12 Lists and Recursive Search

Chapter Objectives

Lisp processing of arbitrary symbol structures
- Building blocks for data structures
- Designing accessors

The symbol list as building block
- `car`
- `cdr`
- `cons`

Recursion as basis of list processing
- `cdr` recursion
- `car-cdr` recursion

The tree as representative of the structure of a list

Chapter Contents

12.1 Functions, Lists, and Symbolic Computing

Symbolic Computing

Although the Chapter 11 introduced Lisp syntax and demonstrated a few useful Lisp functions, it did so in the context of simple arithmetic examples. The real power of Lisp is in symbolic computing and is based on the use of lists to construct arbitrarily complex data structures of symbolic and numeric atoms, along with the forms needed for manipulating them. We illustrate the ease with which Lisp handles symbolic data structures, as well as the naturalness of data abstraction techniques in Lisp, with a simple database example. Our database application requires the manipulation of employee records containing name, salary, and employee number fields.

These records are represented as lists, with the name, salary, and number fields as the first, second, and third elements of a list. Using `nth`, it is possible to define access functions for the various fields of a data record. For example:

```
(defun name-field (record)
     (nth 0 record))
```

will have the behavior:

```
> (name-field '((Ada Lovelace) 45000.00 38519))
(Ada Lovelace)
```

Similarly, the functions `salary-field` and `number-field` may be defined to access the appropriate fields of a data record. Because a name is itself a list containing two elements, a first name and a last name, it is useful to define functions that take a name as argument and return either the first or last name as a result.

```
(defun first-name (name)
     (nth 0 name))
```

will have the behavior:

```
> (first-name (name-field '((Ada Lovelace) 45000.00
338519)))
Ada
```

In addition to accessing individual fields of a data record, it is also necessary to implement functions to create and modify data records. These are defined using the built-in Lisp function `list`. `list` takes any number of arguments, evaluates them, and returns a list containing those values as its elements. For example:

```
> (list 1 2 3 4)
(1 2 3 4)
> (list '(Ada Lovelace) 45000.00 338519)
((Ada Lovelace) 45000.00 338519)
```

As the second of these examples suggests, `list` may be used to define a constructor for records in the database:

```
(defun build-record (name salary emp-number)
     (list name salary emp-number))
```

will have the behavior:

```
> (build-record '(Alan Turing) 50000.00 135772)
((Alan Turing) 50000.00 135772)
```

Now, using `build-record` and the access functions, we may construct functions that return a modified copy of a record. For example `replace-salary` will behave:

```
(defun replace-salary-field (record new-salary)
(build-record (name-field record)
     new-salary
     (number-field record)))
> (replace-salary-field '((Ada Lovelace) 45000.00
338519) 50000.00)
((Ada Lovelace) 50000.00 338519)
```

Note that this function does not actually update the record itself but produces a modified copy of the record. This updated version may be saved by binding it to a global variable using `setf` (Section 13.1). Although Lisp provides forms that allow a particular element in a list to be modified in the original structure (i.e., without making a copy), good Lisp programming style generally avoids their use, and they are not covered in this text. For Lisp applications involving all but extremely large structures, modifications are generally done by creating a new copy of the structure.

In the above examples, we created an abstract data type for employee records. The various access and update functions defined in this section implement a specialized language appropriate to the meaning of the records, freeing the programmer from concerns about the actual list structures being used to implement the records. This simplifies the

development of higher-level code, as well as making that code much easier to maintain and understand.

Generally, AI programs manipulate large amounts of varied knowledge about problem domains. The data structures used to represent this knowledge, such as objects and semantic networks, are complex, and humans generally find it easier to relate to this knowledge in terms of its meaning rather than the particular syntax of its internal representation. Therefore, data abstraction techniques, always good practice in computer science, are essential tools for the AI programmer. Because of the ease with which Lisp supports the definition of new functions, it is an ideal language for data abstraction.

12.2 Lists as Recursive Structures

Car/cdr Recursion

In the previous section, we used `nth` and `list` to implement access functions for records in a simple "employee" database. Because all employee records were of a determinate length (three elements), these two functions were sufficient to access the fields of records. However, these functions are not adequate for performing operations on lists of unknown length, such as searching through an unspecified number of employee records. To do this, we must be able to scan a list iteratively or recursively, terminating when certain conditions are met (e.g., the desired record is found) or the list is exhausted. In this section we introduce list operations, along with the use of recursion to create list-processing functions.

