vertices are included in the edge set and if the edge is not already part of the current graph. If any of these three conditions fail, then the current graph is returned.

To see how these operations can be used to construct a representation of a graph, consider the following graph:



A graph to represent this picture could be generated as shown below. Note that the constructor for the class Graph employs an alternative form we have not seen before, in which no arguments are used. In this situation the data fields in the newly constructed object are initially undefined (that is, have value `NIL`).

```
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));
```

```
class Graph;
    ...

    function includesVertex (val : integer)->boolean;
    begin         { does graph include this vertex ? }
        return defined(vertexSet) & vertexSet.includes(val);
    end;

    function includesEdge (anEdge : Edge)->boolean;
    begin         { does graph include this edge ? }
        return defined(edgeSet) & edgeSet.includes(anEdge);
    end;

    ...
end;
```

Figure 3.5: Implementation of Graph Inclusion Test Operations

The next operation we will consider in the implementation of our graph data type is the test used to determine if a graph contains either a vertex or an edge between two vertex values. Both of these two functions (Figure 3.5) can be implemented by simply invoking the inclusion test for sets, once it is determined that the appropriate set is non-empty.

The removal procedures for vertices and edges (Figure 3.6) could also have been easily implemented using the functionality provided by the underlying sets. However, we have chosen to provide a slightly more complex meaning to the operation of deleting a vertex. If a vertex is removed from a graph, then all edges which connect with the vertex are similarly removed. To achieve this, a new graph is first constructed with a vertex set consisting of the original vertex set minus the specified vertex, and an empty edge set. Each edge of the original graph is then added to the new graph. The nature of the edge add operation ensures that only those edges for which the deleted vertex is neither head nor tail will be inserted.

The final two basic operations (Figure 3.7) are used to iterate over elements of a graph. The relation named `vertex` will, when passed an undefined value, return in sequence all the vertices in a graph. In a similar manner the relation named `edge`, which takes two by-reference parameters, will yield the head and tail of each edge in turn. More interestingly, if either one of the arguments to `edge` is defined, while the other is undefined, the relation will return all the values that can be substituted for the undefined value to yield a valid edge. For example, the following statement will print the names of each edge that lead into vertex number 4 in the graph shown in Figure 3.5.

```
x := NIL;
```

```
class Graph;
    ...

    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;

    function removeEdge (anEdge : Edge)->Graph;
    begin          { remove edge from graph }
        if defined(edgeSet) then
            return Graph(vertexSet, edgeSet.remove(anEdge))
        else
            return self;
    end;
end;
```

Figure 3.6: Further Basic Operations on Graphs

```
class Graph;
    ...

    function vertex (byRef val : integer)->relation;
    begin        { iterate over all vertex values }
        return defined(vertexSet) & vertexSet.items(val);
    end;

    function edge (byRef tail : integer, byRef head : integer)->relation;
    var
        anEdge : Edge;
    begin        { iterate over all edge values }
        return
            defined(edgeSet) &
            edgeSet.items(anEdge)
                & unify[integer] (tail, anEdge.tail)
                & unify[integer] (head, anEdge.head);
    end;

    ...
end;
```

Figure 3.7: Implementation of Iteration Methods for Graphs

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

We will use this property of the `edge` relation in the methods generated in the next section.

3.4 More Graph Methods

In this section we will construct several methods that can be used with graph structures. The implementation of these methods will be simplified through the use of the basic methods we described in the previous section.

3.4.1 In-degree and Out-degree

The observation we made at the end of the last section concerning the relation named `edge` can be used in a pair of functions to determine the in-degree or out-degree of any vertex. These are shown in the functions in Figure 3.8. In both the functions `inDegree` and `outDegree` an invocation is made of the `edge` relation with one argument defined and the second undefined. The relation will then cycle through all the bindings that can be made to the undefined argument in order to yield valid edges. Simply counting the number of such bindings is sufficient to determine the degree of the vertex. Notice the resulting procedure is simultaneously making use of functional techniques, logic programming, and imperative constructs.

The third function shown in Figure 3.8 determines the total degree of a vertex, which is defined as the sum of the in-degree and the out-degree of the vertex.

3.4.2 Path Between Two Vertices

Perhaps the most common operation performed using graphs is to determine if a path exists between two arbitrary vertices. The following observation is used by many classic algorithms (for example, Warhall's algorithm): if there is an edge from vertex i to vertex k, and a path from vertex k to to a vertex j, then there is a path from vertex i to vertex j. The relation named `path` is an almost direct encoding of this observation.

```
class Graph;
    ...

    function path (byRef i : integer, byRef j : integer)->relation;
    var
        k : integer;
    begin
```

```
class Graph;
    ...

    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;

    function outDegree (aVertex : integer)->integer;
    var
        tail : integer;
        count : integer;
    begin
        count := 0;
        tail := aVertex;

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

        return count;
    end;

    function degree (aVertex : integer)->integer;
    begin
        return inDegree(aVertex) + outDegree(aVertex);
    end;
end;
```

Figure 3.8: Functions to Computer Degrees of Vertices

```
        return vertex(i) & vertex(j) &
            ((i = j)
             | edge(i, j)
             | (edge(i, k) & removeVertex(i).path(k, j)));
    end;

    ...
end;
```

As long as vertex i and vertex j represent valid vertices, there are three conditions under which a path exists:

1. By definition, every vertex has a path to itself.
2. If there is an edge that directly connects vertex i to vertex j, then there is a path from i to j.
3. In the general case, select a vertex k which is connected by an edge to vertex i, and ask if there is a path from vertex k to vertex j. In order to prevent loops, the vertex i is first removed. Removing vertex i will also remove all edges which directly connect to i. Because removal is performed in a functional fashion, we can use the result of the remove operation as the subject for the recursive call on the function `path`.

