13 Variables, Datatypes, and Search

Chapter Objectives

Variables introduced
    Basic support for type systems
Fundamental to search
Creating and binding variables
    `set`
    `let`
Depth-first search in Lisp:
    Use of production system architecture
    Backtracking supports search of all options

Chapter Contents

13.1 Variables and Datatypes

We begin this chapter by demonstrating the creation of variables using `set` and `let` and discussing the scope of their bindings. We then introduce datatypes and discuss run-time type checking. We finish this chapter with a sample search, where we use variables for state and recursion for generation of the state-space graph.

Binding Variables Using Set

Lisp is based on the theory of recursive functions; early Lisp was the first example of a functional or *applicative* programming language. An important aspect of purely functional languages is the lack of any side effects as a result of function execution. This means that the value returned by a function call depends only on the function definition and the value of the parameters in the call. Although Lisp is based on mathematical functions, it is possible to define Lisp forms that violate this property. Consider the following Lisp interaction:

```
> (f 4)
5
> (f 4)
6
> (f 4)
7
```

Note that `f` does not behave as a true function in that its output is not determined solely by its actual parameter: each time it is called with `4`, it returns a different value. This implies that f is retaining its state, and that each execution of the function creates a side effect that influences the behavior of future calls. `f` is implemented using a Lisp built-in function called `set`:

```
(defun f (x)
      (set ‘inc (+ inc 1))
      (+ x inc))
```

`set` takes two arguments. The first must evaluate to a symbol; the second may be an arbitrary s-expression. `set` evaluates the second argument and assigns this value to the symbol defined by the first argument. In the above example, if `inc` is first set to `0` by the call `(set ‘inc 0)`, each subsequent evaluation will increment its parameter by one.

`set` requires that its first argument evaluate to a symbol. In many cases, the first argument is simply a quoted symbol. Because this is done so often, Lisp provides an alternative form, `setq`, which does not evaluate its first argument. Instead, `setq` requires that the first argument be a symbol. For example, the following forms are equivalent:

```
> (set ‘x 0)
0
> (setq x 0)
0
```

Although this use of `set` makes it possible to create Lisp objects that are not pure functions in the mathematical sense, the ability to bind a value to a variable in the global environment is a useful feature. Many programming tasks are most naturally implemented using this ability to define objects whose state persists across function calls. The classic example of this is the "seed" in a random number generator: each call to the function changes and saves the value of the seed. Similarly, it would be natural for a database program (such as was described in Section 11.3) to store the database by binding it to a variable in the global environment.

So far, we have seen two ways of giving a value to a symbol: explicitly, by assignment using `set` or `setq`, or implicitly, when a function call binds the calling parameters to the formal parameters in the definition. In the examples seen so far, all variables in a function body were either *bound* or *free*. A bound variable is one that appears as a formal parameter in the definition of the function, while a free variable is one that appears in the body of the function but is not a formal parameter. When a function is called, any bindings that a bound variable may have in the global environment are saved and the variable is rebound to the calling parameter. After the function has completed execution, the original bindings are restored. Thus, setting the value of a bound variable inside a function body has no effect on the global bindings of that variable, as seen in the Lisp interaction:

```
> (defun foo (x)
      (setq x (+ x 1))    ;increment bound variable x
      x)                          ;return its value.
foo
> (setq y 1)
1
```

```
> (foo y)
2
> y              ;note that value of y is unchanged.
1
```

In the example that began this section, `x` was bound in the function `f`, whereas `inc` was free in that function. As we demonstrated in the example, free variables in a function definition are the primary source of side effects in functions.

