14 Higher-Order Functions and Flexible Search

Chapter Objectives

Lisp higher-order functions
Lisp functions
  `map`
  `filter`
Lisp functions as arguments of functions
  `funcall`
  `apply`
Designing search algorithms in Lisp
  General production system framework
  Use of open and closed lists
Algorithms designed in Lisp
  Depth-first search
  Breadth-first search
  Best first search
  Programmer implements typing as needed

Chapter Contents

14.1 Higher-Order Functions and Abstraction

One of the most powerful techniques that Lisp and other functional programming languages provide is the ability to define functions that take other functions as parameters or return them as results. These functions are called *higher-order functions* and are an important tool for procedural abstraction.

Maps and Filters

A *filter* is a function that applies a test to the elements of a list, eliminating those that fail the test. `filter-negatives`, presented in Section 12.2, was an example of a filter. *Maps* takes a list of data objects and applies a function to each one, returning a list of the results. This idea may be further generalized through the development of general maps and filters that take as arguments both lists and the functions or tests that are to be applied to their elements.

To begin with an example, recall the function `filter-negatives` from Section 12.2. This function took as its argument a list of numbers and returned that list with all negative values deleted. Similarly, we can define a function to filter out all the even numbers in a list. Because these two functions differ *only* in the name of the predicate used to filter elements from the list, it is natural to think of generalizing them into a single function that takes the filtering predicate as a second parameter:

```
(defun filter-evens (number-list)
   (cond ((null number-list) nil)
           ((oddp (car number-list))
                (cons (car number-list)
                     (filter-evens
                 (cdr number-list))))
           (t (filter-evens (cdr number-list)))))
```

This combination of function applications may be defined using a Lisp form called `funcall`, which takes as arguments a function and a series of arguments and applies that function to those arguments:

```
(defun filter (list-of-elements test)
     (cond ((null list-of-elements) nil)
           ((funcall test (car list-of-elements))
             (cons (car list-of-elements)
                   (filter (cdr list-of-elements)
                      test)))
           (t (filter (cdr list-of-elements)
                      test))))
```

The function, `filter`, applies the test to the first element of the list. If the test returns non-`nil`, it `cons`es the element onto the result of `filter` applied to the `cdr` of the list; otherwise, it just returns the filtered `cdr`. This function may be used with different predicates passed in as parameters to perform a variety of filtering tasks:

```
> (filter '(1 3 –9 5 –2 –7 6) #'plusp)
                   ;Filter out all negative numbers.
(1 3 5 6)
> (filter '(1 2 3 4 5 6 7 8 9) #'evenp)
                        ;Filter out all odd numbers.
(2 4 6 8)
> (filter '(1 a b 3 c 4 7 d) #'numberp)
                        ;Filter out all non-numbers.
(1 3 4 7)
```

When a function is passed as a parameter, as in the above examples, it should be preceded by a `#'` instead of just `'`. The purpose of this convention is to flag arguments that are functions so that they may be given appropriate treatment by the Lisp interpreter. In particular, when a function is passed as an argument in Common Lisp, the bindings of its free variables (if any) must be retained. This combination of function definition and bindings of free variables is called a *lexical closure*; the `#'` informs Lisp that the lexical closure must be constructed and passed with the function. More formally, `funcall` is defined:

```
(funcall <function> <arg1> <arg2> … <argn>)
```

In this definition, `<function>` is a Lisp function and `<arg1>` … `<argn>` are zero or more arguments to the function. The result of evaluating a `funcall` is the same as the result of evaluating `<function>` with the specified arguments as actual parameters.

`apply` is a similar function that performs the same task as `funcall` but requires that its arguments be in a list. Except for this syntactic difference, `apply` and `funcall` behave the same; the programmer can choose the function that seems more convenient for a given application. These two functions are similar to `eval` in that all three of them allow the user to specify that the function evaluation should take place. The difference is that `eval` requires its argument to be an s-expression that is evaluated; `funcall` and `apply` take a function and its arguments as separate parameters. Examples of the behavior of these functions include:

```
> (funcall #'plus 2 3)
5
> (apply #'plus '(2 3))
5
> (eval '(plus 2 3))
5
> (funcall #'car '(a b c))
a
> (apply #'car '((a b c)))
a
```

Another important class of higher-order functions consists of mapping functions, functions that will apply a given function to all the elements of a list. Using `funcall`, we define the simple mapping function `map-simple`, which returns a list of the results of applying a functional to all the elements of a list. It has the behavior:

```
(defun map-simple (func list)
     (cond ((null list) nil)
           (t (cons (funcall func (car list))
               (map-simple func (cdr list))))))
> (map-simple #'1+ '(1 2 3 4 5 6))
(2 3 4 5 6 7)
> (map-simple #'listp '(1 2 (3 4) 5 (6 7 8)))
(nil nil t nil t)
```

