4

Utility Functions

Common Lisp operators come in three types: functions and macros, which you can write yourself, and special forms, which you can't. This chapter describes techniques for extending Lisp with new functions. But "techniques" here means something different from what it usually does. The important thing to know about such functions is not how they're written, but where they come from. An extension to Lisp will be written using mostly the same techniques you would use to write any other Lisp function. The hard part of writing these extensions is not deciding how to write them, but deciding which ones to write.

4.1 Birth of a Utility

In its simplest form, bottom-up programming means second-guessing whoever designed your Lisp. At the same time as you write your program, you also add to Lisp new operators which make your program easy to write. These new operators are called *utilities.*

The term "utility" has no precise definition. A piece of code can be called a utility if it seems too small to be considered as a separate application, and too general-purpose to be considered as part of a particular program. A database program would not be a utility, for example, but a function which performed a single operation on a list could be. Most utilities resemble the functions and macros that Lisp has already. In fact, many of Common Lisp's built-in operators began life as utilities. The function `remove-if-not`, which collects all the elements of a list satisfying some predicate, was defined by individual programmers for years before it became a part of Common Lisp.

Learning to write utilities would be better described as learning the habit of writing them, rather than the technique of writing them. Bottom-up programming means simultaneously writing a program and a programming language. To do this well, you have to develop a fine sense of which operators a program is lacking. You have to be able to look at a program and say, "Ah, what you really mean to say is *this*."

For example, suppose that `nicknames` is a function which takes a name and builds a list of all the nicknames which could be derived from it. Given this function, how do we collect all the nicknames yielded by a list of names? Someone learning Lisp might write a function like:

```
(defun all-nicknames (names)
  (if (null names)
      nil
      (nconc (nicknames (car names))
             (all-nicknames (cdr names)))))
```

A more experienced Lisp programmer can look at such a function and say "Ah, what you really want is `mapcan`." Then instead of having to define and call a new function to find all the nicknames of a group of people, you can use a single expression:

```
(mapcan #'nicknames people)
```

The definition of `all-nicknames` is reinventing the wheel. However, that's not all that's wrong with it: it is also burying in a specific function something that could be done by a general-purpose operator.

In this case the operator, `mapcan`, already exists. Anyone who knew about `mapcan` would feel a little uncomfortable looking at `all-nicknames`. To be good at bottom-up programming is to feel equally uncomfortable when the missing operator is one which hasn't been written yet. You must be able to say "what you really want is `x`," and at the same time, to know what `x` should be.

Lisp programming entails, among other things, spinning off new utilities as you need them. The aim of this section is to show how such utilities are born. Suppose that `towns` is a list of nearby towns, sorted from nearest to farthest, and that `bookshops` is a function which returns a list of all the bookshops in a city. If we want to find the nearest town which has any bookshops, and the bookshops in it, we could begin with:

```
(let ((town (find-if #'bookshops towns)))
  (values town (bookshops town)))
```

But this is a bit inelegant: when `find-if` finds an element for which `bookshops` returns a non-nil value, the value is thrown away, only to be recomputed as soon as `find-if` returns. If `bookshops` were an expensive call, this idiom would be inefficient as well as ugly. To avoid unnecessary work, we could use the following function instead:

```
(defun find-books (towns)
  (if (null towns)
      nil
      (let ((shops (bookshops (car towns))))
        (if shops
            (values (car towns) shops)
            (find-books (cdr towns))))))
```

Then calling `(find-books towns)` would at least get us what we wanted with no more computation than necessary. But wait—isn't it likely that at some time in the future we will want to do the same kind of search again? What we really want here is a utility which combines `find-if` and `some`, returning both the successful element, and the value returned by the test function. Such a utility could be defined as:

```
(defun find2 (fn lst)
  (if (null lst)
      nil
      (let ((val (funcall fn (car lst))))
        (if val
            (values (car lst) val)
            (find2 fn (cdr lst))))))
```

Notice the similarity between `find-books` and `find2`. Indeed, the latter could be described as the skeleton of the former. Now, using the new utility, we can achieve our original aim with a single expression:

```
(find2 #'bookshops towns)
```

One of the unique characteristics of Lisp programming is the important role of functions as arguments. This is part of why Lisp is well-adapted to bottom-up programming. It's easier to abstract out the bones of a function when you can pass the flesh back as a functional argument.