The basic functions for accessing the components of lists are `car` and `cdr`. `car` takes a single argument, which must be a list, and returns the first element of that list. `cdr` also takes a single argument, which must also be a list, and returns that list with its first argument removed:

```
> (car '(a b c))        ;note that the list is quoted
a
> (cdr '(a b c))
(b c)
> (car '((a b) (c d)))          ;the first element of
(a b)                           ;a list may be a list
> (cdr '((a b) (c d)))
((c d))
> (car (cdr '(a b c d)))
b
```

The way in which `car` and `cdr` operate suggests a recursive approach to manipulating list structures. *To perform an operation on each of the elements of a list:*

> If the list is empty, quit.
>
> Otherwise, operate on the first element and recurse on the remainder of the list.

Using this scheme, we can define a number of useful list-handling functions. For example, Common Lisp includes the predicates `member`, which determines whether one s-expression is a member of a list, and `length`, which determines the length of a list. We define our own

versions of these functions: `my-member` takes two arguments, an arbitrary s-expression and a list, `my-list`. It returns `nil` if the s-expression is not a member of `my-list`; otherwise it returns the list containing the s-expression as its first element:

```
(defun my-member (element my-list)
   (cond ((null my-list) nil)
      ((equal element (car my-list)) my-list)
      (t (my-member element (cdr my-list)))))
```

`my-member` has the behavior:

```
> (my-member 4 '(1 2 3 4 5 6))
(4 5 6)
> (my-member 5 '(a b c d))
nil
```

Similarly, we may define our own versions of `length` and `nth`:

```
(defun my-length (my-list)
     (cond ((null my-list) 0)
           (t (+ (my-length (cdr my-list)) 1))))
(defun my-nth (n my-list)
     (cond ((zerop n) (car my-list))
     ; zerop tests if argument is zero
           (t (my-nth (- n 1) (cdr my-list)))))
```

It is interesting to note that these examples, though presented here to illustrate the use of `car` and `cdr`, reflect the historical development of Lisp. Early versions of the language did not include as many built-in functions as Common Lisp does; programmers defined their own functions for checking list membership, length, etc. Over time, the most generally useful of these functions have been incorporated into the language standard. As an easily extensible language, Common Lisp makes it easy for programmers to create and use their own library of reusable functions.

In addition to the functions `car` and `cdr`, Lisp provides a number of functions for constructing lists. One of these, `list`, which takes as arguments any number of s-expressions, evaluates them, and returns a list of the results, was introduced in Section 10.1. A more primitive list constructor is the function `cons`, that takes two s-expressions as arguments, evaluates them, and returns a list whose `car` is the value of the first argument and whose `cdr` is the value of the second:

```
> (cons 1 '(2 3 4))
(1 2 3 4)
> (cons '(a b) '(c d e))
((a b) c d e)
```

`cons` bears an inverse relationship to `car` and `cdr` in that the `car` of the value returned by a `cons` form is always the first argument to the `cons`, and the `cdr` of the value returned by a `cons` form is always the second argument to that form:

```
> (car (cons 1 '(2 3 4)))
1
> (cdr (cons 1 '(2 3 4)))
(2 3 4)
```

An example of the use of **cons** is seen in the definition of the function **filter-negatives**, which takes a list of numbers as an argument and returns that list with any negative numbers removed. **filter-negatives** recursively examines each element of the list; if the first element is negative, it is discarded and the function returns the result of filtering the negative numbers from the **cdr** of the list. If the first element of the list is positive, it is "**cons**ed" onto the result of **filter-negatives** from the rest of the list:

```
(defun filter-negatives (number-list)
     (cond ((null number-list) nil)
         ((plusp (car number-list))
            (cons (car number-list)
            (filter-negatives
                  (cdr number-list))))
     (t (filter-negatives (cdr number-list)))))
```