An interesting alternative to `set` and `setq` is the generalized assignment function, `setf`. Instead of assigning a value to a symbol, `setf` evaluates its first argument to obtain a memory location and places the value of the second argument in that location. When binding a value to a symbol, `setf` behaves like `setq`:

```
> (setq x 0)
0
> (setf x 0)
0
```

However, because we may call `setf` with any form that corresponds to a memory location, it allows a more general semantics. For example, if we make the first argument to `setf` a call to the `car` function, `setf` will replace the first element of that list. If the first argument to `setf` is a call to the `cdr` function, `setf` will replace the tail of that list. For example:

```
> (setf x '(a b c))          ;x is bound to a list.
(a b c)
> x                          ;The value of x is a list.
(a b c)
> (setf (car x) 1)   ;car of x is a memory location.
1
> x                    ;setf changed value of car of x.
(1 b c)
> (setf (cdr x) '(2 3))
(2 3)
> x                    ;Note that x now has a new tail.
(1 2 3)
```

We may call `setf` with most Lisp forms that correspond to a memory location; these include symbols and functions such as `car`, `cdr`, and `nth`. Thus, `setf` allows the program designer great flexibility in creating, manipulating, and even replacing different components of Lisp data structures.

Defining Local Variables Using `let`

`let` is a useful function for explicitly controlling the binding of variables. `let` allows the creation of local variables.

As an example, consider a function to compute the roots of a quadratic equation. The function quad-roots will take as arguments the three parameters a, b, and c of the equation $ax^2 + bx + c = 0$ and return a list of the two roots of the equation. These roots will be calculated from the formula:

```
    x = -b +/- sqrt(b2 – 4ac)
                 2a
```

For example:

```
> (quad-roots 1 2 1)
(–1.0 –1.0)
> (quad-roots 1 6 8)
(–2.0 –4.0)
```

In computing `quad-roots`, the value of

```
sqrt(b2 – 4ac)
```

is used twice. For reasons of efficiency, as well as elegance, we should compute this value only once, saving it in a variable for use in computing the two roots. Based on this idea, an initial implementation of `quad-roots` might be:

```
(defun quad-roots-1 (a b c)
     (setq temp (sqrt (– (* b b) (* 4 a c))))
          (list (/ (+ (– b) temp) (* 2 a))
                (/ (– (– b) temp) (* 2 a))))
```

Note that the above implementation assumes that the equation does not have imaginary roots, as attempting to take the square root of a negative number would cause the `sqrt` function to halt with an error condition. Modifying the code to handle this case is straightforward and not relevant to this discussion.

Although, with this exception, the code is correct, evaluation of the function body will have the side effect of setting the value of `temp` in the global environment:

```
> (quad-roots-1 1 2 1)
(–1.0 –1.0)
> temp
0.0
```

It is much more desirable to make `temp` local to the function `quad-roots`, thereby eliminating this side effect. This can be done through the use of a `let` block. A `let` expression has the syntax:

```
(let (<local-variables>) <expressions>)
```

where the elements of `(<local-variables>)` are either symbolic atoms or pairs of the form:

```
(<symbol> <expression>)
```

When a `let` form (or *block* as it is usually called) is evaluated, it establishes a local environment consisting of all of the symbols in `(<local-variables>)`. If a symbol is the first element of a pair, the second element is evaluated and the symbol is bound to this result; symbols that are not included in pairs are bound to `nil`. If any of these symbols are already bound in the global environment, these global bindings are saved and restored when the `let` block terminates.

After these local bindings are established, the `<expressions>` are evaluated in order within this environment. When the `let` statement terminates, it returns the value of the last expression evaluated within the block. The behavior of the `let` block is illustrated by the following example:

```
> (setq a 0)
0
> (let ((a 3) b)
      (setq b 4)
      (+ a b))
7
> a
0
> b
ERROR – b is not bound at top level.
```

In this example, before the `let` block is executed, `a` is bound to `0` and `b` is unbound at the top-level environment. When the `let` is evaluated, `a` is bound to `3` and `b` is bound to `nil`. The `setq` assigns `b` to `4`, and the sum of `a` and `b` is returned by the `let` statement. Upon termination of the `let`, `a` and `b` are restored to their previous values, including the unbound status of `b`.