Introductory programming courses teach early on that abstraction leads to less duplication of effort. One of the first lessons is: don't wire in behavior. For example, instead of defining two functions which do the same thing but for one or two constants, define a single function and pass the constants as arguments.

In Lisp we can carry this idea further, because we can pass whole functions as arguments. In both of the previous examples we went from a specific function to a more general function which took a functional argument. In the first case we used the predefined `mapcan` and in the second we wrote a new utility, `find2`, but the general principle is the same: instead of mixing the general and the specific, define the general and pass the specific as an argument.

When carefully applied, this principle yields noticeably more elegant programs. It is not the only force driving bottom-up design, but it is a major one. Of the 32 utilities defined in this chapter, 18 take functional arguments.

4.2 Invest in Abstraction

If brevity is the soul of wit, it is also, along with efficiency, the essence of good software. The cost of writing or maintaining a program increases with its length. ○ All other things being equal, the shorter program is the better.

From this point of view, the writing of utilities should be treated as a capital expenditure. By replacing `find-books` with the utility `find2`, we end up with just as many lines of code. But we have made the program shorter in one sense, because the length of the utility does not have to be charged against the current program.

It is not just an accounting trick to treat extensions to Lisp as capital expenditures. Utilities can go into a separate file; they will not clutter our view as we're working on the program, nor are they likely to be involved if we have to return later to change the program in some respect.

As capital expenditures, however, utilities demand extra attention. It is especially important that they be well-written. They are going to be used repeatedly, so any incorrectness or inefficiency will be multiplied. Extra care must also go into their design: a new utility must be written for the general case, not just for the problem at hand. Finally, like any capital expenditure, we need not be in a hurry about it. If you're thinking of spinning off some new operator, but aren't sure that you will want it elsewhere, write it anyway, but leave it with the particular program which uses it. Later if you use the new operator in other programs, you can promote it from a subroutine to a utility and make it generally accessible.

The utility `find2` seems to be a good investment. By making a capital outlay of 7 lines, we get an immediate savings of 7. The utility has paid for itself in the first use. A programming language, Guy Steele wrote, should "cooperate with our natural tendency towards brevity:"

> . . .we tend to believe that the expense of a programming construct is proportional to the amount of writer's cramp that it causes us (by "belief" I mean here an unconscious tendency rather than a fervent

○ conviction). Indeed, this is not a bad psychological principle for language designers to keep in mind. We think of addition as cheap partly because we can notate it with a single character: "+". Even if we believe that a construct is expensive, we will often prefer it to a cheaper one if it will cut our writing effort in half.

In any language, the "tendency towards brevity" will cause trouble unless it is allowed to vent itself in new utilities. The shortest idioms are rarely the most efficient ones. If we want to know whether one list is longer than another, raw Lisp will tempt us to write

```
(> (length x) (length y))
```

If we want to map a function over several lists, we will likewise be tempted to join them together first:

```
(mapcar fn (append x y z))
```

Such examples show that it's especially important to write utilities for situations we might otherwise handle inefficiently. A language augmented with the right utilities will lead us to write more abstract programs. If these utilities are properly defined, it will also lead us to write more efficient ones.

A collection of utilities will certainly make programming easier. But they can do more than that: they can make you write better programs. The muses, like cooks, spring into action at the sight of ingredients. This is why artists like to have a lot of tools and materials in their studios. They know that they are more likely to start something new if they have what they need ready at hand. The same phenomenon appears with programs written bottom-up. Once you have written a new utility, you may find yourself using it more than you would have expected.

The following sections describe several classes of utility functions. They do not by any means represent all the different types of functions you might add to Lisp. However, all the utilities given as examples are ones that have proven their worth in practice.