This function behaves:

```
> (filter-negatives '(1 –1 2 –2 3 –4))
(1 2 3)
```

This example is typical of the way **cons** is often used in recursive functions on lists. **car** and **cdr** tear lists apart and "drive" the recursion; **cons** selectively constructs the result of the processing as the recursion "unwinds." Another example of this use of **cons** is in redefining the built-in function **append**:

```
(defun my-append (list1 list2)
     (cond ((null list1) list2)
            (t (cons (car list1)
            (my-append (cdr list1) list2)))))
```

which yields the behavior:

```
> (my-append '(1 2 3) '(4 5 6))
(1 2 3 4 5 6)
```

Note that the same recursive scheme is used in the definitions of **my-append**, **my-length**, and **my-member**. Each definition uses the **car** function to remove (and process) the first element of the list, followed by a recursive call on the shortened (tail of the) list; the recursion "bottoms out" on the empty list. As the recursion unwinds, the **cons** function reassembles the solution. This particular scheme is known as ***cdr*** *recursion*, because it uses the **cdr** function to linearly scan and process the elements of a list.

12.3 Nested Lists, Structure, and car/cdr Recursion

Car/cdr Recursion and Nested Structure

Although both `cons` and `append` may be used to combine smaller lists into a single list, it is important to note the difference between these two functions. If `cons` is called with two lists as arguments, it makes the first of these a new first element of the second list, whereas `append` returns a list whose elements are the elements of the two arguments:

```
> (cons '(1 2) '(3 4))
((1 2) 3 4)
> (append '(1 2) '(3 4))
(1 2 3 4)
```

The lists `(1 2 3 4)` and `((1 2) 3 4)` have fundamentally different structures. This difference may be noted graphically by exploiting the isomorphism between lists and trees. The simplest way to map lists onto trees is to create an unlabeled node for each list, with descendants equal to the elements of that list. This rule is applied recursively to the elements of the list that are themselves lists; elements that are atoms are mapped onto leaf nodes of the tree. Thus, the two lists mentioned above generate the different tree structures illustrated in Figure 12.1.

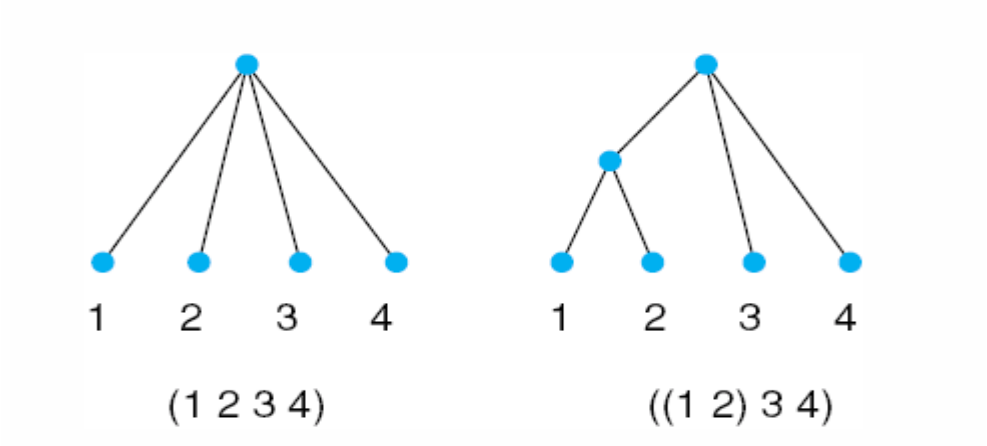

Figure 12.1. Mapping lists onto trees showing structural differences.

This example illustrates the representational power of lists, particularly as a means of representing any tree structure such as a search tree or a parse tree (Figure 16.1). In addition, nested lists provide a way of hierarchically structuring complex data. In the employee records example of Section 12.1, the name field was itself a list consisting of a first name and a last name. This list could be treated as a single entity or its individual components could be accessed.

The simple `cdr`-recursive scheme discussed in the previous section is not sufficient to implement all manipulations on nested lists, because it does not distinguish between items that are lists and those that are simple atoms. Suppose, for example, that the length function defined in Section 12.2 is applied to a nested list structure:

```
> (length '((1 2) 3 (1 (4 (5)))))
3
```

In this example, `length` returns 3 because the list has 3 elements, `(1 2)`, 3, and `(1 (4 (5)))`. This is, of course, the correct and desired behavior for a `length` function.