Using the `let` statement, `quad-roots` can be implemented with no global side effects:

```
(defun quad-roots-2 (a b c)
     (let (temp) (setq temp (sqrt (– (* b b)
                                      (* 4 a c))))
          (list (/ (+ (–b) temp) (* 2 a))
                (/ (– (– b) temp) (* 2 a)))))
```

Alternatively, `temp` may be bound when it is declared in the `let` statement, giving a somewhat more concise implementation of `quad-roots`. In this final version, the denominator of the formula, `2a`, is also computed once and saved in a local variable, `denom`:

```
(defun quad-roots-3 (a b c)
     (let ((temp (sqrt (–. (* b b) (* 4 a c))))
          (denom (* 2 a)))
               (list (/ (+ (– b) temp) denom)
                     (/ (– (– b) temp) denom))))
```

In addition to avoiding side effects, `quad-roots-3` is the most efficient of the three versions, because it does not recompute values unnecessarily.

Data Types in Common Lisp

Lisp provides a number of built-in data types. These include integers, floating-point numbers, strings, and characters. Lisp also includes such structured types as arrays, hash tables, sets, and structures. All of these types include the appropriate operations on the type and predicates for testing whether an object is an instance of the type. For example, lists are supported by such functions as `listp`, which identifies an object as a list;

`null`, which identifies the empty list, and constructors and accessors such as `list`, `nth`, `car`, and `cdr`.

However, unlike such strongly typed languages as C or Pascal, where all expressions can be checked for type consistency before run time, in Lisp it is the data objects that are typed, rather than variables. Any Lisp symbol may bind to any object. This provides the programmer with the power of typing but also with a great deal of flexibility in manipulating objects of different or even unknown types. For example, we may bind any object to any variable at run time. This means that we may define data structures such as frames, without fully specifying the types of the values stored in them. To support this flexibility, Lisp implements run-time type checking. So if we bind a value to a symbol, and try to use this value in an erroneous fashion at run time, the Lisp interpreter will detect an error:

```
> (setq x 'a)
a
> (+ x 2)
> > Error: a is not a valid argument to +.
> > While executing: +
```

Users may implement their own type checking using either built-in or user-defined type predicates. This allows the detection and management of type errors as needed.

The preceding pages are not a complete description of Lisp. Instead, they are intended to call the reader's attention to interesting features of the language that will be of use in implementing AI data structures and algorithms. These features include:

- The naturalness with which Lisp supports a data abstraction approach to programming.
- The use of lists to create symbolic data structures.
- The use of `cond` and recursion to control program flow.
- The recursive nature of list structures and the recursive schemes involved in their manipulation.
- The use of `quote` and `eval` to control function evaluation
- The use of `set` and `let` to control variable bindings and side effects.

The remainder of the Lisp section builds on these ideas to demonstrate the use of Lisp for typical AI programming tasks such as pattern matching and the design of graph search algorithms. We begin with a simple example, the Farmer, Wolf, Goat, and Cabbage problem, where an abstract datatype is used to describe states of the world.

13.2 Search: The Farmer, Wolf, Goat, and Cabbage Problem

A Functional Approach to Search

To introduce graph search programming in Lisp, we next represent and solve the farmer, wolf, goat, and cabbage problem:

> A farmer with his wolf, goat, and cabbage come to the edge of a river they wish to cross. There is a boat at the river's edge, but, of course, only the farmer can row it. The boat also can carry only two things (including the rower) at a time. If the wolf is ever left alone with the goat, the wolf will eat the goat; similarly, if the goat is left alone with the cabbage, the goat will eat the cabbage. Devise a sequence of crossings of the river so that all four characters arrive safely on the other side of the river.

This problem was first presented in Prolog in Section 4.2. The Lisp version searches the same space and has structural similarities to the Prolog solution; however, it differs in ways that reflect Lisp's imperative/functional orientation. The Lisp solution searches the state space in a depth-first fashion using a list of visited states to avoid loops.

The heart of the program is a set of functions that define states of the world as an abstract data type. These functions hide the internals of state representation from higher-level components of the program. States are represented as lists of four elements, where each element denotes the location of the farmer, wolf, goat, or cabbage, respectively. Thus, `(e w e w)` represents the state in which the farmer (the first element) and the goat (the third element) are on the east bank and the wolf and cabbage are on the west. The basic functions defining the state data type will be a constructor, `make-state`, which takes as arguments the locations of the farmer, wolf, goat, and cabbage and returns a state, and four access functions, `farmer-side`, `wolf-side`, `goat-side`, and `cabbage-side`, which take a state and return the location of an individual. These functions are defined:

```
(defun make-state (f w g c) (list f w g c))
(defun farmer-side (state)
     (nth 0 state))
(defun wolf-side (state)
     (nth 1 state))
(defun goat-side (state)
     (nth 2 state))
(defun cabbage-side (state)
     (nth 3 state))
```

The rest of the program is built on these state access and construction functions. In particular, they are used to implement the four possible actions the farmer may take: rowing across the river alone or with either of the wolf, goat, or cabbage.