4.3 Operations on Lists

Lists were originally Lisp's main data structure. Indeed, the name "Lisp" comes from "LISt Processing." It is as well not to be misled by this historical fact, however. Lisp is not inherently about processing lists any more than Polo shirts are for Polo. A highly optimized Common Lisp program might never see a list.

It would still *be* a list, though, at least at compile-time. The most sophisticated programs, which use lists less at runtime, use them proportionately more at

```
(proclaim '(inline last1 single append1 conc1 mklist))

(defun last1 (lst)
  (car (last lst)))

(defun single (lst)
  (and (consp lst) (not (cdr lst))))

(defun append1 (lst obj)
  (append lst (list obj)))

(defun conc1 (lst obj)
  (nconc lst (list obj)))

(defun mklist (obj)
  (if (listp obj) obj (list obj)))
```

Figure 4.1: Small functions which operate on lists.

compile-time, when generating macro expansions. So although the role of lists is decreased in modern dialects, operations on lists can still make up the greater part of a Lisp program.

Figures 4.1 and 4.2 contain a selection of functions which build or examine lists. Those given in Figure 4.1 are among the smallest utilities worth defining. For efficiency, they should all be declared inline (page 26).

The first, `last1`, returns the last element in a list. The built-in function `last` returns the last *cons* in a list, not the last element. Most of the time one uses it to get the last element, by saying `(car (last ...))`. Is it worth writing a new utility for such a case? Yes, when it effectively replaces one of the built-in operators.

Notice that `last1` does no error-checking. In general, none of the code defined in this book will do error-checking. Partly this is just to make the examples clearer. But in shorter utilities it is reasonable not to do any error-checking anyway. If we try:

```
> (last1 "blub")
>>Error: "blub" is not a list.
Broken at LAST...
```

the error will be caught by `last` itself. When utilities are small, they form a layer of abstraction so thin that it starts to be transparent. As one can see through a thin

layer of ice, one can see through utilities like `last1` to interpret errors which arise in the underlying functions.

The function `single` tests whether something is a list of one element. Lisp programs need to make this test rather often. At first one might be tempted to use the natural translation from English:

```
(= (length lst) 1)
```

Written this way, the test would be very inefficient. We know all we need to know as soon as we've looked past the first element.

Next come `append1` and `conc1`. Both attach a new element to the end of a list, the latter destructively. These functions are small, but so frequently needed that they are worth defining. Indeed, `append1` has been predefined in previous Lisp dialects.

So has `mklist`, which was predefined in (at least) Interlisp. Its purpose is to ensure that something is a list. Many Lisp functions are written to return either a single value or a list of values. Suppose that `lookup` is such a function, and that we want to collect the results of calling it on all the elements of a list called `data`. We can do so by writing:

```
(mapcan #'(lambda (d) (mklist (lookup d)))
        data)
```

Figure 4.2 contains some larger examples of list utilities. The first, `longer`, is useful from the point of view of efficiency as well as abstraction. It compares two sequences and returns true only if the first is longer. When comparing the lengths of two lists, it is tempting to do just that:

```
(> (length x) (length y))
```

This idiom is inefficient because it requires the program to traverse the entire length of both lists. If one list is much longer than the other, all the effort of traversing the difference in their lengths will be wasted. It is faster to do as `longer` does and traverse the two lists in parallel.

Embedded within `longer` is a recursive function to compare the lengths of two lists. Since `longer` is for comparing lengths, it should work for anything that you could give as an argument to `length`. But the possibility of comparing lengths in parallel only applies to lists, so the internal function is only called if both arguments are lists.