`map-simple` is a simplified version of a Lisp built-in function `mapcar`, that allows more than one argument list, so that functions of more than one argument can be applied to corresponding elements of several lists:

```
> (mapcar #'1+ '(1 2 3 4 5 6))  ;Same as map-simple.
(2 3 4 5 6 7)
> (mapcar #'+ '(1 2 3 4) '(5 6 7 8))
(6 8 10 12)
> (mapcar #'max '(3 9 1 7) '(2 5 6 8))
(3 9 6 8)
```

`mapcar` is only one of many mapping functions provided by Lisp, as well as only one of many higher-order functions built into the language.

Functional Arguments and Lambda Expressions

In the preceding examples, function arguments were passed by their name and applied to a series of arguments. This requires that the functions be previously defined in the global environment. Frequently, however, it is desirable to pass a function definition directly, without first defining the function globally. This is made possible through the `lambda` expression.

Essentially, the `lambda` expression allows us to separate a function definition from the function name. The origin of lambda expressions is in the *lambda calculus*, a mathematical model of computation that provides (among other things) a particularly thoughtful treatment of this distinction between an object and its name. The syntax of a `lambda` expression is similar to the function definition in a `defun`, except that the function name is replaced by the term `lambda`. That is:

```
(lambda (<formal-parameters>) <body>)
```

`Lambda` expressions may be used in place of a function name in a `funcall` or `apply`. The `funcall` will execute the body of the `lambda` expression with the arguments bound to the parameters of the `funcall`. As with named functions, the number of formal parameters and the number of actual parameters must be the same. For example:

```
> (funcall #'(lambda (x) (* x x)) 4)
16
```

Here, `x` is bound to `4` and the body of the `lambda` expression is then evaluated. The result, the square of `4`, is returned by `funcall`. Other examples of the use of `lambda` expressions with `funcall` and `apply` include:

```
> (apply #'(lambda (x y) (+ (* x x) y)) '(2 3))
7
> (funcall #'(lambda (x) (append x x)) '(a b c))
(a b c a b c)
> (funcall #'(lambda (x1 x2)
      (append (reverse x1) x2)) '(a b c) '(d e f))
(c b a d e f)
```

`Lambda` expressions may be used in a higher-order function such as `mapcar` in place of the names of globally defined functions. For example:

```
> (mapcar #'(lambda (x) (* x x)) '(1 2 3 4 5))
(1 4 9 16 25)
> (mapcar #'(lambda (x) (* x 2)) '(1 2 3 4 5))
(2 4 6 8 10)
> (mapcar #'(lambda (x) (and (> x 0) (< x 10)))
        '(1 24 5 –9 8 23))
(t nil t nil t nil)
```

Without `lambda` expressions the programmer must define every function in the global environment using a `defun`, even though that function may be used only once. `Lambda` expressions free the programmer from this necessity: for example, if it is desired to square each element in a list, the `lambda` form is passed to `mapcar` as the first of the above examples illustrates. It is not necessary to define a squaring function first.

14.2 Search Strategies in Lisp

The use of higher-order functions provides Lisp with a powerful tool for procedural abstraction. In this section, we use this abstraction technique to implement general algorithms for breadth-first, depth-first, and best-first search. These algorithms implement the search algorithms using the open list – the current state list – and the closed list – the already visited states – to manage search through the state space, see Luger (2009, Chapters 3 and 4) and Chapter 4 of this book for similar search algorithms in Prolog.

Breadth-First and Depth-First Search

The Lisp implementation of breadth-first search maintains the open list as a first-in-first-out (FIFO) structure. We will define open and closed as global variables. This is done for several reasons: first to demonstrate the use of global structures in Lisp; second, to contrast the Lisp solution with that in Prolog; and third, it can be argued that since the primary task of this program is to solve a search problem, the state of the search may be represented globally. Finally, since open and closed may be large, their use as global variables seems justified. General arguments of efficiency for the local versus the global approach often depend on the implementation details of a particular language. Global variables in Common Lisp are written to begin and end with `*`. Breadth-first search may be defined:

```
(defun breadth-first ( )
     (cond ((null *open*) nil)
           (t (let ((state (car *open*)))
              (cond ((equal state *goal*) 'success)
                    (t (setq *closed* (cons state
                                      *closed*))
                       (setq *open* (append
                                    (cdr *open*)
                          generate-descendants
                                   state *moves*)))
                       (breadth-first))))))
(defun run-breadth (start goal)
     (setq *open* (list start))
     (setq *closed* nil)
     (setq *goal* goal)
     (breadth-first))
```