On the other hand, if we want the function to count the number of *atoms* in the list, we need a different recursive scheme, one that, in addition to scanning along the elements of the list, "opens up" non-atomic list elements and recursively applies itself to the task of counting their atoms. We define this function, called `count-atoms`, and observe its behavior:

```
(defun count-atoms (my-list)
      (cond ((null my-list) 0)
            ((atom my-list) 1)
            (t (+ (count-atoms (car my-list))
                  (count-atoms
                 (cdr my-list))))))
> (count-atoms '((1 2) 3 (((4 5 (6))))))
6
```

The above definition is an example of `car-cdr` recursion. Instead of just recurring on the `cdr` of the list, `count-atoms` also recurs on the `car` of its argument, with the `+` function combining the two components into an answer. Recursion halts when it encounters an `atom` or empty list (`null`). One way of thinking of this scheme is that it adds a second dimension to simple `cdr` recursion, that of "going down into" each of the list elements. Compare the diagrams of calls to `length` and `count-atoms` in Figure 12.2. Note the similarity of `car-cdr` recursion and the recursive definition of s-expressions given in Section 11.1.1.

Another example of the use of `car-cdr` recursion is in the definition of the function `flatten`. `flatten` takes as argument a list of arbitrary structure and returns a list that consists of the same atoms in the same order but with all the atoms at the same level. Note the similarity between the definition of `flatten` and that of `count-atoms`: both use `car-cdr` recursion to tear apart lists and drive the recursion, both terminate when the argument is either `null` or an `atom`, and both use a second function (`append` or `+`) to construct an answer from the results of the recursive calls.

```
(defun flatten (lst)
      (cond ((null lst) nil)
            ((atom lst) (list lst))
            (t (append (flatten (car lst))
                       (flatten (cdr lst))))))
```

Examples of the behavior of `flatten` include:

```
> (flatten '(a (b c) (((d) e f))))
(a b c d e f)
> (flatten '(a b c)); already flattened
(a b c)
```

```
> (flatten '(1 (2 3) (4 (5 6) 7)))
(1 2 3 4 5 6 7)
```

`car-cdr` recursion is the basis of our implementation of unification in Section 15.2. In Chapter 13, we introduce variables and design algorithms for search.