Because the function named `path` is written as a relation, it can do much more than simply tell us whether there exists a path from one vertex to another. It can also be used to *generate* the set of vertices which can either be reached from a given vertex, or which will reach a vertex. For example, if we use once more the variable `aGraph` constructed in Section 3.3, the following statement will print those vertices which can be reached starting at vertex number 3.

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

Conversely, the following statement will enumerate those vertices which can reach vertex number 5.

```
i := NIL;
for aGraph.path(i, 5) do begin
    print("vertex");
```

```
    print(i);
    print("can reach vertex 5\n");
end;
```

Note that in both these cases the same element can be generated multiple times if there are multiple distinct paths.

In the first code sequence the first argument is used for "input" values, while the second is used for "output" values, while in the second code sequence these roles are reversed. Because execution in this manner sometimes seems to "flow" one direction while other times seems to "flow" in the reverse, logic programming relations are sometimes described as being *bi-directional*.

3.5 Functions using Graphs

In this section we will examine a number of functions which are not so fundamental that they should be defined as methods in the class `Graph`, but which are nevertheless useful.

3.5.1 Finding Cycles

Once we have defined a method for determining if there is a path between two vertices, it is a simple matter to determine if a graph contains a cycle. Given a vertex v, simply select an edge to which v is connected. Then ask if there is a return path from the head of this edge back to v. This algorithm is made manifest in the following relational function:

```
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;
```

As is the nature of relations, this function can be used both to test if a particular vertex v is part of a cycle, and to enumerate the entire set of vertices which are part of cycles. Often we are not interested so much in iterating over the entire set of vertices which participate in a cycle, but simply in determining whether there exists at least one such vertex. A relation can be invoked as the condition portion of an `if` statement for just this purpose. If there is at least one binding of the variables which makes the relation true, then the condition is defined as being satisfied. If there are no possible bindings for the relation, then the condition is false.

```
v := NIL;
if cycle(aGraph, v) then
    print("graph contains a cycle\n")
else
    print("graph contains no cycles\n");
```

3.5.2 Determine if a Graph is Acyclic

If a directed graph contains no cycles, then the graph is said to be a DAG (Directed Acyclic Graph). By inverting the `cycle` test, we can determine whether or not a graph represents a DAG:

```
function isDag (aGraph : Graph)->boolean;
var
    v : integer;
begin
        { a graph is a dag if there are no cycles }
    return ~ cycle(aGraph, v);
end;
```

3.5.3 Spanning Trees

A spanning tree of a connected graph is a connected, acyclic subgraph that includes all the vertices in the original graph. There is a simple algorithm that can be used to generate a spanning tree:

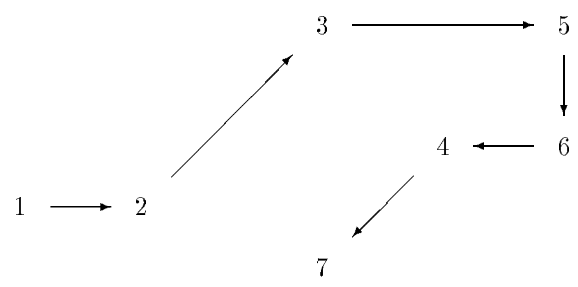
1. Select any vertex as an initial vertex in a new subgraph.
2. Repeatedly iterate through the list of edges. If an edge connects the subgraph to a previously unconnected vertex, then add the new vertex and the edge to the graph.
3. Halt when no new vertices or edges can be added to the graph.

This algorithm is implemented by the following procedure:

```
function spanningTree (aGraph : Graph, byRef st : Graph);
var
    v, i, j : integer;
    flag : boolean;
begin
```

```
    st := Graph();

        { find an initial vertex }
    if aGraph.vertex(v) then
        st := st.addVertex(v);

        { loop as long as we can find new }
        { vertices to add to the spanning tree }
    flag := true;
    while flag do begin
        flag := false;
        for aGraph.edge(i, j) do
            if st.includesVertex(i)
                    & ~ st.includesVertex(j) then
                begin
                    st := st.addVertex(j);
                    st := st.addEdge(Edge(i, j));
                    flag := true;
                end
            else if st.includesVertex(j) &
                    ~ st.includesVertex(i) then
                begin
                    st := st.addVertex(i);
                    st := st.addEdge(Edge(i, j));
                    flag := true;
                end;
        end;
end;
```

Executing this algorithm on the example graph given earlier generates the following spanning tree:



3.5.4 Hamilton Path

A Hamilton path is a traversal through a graph that contains every vertex and touches no vertex twice. The discovery of a Hamilton path is a natural application for the implicit backtracking facility provided by the logic programming paradigm. The function `hamiltonPath` determines whether there is a Hamilton path through a graph which begins at a given vertex. If so, then the path is returned through a pass-by-reference parameter of type `intList`, the list data type we developed in the last chapter.

```
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;
```

The relation divides naturally into two cases. If the graph consists of a single vertex, then it possesses a trivial Hamilton path. This case can be tested by forming the graph which results from removing the starting vertex, then testing to see if the new graph contains an empty set of vertices. In this situation the path returned is a list containing only the starting vertex.