In our implementation, the `*open*` list is tested: if it is `nil`, the algorithm returns `nil`, indicating failure as there are no more states to evaluste; If `*open*` is not `nil`, it examines the first element of `*open*`. If this is `equal` to the `goal`, the algorithm halts and returns `success`; otherwise, it calls `generate-descendants` to produce the children of the current `state`, adds them to the `*open*` list, and recurs. `run-breadth` is an initialization function that sets the initial values of `*open*`, `*closed*`, and `*goal*`. `generate-descendants is passed both the state` and `*moves*` as parameters. `*moves*` is a list of the functions that generate moves. In the farmer, wolf, goat, and cabbage problem, assuming the move definitions of Section 13.2, `*moves*` would be:

```
(setq *moves*
     '(farmer-takes-self farmer-takes-wolf
        farmer-takes-goat farmer-takes-cabbage))
```

`generate-descendants` takes a `state` and returns a list of its children. In addition to generating `child` states, it disallows duplicates in the list of children and eliminates any children that are already in the `*open*` or `*closed*` list. In addition to the state, `generate-descendants` is given a list of `moves`; these may be the names of defined functions, or they may be lambda definitions. `generate-descendants` uses a `let` block to save the result of a move in the local variable `child`. We define `generate-descendants`:

```
(defun generate-descendants (state moves)
     (cond ((null moves) nil)
           (t (let ((child (funcall (car moves)
                                         state))
                  (rest (generate-descendants state
                                   (cdr moves))))
                (cond ((null child) rest)
                   ((member child rest :test
                                 #'equal) rest)
                   ((member child *open* :test
                                 #'equal) rest)
                   ((member child *closed* :test
                                 #'equal) rest)
                   (t (cons child rest)))))))
```

As first noted in Section 13.2, the calls to the `member` function use an additional parameter, `:test #'equal`. The `member` function allows the user to specify any test for membership. This allows us to use predicates of arbitrary complexity and semantics to test membership. Though Lisp does not require that we specify the test, the default comparison is the predicate `eq`. `eq` requires that two objects be identical, which means they have the same location in memory; we are using a weaker comparison, `equal`, that only requires that the objects have the same value. By binding the global variable `*moves*` to an appropriate set of move functions, the search algorithm just presented may be used to search any state space graph in a `breadth-first` fashion.

One difficulty that remains with this implementation is its inability to print the list of states along the path from a start to a goal. Although all the states that lead to the goal are present in the closed list when the algorithm halts, these are mixed with all other states from earlier levels of the search space. We can solve this problem by recording both the state and its parent, and reconstructing the solution path from this information. For example, if the state `(e e e e)` generates the state `(w e w e)`, a record of both states, `((w e w e) (e e e e))`, is placed on `*open*`. Later, after the children of the state have been generated, the same `(<state> <parent>)` pair is placed on `*closed*`.

When the current `state` equals the goal, the ancestor information is used to build the path from the goal to the start state by going back to successive parents. This augmented version of `breadth-first` search begins by defining state records as an abstract data type:

```
(defun build-record (state parent)
            (list state parent))

(defun get-state (state-tuple) (nth 0 state-tuple))

(defun get-parent (state-tuple) (nth 1 state-tuple))

(defun retrieve-by-state (state list)
     (cond ((null list) nil)
           ((equal state (get-state (car list)))
                    (car list))
           (t (retrieve-by-state state
                    (cdr list)))))
```

`build-record` constructs a `(<state> <parent>)` pair. `get-state` and `get-parent` access the appropriate fields of a record. `retrieve-by-state` takes a `state` and a list of state records and returns the record whose `state` field matches that `state`.

`build-solution` uses `retrieve-by-state` to chain back from state to parent, constructing a list of successive states that led to a goal. When initializing `*open*`, we will give the starting state a parent of `nil`; `build-solution` stops when passed a `null` state.

```
(defun build-solution (state)
     (cond ((null state) nil)
         (t (cons state (build-solution (get-parent
            (retrieve-by-state state *closed*)))))))
```