The next function, `filter`, is to `some` what `remove-if-not` is to `find-if`. The built-in `remove-if-not` returns all the values that might have been returned if you called `find-if` with the same function on successive cdrs of a list. Analogously, `filter` returns what `some` would have returned for successive cdrs of the list:

```
(defun longer (x y)
  (labels ((compare (x y)
             (and (consp x)
                  (or (null y)
                      (compare (cdr x) (cdr y))))))
    (if (and (listp x) (listp y))
        (compare x y)
        (> (length x) (length y)))))

(defun filter (fn lst)
  (let ((acc nil))
    (dolist (x lst)
      (let ((val (funcall fn x)))
        (if val (push val acc))))
    (nreverse acc)))

(defun group (source n)
  (if (zerop n) (error "zero length"))
  (labels ((rec (source acc)
             (let ((rest (nthcdr n source)))
               (if (consp rest)
                   (rec rest (cons (subseq source 0 n) acc))
                   (nreverse (cons source acc))))))
    (if source (rec source nil) nil)))
```

Figure 4.2: Larger functions that operate on lists.

```
> (filter #'(lambda (x) (if (numberp x) (1+ x)))
          '(a 1 2 b 3 c d 4))
(2 3 4 5)
```

You give `filter` a function and a list, and get back a list of whatever non-nil values are returned by the function as it is applied to the elements of the list.

Notice that `filter` uses an accumulator in the same way as the tail-recursive functions described in Section 2.8. Indeed, the aim in writing a tail-recursive function is to have the compiler generate code in the shape of `filter`. For `filter`, the straightforward iterative definition is simpler than the tail-recursive one. The combination of `push` and `nreverse` in the definition of `filter` is the standard Lisp idiom for accumulating a list.

The last function in Figure 4.2 is for grouping lists into sublists. You give `group` a list *l* and a number *n*, and it will return a new list in which the elements

of *l* are grouped into sublists of length *n*. The remainder is put in a final sublist. Thus if we give 2 as the second argument, we get an assoc-list:

```
> (group '(a b c d e f g) 2)
((A B) (C D) (E F) (G))
```

This function is written in a rather convoluted way in order to make it tail-recursive (Section 2.8). The principle of rapid prototyping applies to individual functions as well as to whole programs. When writing a function like `flatten`, it can be a good idea to begin with the simplest possible implementation. Then, once the simpler version works, you can replace it if necessary with a more efficient tail-recursive or iterative version. If it's short enough, the initial version could be left as a comment to describe the behavior of its replacement. (Simpler versions of `group` and several other functions in Figures 4.2 and 4.3 are included in the
○ note on page 389.)

The definition of `group` is unusual in that it checks for at least one error: a second argument of `0`, which would otherwise send the function into an infinite recursion.

In one respect, the examples in this book deviate from usual Lisp practice: to make the chapters independent of one another, the code examples are as much as possible written in raw Lisp. Because it is so useful in defining macros, `group` is an exception, and will reappear at several points in later chapters.

The functions in Figure 4.2 all work their way along the top-level structure of a list. Figure 4.3 shows two examples of functions that descend into nested lists. The first, `flatten`, was also predefined in Interlisp. It returns a list of all the atoms that are elements of a list, or elements of its elements, and so on:

```
> (flatten '(a (b c) ((d e) f)))
(A B C D E F)
```

The other function in Figure 4.3, `prune`, is to `remove-if` as `copy-tree` is to `copy-list`. That is, it recurses down into sublists:

```
> (prune #'evenp '(1 2 (3 (4 5) 6) 7 8 (9)))
(1 (3 (5)) 7 (9))
```

Every leaf for which the function returns true is removed.

4.4 Search

This section gives some examples of functions for searching lists. Common Lisp provides a rich set of built-in operators for this purpose, but some tasks

```
(defun flatten (x)
  (labels ((rec (x acc)
             (cond ((null x) acc)
                   ((atom x) (cons x acc))
                   (t (rec (car x) (rec (cdr x) acc))))))
    (rec x nil)))

(defun prune (test tree)
  (labels ((rec (tree acc)
             (cond ((null tree) (nreverse acc))
                   ((consp (car tree))
                    (rec (cdr tree)
                         (cons (rec (car tree) nil) acc)))
                   (t (rec (cdr tree)
                           (if (funcall test (car tree))
                               acc
                               (cons (car tree) acc)))))))
    (rec tree nil)))
```

Figure 4.3: Doubly-recursive list utilities.

are still difficult—or at least difficult to perform efficiently. We saw this in the
○ hypothetical case described on page 41. The first utility in Figure 4.4, `find2`, is the one we defined in response to it.