Each move uses the access functions to tear a state apart into its components. A function called `opposite` (to be defined shortly) determines the new location of the individuals that cross the river, and `make-state` reassembles these into the new state. For example, the function `farmer-takes-self` may be defined:

```
(defun farmer-takes-self (state)
     (make-state (opposite (farmer-side state))
           (wolf-side state)
           (goat-side state)
           (cabbage-side state)))
```

Note that `farmer-takes-self` returns the new state, regardless of whether it is safe or not. A state is unsafe if the farmer has left the goat alone with the cabbage or left the wolf alone with the goat. The program must find a solution path that does not contain any unsafe states. Although this "safe" check may be done at a number of different stages in the execution of the program, our approach is to perform it in the move functions. This is implemented by using a function called `safe`, which we also define shortly. `safe` has the following behavior:

```
> (safe '(w w w w)) ;safe state, return unchanged
(w w w w)
> (safe '(e w w e)) ;wolf eats goat, return nil
nil
> (safe '(w w e e)) ;goat eats cabbage, return nil
nil
```

`safe` is used in each move-across-the-river function to filter out the unsafe states. Thus, any move that moves to an unsafe state will return `nil` instead of that state. The recursive `path` algorithm can check for this `nil` and use it to prune that state. In a sense, we are using `safe` to implement a *production system* style condition-check prior to determining if a move rule can be applied. For a detailed discussion of the production system pattern for computation see Luger (2009, Chapter 6). Using `safe`, we next present a final definition for the four move functions:

```
(defun farmer-takes-self (state)
     (safe
          (make-state (opposite (farmer-side state))
                 (wolf-side state)
                 (goat-side state)
                 (cabbage-side state))))

(defun farmer-takes-wolf (state)
     (cond ((equal (farmer-side state)
                (wolf-side state))
              (safe  (make-state
                (opposite (farmer-side state))
                (opposite (wolf-side state))
                (goat-side state)
                (cabbage-side state))))
            (t nil)))
```

```
(defun farmer-takes-goat (state)
     (cond ((equal (farmer-side state)
                   (goat-side state))
            (safe (make-state
              (opposite (farmer-side state))
               (wolf-side state)
               (opposite (goat-side state))
               (cabbage-side state))))
          (t nil)))
(defun farmer-takes-cabbage (state)
     (cond ((equal (farmer-side state)
                   (cabbage-side state))
            (safe (make-state
              (opposite (farmer-side state))
              (wolf-side state)
              (goat-side state)
               (opposite
                  (cabbage-side state)))))
          (t nil)))
```

Note that the last three move functions include a conditional test to determine whether the farmer and the prospective passenger are on the same side of the river. If they are not, the functions return `nil`. The move definitions use the `state` manipulation functions already presented and a function `opposite`, which, for any given side, returns the other side of the river:

```
(defun opposite (side)
     (cond ((equal side 'e) 'w)
           ((equal side 'w) 'e)))
```

Lisp provides a number of different predicates for equality. The most stringent, `eq`, is true only if its arguments evaluate to the same object, i.e., point to *the same memory location*. `equal` is less strict: it requires that its arguments be syntactically identical, as in:

```
> (setq l1 '(1 2 3))
(1 2 3)
> (setq l2 '(1 2 3))
(1 2 3)
> (equal l1 l2)
t
> (eq l1 l2)
nil
> (setq l3 l1)
(1 2 3)
> (eq l1 l3)
t
```