The remainder of the algorithm is similar to the breadth-first search of Section 3.2:

```
(defun run-breadth (start goal)
     (setq *open* (list (build-record start nil)))
     (setq *closed* nil)
     (setq *goal* goal)
     (breadth-first))

(defun breadth-first ( )
     (cond ((null *open*) nil)
           (t (let ((state (car *open*)))
              (setq *closed* (cons state *closed*))
                (cond ((equal (get-state state)
                               *goal*)
                           (build-solution *goal*))
                      (t (setq *open* (append (cdr
                               *open*)
                         (generate-descendants
                                     (get-state state)
                                    *moves*)))
                              (breadth-first)))))))
(defun generate-descendants (state moves)
      (cond ((null moves) nil)
            (t (let ((child (funcall
                             (car moves) state))
                     (rest (generate-descendants
                            state (cdr moves))))
                 (cond ((null child) rest)
                    ((retrieve-by-state child rest)
                            rest)
                    ((retrieve-by-state child *open*)
                            rest)
                    ((retrieve-by-state child
                            *closed*) rest)
                    (t (cons (build-record child
                                               state)
                                rest))))))))
```

Depth-first search is implemented by modifying breadth-first search to maintain `*open*` as a stack. This simply involves reversing the order of the arguments to `append`.

Best-First Search

Best-first search may be implemented through straightforward modifications to the breadth-first search algorithm. Specifically, the heuristic evaluation is saved along with each state. The tuples on `*open*` are then sorted according to this evaluation. The data type definitions for state records are an extension of those used in breadth-first search:

```
(defun build-record (state parent depth weight)
   (list state parent depth weight))
(defun get-state (state-tuple) (nth 0 state-tuple))
(defun get-parent (state-tuple) (nth 1 state-tuple))
(defun get-depth (state-tuple) (nth 2 state-tuple))
(defun get-weight (state-tuple) (nth 3 state-tuple))
(defun retrieve-by-state (state list)
     (cond ((null list) nil)
           ((equal state (get-state (car list)))
                  (car list))
           (t (retrieve-by-state state
                        (cdr list)))))
```

`best-first` and `generate-descendants` are defined:

```
(defun best-first ( )
      (cond ((null *open*) nil)
           (t (let ((state (car *open*)))
              (setq *closed* (cons state *closed*))
```

```
                (cond ((equal (get-state state)
                                          *goal*)
                           (build-solution *goal*))
                       (t (setq *open*
                           (insert-by-weight
                              (generate-descendants
                                    (get-state state)
                                 (+ 1 (get-depth
                                   state))
                                   *moves*)
                                    (cdr *open*)))
                (best-first)))))))

(defun generate-descendants (state depth moves)
   (cond ((null moves) nil)
      (t (let ((child (funcall (car moves) state))
               (rest (generate-descendants state
                        depth (cdr moves))))
            (cond ((null child) rest)
               ((retrieve-by-state child rest)
                    rest)
               ((retrieve-by-state child *open*)
                    rest)
               ((retrieve-by-state child *closed*)
                    rest)
               (t (cons (build-record child state
                        depth (+ depth (heuristic
                                      child)))
                          rest)))))))
```

The only differences between `best-first` and `breadth-first` search are the use of `insert-by-weight` to sort the records on `*open*` by their heuristic weights and the computation of search depth and heuristic weights in `generate-descendants`.

Completion of `best-first` requires a definition of `insert-by-weight`. This function takes an unsorted list of state records and inserts them, one at a time, into their appropriate positions in `*open*`. It also requires a problem-specific definition of a function `heuristic`. This function takes a state and, using the global `*goal*`, computes a heuristic weight for that state. We leave the creation of these functions as an exercise for the reader.

Exercises

1. Create a type check that prevents the member check predicate (that checks whether an item is a member of a list of items) from crashing when called on `member(a, a)`. Will this "fix" address the `append(nil, 6, 6)` anomaly that is described in Chapter 10? Test it and determine your success.

2. Implement `build-solution` and `eliminate-duplicates` for the breadth-first search algorithm of Section 14.2.

3. Create a depth-first, a breadth-first, and best first search for the Water Jugs problem (Chapter 13, number 5). This will require you to create a heuristic measure for the Water Jugs problem, as well as create an `insert-by-weight` function for maintaining the priority queue.

4. Create a depth-first, a breadth-first, and best first search for the Missionaries and Cannibals problem (Chapter 13, number 6). This will require you to create a heuristic measure for the Missionaries and Cannibals problem, as well as create an `insert-by-weight` function for maintaining the priority queue.

5. Write a Lisp program to solve the 8-queens problem. (This problem is to find a way to place eight queens on a chessboard so that no queen may capture any other through a single move, i.e., no two queens are on the same row, column, or diagonal.) Do depth-first, breadth-first, and best-first solutions to this problem.

6. Write a Lisp program to solve the full 8 x 8 version of the Knight's Tour problem. This problem asks you to find a path from any square to any other square on the chessboard, using only the knight. Do a depth-first, breadth-first, and best-first solutions for this problem.