The next utility, `before`, is written with similar intentions. It tells you if one object is found before another in a list:

```
> (before 'b 'd '(a b c d))
(B C D)
```

It is easy enough to do this sloppily in raw Lisp:

```
(< (position 'b '(a b c d)) (position 'd '(a b c d)))
```

But the latter idiom is inefficient and error-prone: inefficient because we don't need to find both objects, only the one that occurs first; and error-prone because if either object isn't in the list, `nil` will be passed as an argument to `<`. Using `before` fixes both problems.

Since `before` is similar in spirit to a test for membership, it is written to resemble the built-in `member` function. Like `member` it takes an optional `test` argument, which defaults to `eql`. Also, instead of simply returning `t`, it tries to

```
(defun find2 (fn lst)
  (if (null lst)
      nil
      (let ((val (funcall fn (car lst))))
        (if val
            (values (car lst) val)
            (find2 fn (cdr lst))))))

(defun before (x y lst &key (test #'eql))
  (and lst
       (let ((first (car lst)))
         (cond ((funcall test y first) nil)
               ((funcall test x first) lst)
               (t (before x y (cdr lst) :test test))))))

(defun after (x y lst &key (test #'eql))
  (let ((rest (before y x lst :test test)))
    (and rest (member x rest :test test))))

(defun duplicate (obj lst &key (test #'eql))
  (member obj (cdr (member obj lst :test test))
          :test test))

(defun split-if (fn lst)
  (let ((acc nil))
    (do ((src lst (cdr src)))
        ((or (null src) (funcall fn (car src)))
         (values (nreverse acc) src))
      (push (car src) acc))))
```

Figure 4.4: Functions which search lists.

return potentially useful information: the cdr beginning with the object given as the first argument.

Note that `before` returns true if we encounter the first argument before encountering the second. Thus it will return true if the second argument doesn't occur in the list at all:

```
> (before 'a 'b '(a))
(A)
```

We can peform a more exacting test by calling `after`, which requires that both

its arguments occur in the list:

```
> (after 'a 'b '(b a d))
(A D)
> (after 'a 'b '(a))
NIL
```

If `(member o l)` finds *o* in the list *l*, it also returns the cdr of *l* beginning with *o*. This return value can be used, for example, to test for duplication. If *o* is duplicated in *l*, then it will also be found in the cdr of the list returned by `member`. This idiom is embodied in the next utility, `duplicate`:

```
> (duplicate 'a '(a b c a d))
(A D)
```

Other utilities to test for duplication could be written on the same principle.

More fastidious language designers are shocked that Common Lisp uses `nil` to represent both falsity and the empty list. It does cause trouble sometimes (see Section 14.2), but it is convenient in functions like `duplicate`. In questions of sequence membership, it seems natural to represent falsity as the empty sequence.

The last function in Figure 4.4 is also a kind of generalization of `member`. While `member` returns the cdr of the list beginning with the element it finds, `split-if` returns both halves. This utility is mainly used with lists that are ordered in some respect:

```
> (split-if #'(lambda (x) (> x 4))
            '(1 2 3 4 5 6 7 8 9 10))
(1 2 3 4)
(5 6 7 8 9 10)
```

Figure 4.5 contains search functions of another kind: those which compare elements against one another. The first, `most`, looks at one element at a time. It takes a list and a scoring function, and returns the element with the highest score. In case of ties, the element occurring first wins.

```
> (most #'length '((a b) (a b c) (a) (e f g)))
(A B C)
3
```

For convenience, `most` also returns the score of the winner.