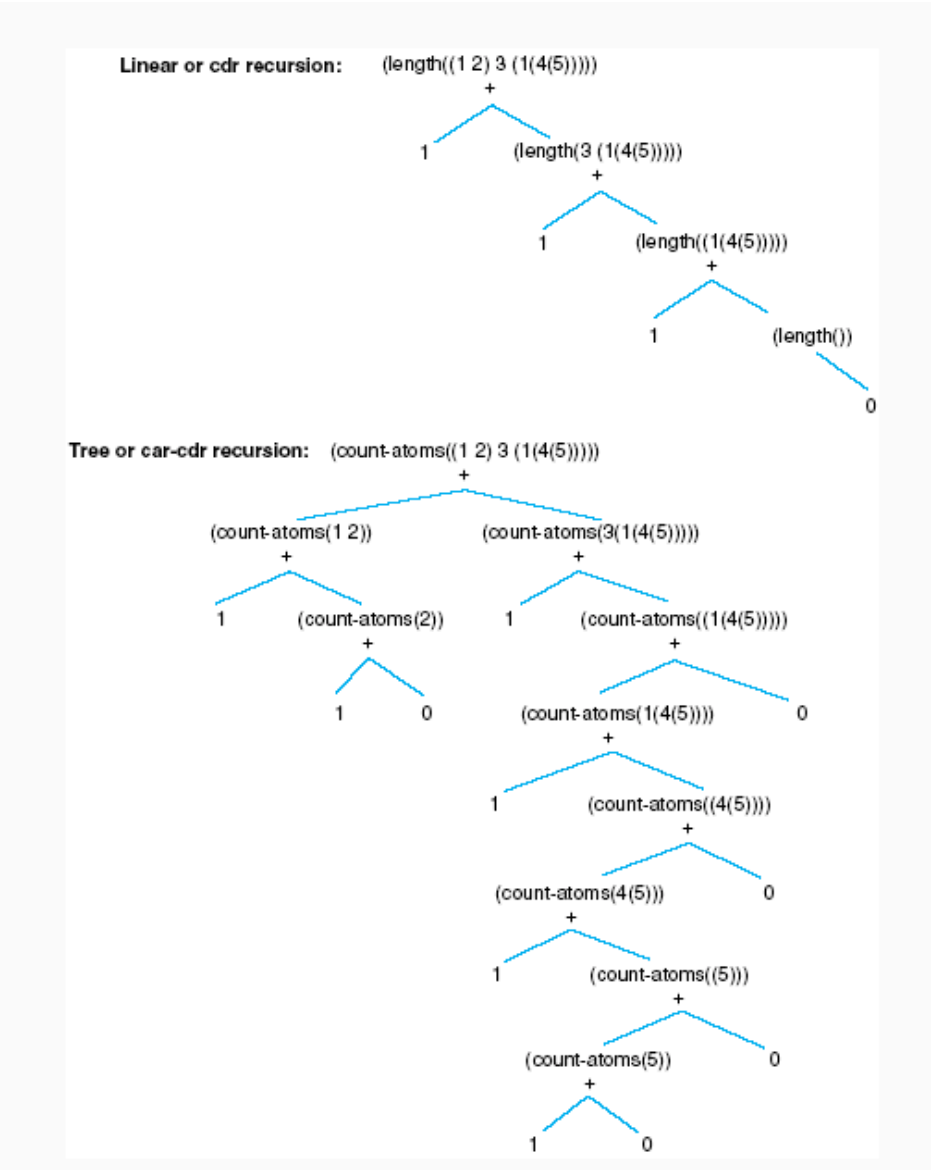

Figure 12.2. Tree representations of linear and tree-recursive functions.

Exercises

1. Create trees, similar to those of Figure 12.1, which show the structures of the following lists.

```
(+ 4 (* 5 (+ 6 7 8)))
(+ (* (+ 4 5) 6 7 8))
(+ (* (+ 4 (* 5 6)) 7) 8)
```

2. Write a recursive Lisp function that will reverse the elements of a list. (Do not use the built-in `reverse` function.) What is the complexity of your implementation? It is possible to reverse a list in linear time; can you do so?

3. Write a Lisp function that will take a list nested to any depth and print the mirror image of that list. For instance, the function should have the behavior:

```
> (mirror '((a b) (c (d e))))
(((e d) c) (b a))
```

Note that the mirroring operation operates at all levels of the list's representation.

4. Consider the database example of section 12.1. Write a function, `find`, to return all records that have a given value particular value for a particular field. To make this more interesting, allow users to specify the fields to be searched by name. For example, evaluating the expression:

```
(find 'salary-field '50000.00
   '(((Alan Turing) 50000.00 135772)
   ((Ada Lovelace) 45000.00 338519)))
```

should return:

```
((Alan Turing) 50000.00 135772)
```

5. The Towers of Hanoi problem is based on the following legend:

> In a Far Eastern monastery, there is a puzzle consisting of three diamond needles and 64 gold disks. The disks are of graduated sizes. Initially, the disks are all stacked on a single needle in decreasing order of size. The monks are attempting to move all the disks to another needle under the following rules:
>
> > Only one disk may be moved at a time.
> >
> > No disk can ever rest on a smaller disk.

Legend has it that when the task has been completed, the universe will end. Write a Lisp program to solve this problem. For safety's sake (and to write a program that will finish in your lifetime) do not attempt the full 64-disk problem. Four or five disks is more reasonable.

6. Write a compiler for arithmetic expressions of the form:

```
(op operand1 operand2)
```

where `op` is either `+`, `–`, `*`, or `/` and the operands are either numbers or nested expressions. An example is `(* (+ 3 6) (– 7 9))`. Assume that the target machine has instructions:

```
(move value register)
(add register-1 register-2)
(subtract register-1 register-2)
(times register-1 register-2)
(divide register-1 register-2)
```

All the arithmetic operations will leave the result in the first register argument. To simplify, assume an unlimited number of registers. Your compiler should take an arithmetic expression and return a list of these machine operations.