We define safe using a `cond` to check for the two unsafe conditions: (1) the farmer on the opposite bank from the wolf and the goat and (2) the farmer on the opposite bank from the goat and the cabbage. If the state is `safe`, it is returned unchanged; otherwise, `safe` returns `nil`:

```
(defun safe (state)
      (cond ((and (equal (goat-side state)
                         (wolf-side state))
                  (not (equal (farmer-side state)
                         (wolf-side state))))
             nil)
            ((and (equal (goat-side state)
                         (cabbage-side state))
                  (not (equal (farmer-side state)
                         (goat-side state))))
             nil)
            (t state)))
```

`path` implements the backtracking search of the state space. It takes as arguments a `state` and a `goal` and first checks to see whether they are `equal`, indicating a successful termination of the search. If they are not `equal`, `path` generates all four of the neighboring states in the state space graph, calling itself recursively on each of these neighboring states in turn to try to find a path from them to a goal. Translating this simple definition directly into Lisp yields:

```
(defun path (state goal)
      (cond ((equal state goal) 'success)
            (t (or
              (path (farmer-takes-self state) goal)
              (path (farmer-takes-wolf state) goal)
              (path (farmer-takes-goat state) goal)
              (path (farmer-takes-cabbage state)
                  goal)))))
```

This version of the path function is a simple translation of the recursive path algorithm from English into Lisp and has several "bugs" that need to be corrected. It does, however, capture the essential structure of the algorithm and should be examined before continuing to correct the bugs. The first test in the `cond` statement is necessary for a successful completion of the search algorithm. When the `equal state goal` pattern matches, the recursion stops and the atom `success` is returned. Otherwise, `path` generates the four descendant nodes of the search graph and then calls itself on each of the nodes in turn.

In particular, note the use of the `or` form to control evaluation of its arguments. Recall that an `or` evaluates its arguments in turn until one of them returns a non-`nil` value. When this occurs, the `or` terminates without evaluating the other arguments and returns this non-`nil` value as a result. Thus, the `or` not only is used as a logical operator but also provides a way of controlling branching within the space to be searched. The `or` form is used here instead of a `cond` because the value that is being tested and the value that should be returned if the test is non-`nil` are the same.

One problem with using this definition to change the problem state is that a move function may return a value of `nil` if the move may not be made or if it leads to an unsafe state. To prevent `path` from attempting to generate the children of a `nil` state, it must first check whether the current state is `nil`. If it is, `path` should return `nil`.

The other issue that needs to be addressed in the implementation of `path` is that of detecting potential loops in the search space. If the above implementation of `path` is run, the farmer will soon find himself going back and forth alone between the two banks of the river; that is, the algorithm will be stuck in an infinite loop between identical states, both of which it has already visited.

To prevent this looping from happening, `path` is given a third parameter, `been-list`, a list of all the states that have already been visited. Each time that `path` is called recursively on a new state of the world, the parent state will be added to `been-list`. `path` uses the `member` predicate to make sure the current `state` is not a member of `been-list`, i.e., that it has not already been visited. This is accomplished by checking the current problem state for membership in `been-list` before generating its descendants. `path` is now defined:

```
(defun path (state goal been-list)
      (cond ((null state) nil)
            ((equal state goal)
                (reverse (cons state been-list)))
            ((not (member state been-list
                          :test #'equal))
            (or (path (farmer-takes-self state) goal
                      (cons state been-list))
                (path (farmer-takes-wolf state) goal
                      (cons state been-list))
                (path (farmer-takes-goat state) goal
                      (cons state been-list))
                (path (farmer-takes-cabbage state)
                         goal
                      (cons state been-list))))))
```

In the above implementation, `member` is a Common Lisp built-in function that behaves in essentially the same way as the `my-member` function defined in Section 12.2. The only difference is the inclusion of `:test #'equal` in the argument list. Unlike our "home-grown" member function, the Common Lisp built-in form allows the programmer to specify the function that is used in testing for membership. This wrinkle increases the flexibility of the function and should not cause too much concern in this discussion.

Rather than having the function return just the atom `success`, it is better to have it return the actual solution path. Because the series of states on the solution path is already contained in the `been-list`, this list is returned instead. Because the `goal` is not already on `been-list`, it is `cons`ed onto the list. Also, because the list is constructed in reverse order (with the start state as the last element), the list is reversed (constructed in reverse order using another Lisp built-in function, `reverse`) prior to being returned.