A more general kind of search is provided by `best`. This utility also takes a function and a list, but here the function must be a predicate of two arguments. It returns the element which, according to the predicate, beats all the others.

```
(defun most (fn lst)
  (if (null lst)
      (values nil nil)
      (let* ((wins (car lst))
             (max (funcall fn wins)))
        (dolist (obj (cdr lst))
          (let ((score (funcall fn obj)))
            (when (> score max)
              (setq wins obj
                    max  score))))
        (values wins max))))

(defun best (fn lst)
  (if (null lst)
      nil
      (let ((wins (car lst)))
        (dolist (obj (cdr lst))
          (if (funcall fn obj wins)
              (setq wins obj)))
        wins)))

(defun mostn (fn lst)
  (if (null lst)
      (values nil nil)
      (let ((result (list (car lst)))
            (max (funcall fn (car lst))))
        (dolist (obj (cdr lst))
          (let ((score (funcall fn obj)))
            (cond ((> score max)
                   (setq max    score
                         result (list obj)))
                  ((= score max)
                   (push obj result)))))
        (values (nreverse result) max))))
```

Figure 4.5: Search functions which compare elements.

```
> (best #'> '(1 2 3 4 5))
5
```

We can think of `best` as being equivalent to `car` of `sort`, but much more efficient.

It is up to the caller to provide a predicate which defines a total order on the elements of the list. Otherwise the order of the elements will influence the result; as before, in case of ties, the first element wins.

Finally, `mostn` takes a function and a list and returns a *list* of all the elements for which the function yields the highest score (along with the score itself):

```
> (mostn  #'length '((a b) (a b c) (a) (e f g)))
((A B C) (E F G))
3
```

4.5 Mapping

Another widely used class of Lisp functions are the mapping functions, which apply a function to a sequence of arguments. Figure 4.6 shows some examples of new mapping functions. The first three are for applying a function to a range of numbers without having to cons up a list to contain them. The first two, `map0-n` and `map1-n`, work for ranges of positive integers:

```
> (map0-n #'1+ 5)
(1 2 3 4 5 6)
```

Both are written using the more general `mapa-b`, which works for any range of numbers:

```
> (mapa-b #'1+ -2 0 .5)
(-1 -0.5 0.0 0.5 1.0)
```

Following `mapa-b` is the still more general `map->`, which works for sequences of objects of any kind. The sequence begins with the object given as the second argument, the end of the sequence is defined by the function given as the third argument, and successors are generated by the function given as the fourth argument. With `map->` it is possible to navigate arbitrary data structures, as well as operate on sequences of numbers. We could define `mapa-b` in terms of `map->` as follows:

```
(defun mapa-b (fn a b &optional (step 1))
  (map-> fn
         a
         #'(lambda (x) (> x b))
         #'(lambda (x) (+ x step))))
```

```
(defun map0-n (fn n)
  (mapa-b fn 0 n))

(defun map1-n (fn n)
  (mapa-b fn 1 n))

(defun mapa-b (fn a b &optional (step 1))
  (do ((i a (+ i step))
       (result nil))
      ((> i b) (nreverse result))
    (push (funcall fn i) result)))

(defun map-> (fn start test-fn succ-fn)
  (do ((i start (funcall succ-fn i))
       (result nil))
      ((funcall test-fn i) (nreverse result))
    (push (funcall fn i) result)))

(defun mappend (fn &rest lsts)
  (apply #'append (apply #'mapcar fn lsts)))

(defun mapcars (fn &rest lsts)
  (let ((result nil))
    (dolist (lst lsts)
      (dolist (obj lst)
        (push (funcall fn obj) result)))
    (nreverse result)))

(defun rmapcar (fn &rest args)
  (if (some #'atom args)
      (apply fn args)
      (apply #'mapcar
             #'(lambda (&rest args)
                 (apply #'rmapcar fn args))
             args)))
```

Figure 4.6: Mapping functions.

For efficiency, the built-in mapcan is destructive. It could be duplicated by:

```
(defun our-mapcan (fn &rest lsts)
  (apply #'nconc (apply #'mapcar fn lsts)))
```

Because mapcan splices together lists with nconc, the lists returned by the first argument had better be newly created, or the next time we look at them they might be altered. That's why nicknames (page 41) was defined as a function which "builds a list" of nicknames. If it simply returned a list stored elsewhere, it wouldn't have been safe to use mapcan. Instead we would have had to splice the returned lists with append. For such cases, mappend offers a nondestructive alternative to mapcan.