If, on the other hand, the graph consists of more than one vertex, then the following steps are performed:

1. An edge is selected which begins at the starting vertex.
2. The graph formed by removing the starting vertex is constructed. (In fact, since this step is common with the first case, it is factored out of the or-relation and performed first).
3. A test is performed to determine if the new graph possess a Hamilton path.
4. If so, then the starting vertex is added to the resulting Hamilton path.

Of course, a naïve or random selection of edges will likely not yield a Hamilton path, even if one exists. The beauty of the backtracking mechanism used by relations is that a systematic search of all possible alternatives will be undertaken, and this search is guaranteed to discover a solution if one exists.

Knight's Tour

A classic puzzle asks, given an N by M chess board, whether it is possible for a knight to traverse the board in such a way that it visits every square on the board exactly once. We can rephrase this problem as a graph problem, in which case it reduces to finding a Hamilton path. The vertices in this case represent the squares of the board, while the edges represent legal moves. The details of this solution are explored in the exercises.

```
if hamiltonPath(board, 1, path) then begin
    print("knights Tour:");
    for path.items(i) do begin
        print(i);
        print(" ");
    end;
    print("\n");
end;
```

3.6 Combinations of Paradigms

We reiterate how each of the various paradigms finds use in the graph data structure we have developed:

- The overall structure of the graph is object-oriented. By this we mean that a graph is a structure that maintains an internal data state (a set of vertices, and a set of edges) and can perform certain actions.
- Operations which make changes to the graph are written in a functional style. That is, rather than making changes to an existing structure, operations on graphs yield a new structure that reflects the changes.

  Some might argue that this functional style makes operations more difficult to express than would otherwise be necessary. For example, it forces the construction of a graph to be either an extremely large expression formed out of nested function calls, or a lengthy series of assignment statements. On the other hand, the functional approach greatly simplifies many other operations, such as the `path` relation, or the Hamilton path generator.
- The logical paradigm finds use not only in obvious searching problems (the Hamilton path problem) but also as a mechanism for iteration. The advantages of using the

backtracking mechanism for iteration is that loops can then be constructed without requiring detailed knowledge of the structure of the collection over which iteration is being performed.

The seamless integration of the various paradigms provides benefits that would not be easy to emulate using any single paradigm by itself.

Notes and Bibliography

The examples described in this chapter were originally developed as class projects by two students at Oregon State University. I am grateful to Tim Justice and Rebecca Walpole for letting me adapt their work.

Exercises

1. An important operation in the mathematical concept of a set is the ability to tell if one set is a subset of a second set. Write a function that takes two set arguments and performs this task.

2. Rewrite the `intList` class description from Chapter 2 as a parameterized class.

3. Add an iterator method, name `items` as in the `Set` data structure, to the list data type you developed in the previous question.

4. Rewrite the `includes` method in class `Set` as a non-recursive procedure. Is the recursive form "better" than the non-recursive form? How might you define "better"? That is, what factors might you consider in comparing the two implementations?

5. What changes would you need to make to the `Graph` data type in order to support graphs with weighted edges?

6. A more object-oriented version of the `Graph` data type would not only represent the graph as an internal data value, but would make changes to the graph being maintained in response to add or remove operations, rather than returning a new graph. Show how such a data structure (named `OGraph`, to distinguish it from class `Graph`) can be trivially constructed. (Hint: have each instance of `OGraph` maintain an instance of `Graph` as a data field.)

7. How might you implement undirected graphs? Can you simplify the implementation of undirected graphs by building on the implementation of directed graphs?

8. It might appear reasonable to use the function `path` twice in the function `cycle`, rather than a call on `edge` and a call on `path`. Doing so, however, would cause the function `cycle` to return true on all graphs that contain at least one vertex. Explain why.

9. Write a method `lastElement` which returns the final value in a list structure. With this, create a relation which will test for a graph possessing a Hamiltonian circuit. As you will recall, a Hamiltonian path is a circuit if there exists an edge from the final vertex in the path back to the first.

10. Number the squares on an N by M chess board by values from 1 to $N \times M$. Write a procedure which will then generate a graph representing legal moves for a knight chess-piece. That is, vertex i is connected to vertex j if and only if it is legal for a knight in square i to move to square j.

11. Using the representation of the board you developed in the last question, produce a program to generate a knights tour. (Warning, for board sizes larger than 5 by 5, the discovery of a knights tour can take a considerable amount of computer time).