Finally, because the `been-list` parameter should be kept "hidden" from the user, a top-level calling function may be written that takes as arguments a `start` and a `goal` state and calls `path` with a `nil` value of `been-list`:

```
(defun solve-fwgc (state goal)
      (path state goal nil))
```

Finally, let us compare our Lisp version of the farmer, wolf, goat, and cabbage problem with the Prolog solution presented in Section 4.2. Not only does the Lisp program solve the same problem, but it also searches exactly the same state space as the Prolog version. This underscores the point that the state space conceptualization of a problem is independent of the implementation of a program for searching that space. Because both programs search the same space, the two implementations have strong similarities; the differences tend to be subtle but provide an interesting contrast between declarative and procedural programming styles.

States in the Prolog version are represented using a predicate, `state(e,e,e,e)`, and the Lisp implementation uses a list. These two representations are more than syntactic variations on one another. The Lisp representation of `state` is defined not only by its list syntax but also by the access and move functions that constitute the abstract data type "state." In the Prolog version, states are patterns; their meaning is determined by the way in which they match other patterns in the Prolog rules.

The Lisp version of `path` is slightly longer than the Prolog version. One reason for this is that the Lisp version must implement a search strategy, whereas the Prolog version takes advantage of Prolog's built-in search algorithm. The control algorithm is explicit in the Lisp version but is implicit in the Prolog version. Because Prolog is built on declarative representation and theorem-proving techniques, the Prolog program is more concise and has a flavor of describing the problem domain, without directly implementing the search algorithm. The price paid for this conciseness is that much of the program's behavior is hidden, determined by Prolog's built-in inference strategies. Programmers may also feel more pressure to make the problem solution conform to Prolog's representational formalism and search strategies. Lisp, on the other hand, allows greater flexibility for the programmer. The price paid here is that the programmer cannot draw on a built-in representation or search strategy and must implement this explicitly.

In Chapter 14 we present higher-level functions, that is, functions that can take other functions as arguments. This gives the Lisp language much of the representational flexibility that meta-predicates (Chapter 5) give to Prolog.

Exercises

1. Write a random number generator in Lisp. This function must maintain a global variable, seed, and return a different random number each time the function is called. For a description of a reasonable random number algorithm, consult any basic algorithms text.

2. Create an "inventory supply" database. Build type checks for a set of six useful queries on these data tuples. Compare your results with the Prolog approach to this same problem as seen in Chapter 5. 2.

3. Write the functions `initialize`, `push`, `top`, `pop`, and `list-stack` to maintain a global stack. These functions should behave:

```
> (initialize)
nil
> (push 'foo)
foo
> (push 'bar)
bar
> (top)
bar
> (list-stack)
(bar foo)
> (pop)
bar
> (list-stack)
(foo)
> (pop)
foo
> (list-stack)
()
```

4. Sets may be represented using lists. Note that these lists should not contain any duplicate elements. Write your own Lisp implementations of the set operations of union, intersection, and set difference. (Do not use Common Lisp's built-in versions of these functions.)

5. Solve the Water Jug problem, using a production system architecture similar to the Farmer, Wolf, Goat, and Cabbage problem presented in Section 13.2.

> There are two jugs, one holding 3 gallons and the other 5 gallons of water. A number of things that can be done with the jugs: they can be filled, emptied, and dumped one into the other either until the poured-into jug is full or until the poured-out-of jug is empty. Devise a sequence of actions that will produce 4 gallons of water in the larger jug. (Hint: only integer values of water are used.)

6. Implement a depth-first backtracking solution (such as was used to solve the farmer, wolf, goat, and cabbage problem in Section 13.2) to the

missionary and cannibal problem:

> Three missionaries and three cannibals come to the bank of a river they wish to cross. There is a boat that will hold only two people, and any of the group can row it. If there are ever more missionaries than cannibals on any side of the river the cannibals will get converted. Devise a series of moves to get everyone across the river with no conversions.