The next utility, mapcars, is for cases where we want to mapcar a function over several lists. If we have two lists of numbers and we want to get a single list of the square roots of both, using raw Lisp we could say

```
(mapcar #'sqrt (append list1 list2))
```

but this conses unnecessarily. We append together list1 and list2 only to discard the result immediately. With mapcars we can get the same result from:

```
(mapcars #'sqrt list1 list2)
```

and do no unnecessary consing.

The final function in Figure 4.6 is a version of mapcar for trees. Its name, rmapcar, is short for "recursive mapcar," and what mapcar does on flat lists, it does on trees:

```
> (rmapcar #'princ '(1 2 (3 4 (5) 6) 7 (8 9)))
123456789
(1 2 (3 4 (5) 6) 7 (8 9))
```

Like mapcar, it can take more than one list argument.

```
> (rmapcar #'+ '(1 (2 (3) 4)) '(10 (20 (30) 40)))
(11 (22 (33) 44))
```

Several of the functions which appear later on ought really to call rmapcar, including rep_ on page 324.

To some extent, traditional list mapping functions may be rendered obsolete by the new series macros introduced in CLTL2. For example,

```
(mapa-b #'fn a b c)
```

could be rendered

```
(defun readlist (&rest args)
  (values (read-from-string
           (concatenate 'string "("
                                (apply #'read-line args)
                                ")"))))

(defun prompt (&rest args)
  (apply #'format *query-io* args)
  (read *query-io*))

(defun break-loop (fn quit &rest args)
  (format *query-io* "Entering break-loop.~%")
  (loop
    (let ((in (apply #'prompt args)))
      (if (funcall quit in)
          (return)
          (format *query-io* "~A~%" (funcall fn in))))))
```

Figure 4.7: I/O functions.

```
(collect (#Mfn (scan-range :from a :upto b :by c)))
```

However, there is still some call for mapping functions. A mapping function may in some cases be clearer or more elegant. Some things we could express with `map->` might be difficult to express using series. Finally, mapping functions, as functions, can be passed as arguments.

4.6 I/O

Figure 4.7 contains three examples of I/O utilities. The need for this kind of utility varies from program to program. Those in Figure 4.7 are just a representative sample. The first is for the case where you want users to be able to type in expressions without parentheses; it reads a line of input and returns it as a list:

```
> (readlist)
Call me "Ed"
(CALL ME "Ed")
```

The call to `values` ensures that we get only one value back (`read-from-string` itself returns a second value that is irrelevant in this case).

The function `prompt` combines printing a question and reading the answer. It takes the arguments of `format`, except the initial stream argument.

```
> (prompt "Enter a number between ~A and ~A.~%>> " 1 10)
Enter a number between 1 and 10.
>> 3
3
```

Finally, `break-loop` is for situations where you want to imitate the Lisp toplevel. It takes two functions and an `&rest` argument, which is repeatedly given to `prompt`. As long as the second function returns false for the input, the first function is applied to it. So for example we could simulate the actual Lisp toplevel with:

```
> (break-loop #'eval #'(lambda (x) (eq x :q)) ">> ")
Entering break-loop.
>> (+ 2 3)
5
>> :q
:Q
```

This, by the way, is the reason Common Lisp vendors generally insist on runtime licenses. If you can call `eval` at runtime, then any Lisp program can include Lisp.

4.7 Symbols and Strings

Symbols and strings are closely related. By means of printing and reading functions we can go back and forth between the two representations. Figure 4.8 contains examples of utilities which operate on this border. The first, `mkstr`, takes any number of arguments and concatenates their printed representations into a string:

```
> (mkstr pi " pieces of " 'pi)
"3.141592653589793 pieces of PI"
```

Built upon it is `symb`, which is mostly used for building symbols. It takes one or more arguments and returns the symbol (creating one if necessary) whose printname is their concatenation. It can take as an argument any object which has a printable representation: symbols, strings, numbers, even lists.

```
> (symb 'ar "Madi" #\L #\L 0)
|ARMadiLL0|
```

```
(defun mkstr (&rest args)
  (with-output-to-string (s)
    (dolist (a args) (princ a s))))

(defun symb (&rest args)
  (values (intern (apply #'mkstr args))))

(defun reread (&rest args)
  (values (read-from-string (apply #'mkstr args))))

(defun explode (sym)
  (map 'list #'(lambda (c)
                 (intern (make-string 1
                                      :initial-element c)))
             (symbol-name sym)))
```

Figure 4.8: Functions which operate on symbols and strings.

After calling `mkstr` to concatenate all its arguments into a single string, `symb` sends the string to `intern`. This function is Lisp's traditional symbol-builder: it takes a string and either finds the symbol which prints as the string, or makes a new one which does.

Any string can be the print-name of a symbol, even a string containing lowercase letters or macro characters like parentheses. When a symbol's name contains such oddities, it is printed within vertical bars, as above. In source code, such symbols should either be enclosed in vertical bars, or the offending characters preceded by backslashes:

```
> (let ((s (symb '(a b))))
    (and (eq s '|(A B)|) (eq s '\(A\ B\))))
T
```

The next function, `reread`, is a generalization of `symb`. It takes a series of objects, and prints and rereads them. It can return symbols like `symb`, but it can also return anything else that `read` can. Read-macros will be invoked instead of being treated as part of a symbol's name, and `a:b` will be read as the symbol `b` in package `a`, instead of the symbol `|a:b|` in the current package.[1] The more general function is also pickier: `reread` will generate an error if its arguments are not proper Lisp syntax.

[1] For an introduction to packages, see the Appendix beginning on page 381.

The last function in Figure 4.8 was predefined in several earlier dialects: `explode` takes a symbol and returns a list of symbols made from the characters in its name.

```
> (explode 'bomb)
(B O M B)
```

It is no accident that this function wasn't included in Common Lisp. If you find yourself wanting to take apart symbols, you're probably doing something inefficient. However, there is a place for this kind of utility in prototypes, if not in production software.

4.8 Density

If your code uses a lot of new utilities, some readers may complain that it is hard to understand. People who are not yet very fluent in Lisp will only be used to reading raw Lisp. In fact, they may not be used to the idea of an extensible language at all. When they look at a program which depends heavily on utilities, it may seem to them that the author has, out of pure eccentricity, decided to write the program in some sort of private language.

All these new operators, it might be argued, make the program harder to read. One has to understand them all before being able to read the program. To see why this kind of statement is mistaken, consider the case described on page 41, in which we want to find the nearest bookshops. If you wrote the program using `find2`, someone could complain that they had to understand the definition of this new utility before they could read your program. Well, suppose you hadn't used `find2`. Then, instead of having to understand the definition of `find2`, the reader would have had to understand the definition of `find-books`, in which the function of `find2` is mixed up with the specific task of finding bookshops. It is no more difficult to understand `find2` than `find-books`. And here we have only used the new utility once. Utilities are meant to be used repeatedly. In a real program, it might be a choice between having to understand `find2`, and having to understand three or four specialized search routines. Surely the former is easier.

So yes, reading a bottom-up program requires one to understand all the new operators defined by the author. But this will nearly always be less work than having to understand all the code that would have been required without them.

If people complain that using utilities makes your code hard to read, they probably don't realize what the code would look like if you hadn't used them. Bottom-up programming makes what would otherwise be a large program look like a small, simple one. This can give the impression that the program doesn't do much, and should therefore be easy to read. When inexperienced readers look closer and find that this isn't so, they react with dismay.

We find the same phenomenon in other fields: a well-designed machine may have fewer parts, and yet look more complicated, because it is packed into a smaller space. Bottom-up programs are conceptually denser. It may take an effort to read them, but not as much as it would take if they hadn't been written that way.

There is one case in which you might deliberately avoid using utilities: if you had to write a small program to be distributed independently of the rest of your code. A utility usually pays for itself after two or three uses, but in a small program, a utility might not be used enough to justify including it.