2

Functions

Functions are the building-blocks of Lisp programs. They are also the building-blocks of Lisp. In most languages the + operator is something quite different from user-defined functions. But Lisp has a single model, function application, to describe all the computation done by a program. The Lisp + operator is a function, just like the ones you can define yourself.

In fact, except for a small number of operators called *special forms*, the core of Lisp is a collection of Lisp functions. What's to stop you from adding to this collection? Nothing at all: if you think of something you wish Lisp could do, you can write it yourself, and your new function will be treated just like the built-in ones.

This fact has important consequences for the programmer. It means that any new function could be considered either as an addition to Lisp, or as part of a specific application. Typically, an experienced Lisp programmer will write some of each, adjusting the boundary between language and application until the two fit one another perfectly. This book is about how to achieve a good fit between language and application. Since everything we do toward this end ultimately depends on functions, functions are the natural place to begin.

2.1 Functions as Data

Two things make Lisp functions different. One, mentioned above, is that Lisp itself is a collection of functions. This means that we can add to Lisp new operators of our own. Another important thing to know about functions is that they are Lisp objects.

Lisp offers most of the data types one finds in other languages. We get integers and floating-point numbers, strings, arrays, structures, and so on. But Lisp supports one data type which may at first seem surprising: the function. Nearly all programming languages provide some form of function or procedure. What does it mean to say that Lisp provides them as a data type? It means that in Lisp we can do with functions all the things we expect to do with more familiar data types, like integers: create new ones at runtime, store them in variables and in structures, pass them as arguments to other functions, and return them as results.

The ability to create and return functions at runtime is particularly useful. This might sound at first like a dubious sort of advantage, like the self-modifying machine language programs one can run on some computers. But creating new functions at runtime turns out to be a routinely used Lisp programming technique.

2.2 Defining Functions

Most people first learn how to make functions with `defun`. The following expression defines a function called `double` which returns twice its argument.

```
> (defun double (x) (* x 2))
DOUBLE
```

Having fed this to Lisp, we can call `double` in other functions, or from the toplevel:

```
> (double 1)
2
```

A file of Lisp code usually consists mainly of such `defun`s, and so resembles a file of procedure definitions in a language like C or Pascal. But something quite different is going on. Those `defun`s are not just procedure definitions, they're Lisp calls. This distinction will become clearer when we see what's going on underneath `defun`.

Functions are objects in their own right. What `defun` really does is build one, and store it under the name given as the first argument. So as well as calling `double`, we can get hold of the function which implements it. The usual way to do so is by using the `#'` (sharp-quote) operator. This operator can be understood as mapping names to actual function objects. By affixing it to the name of `double`

```
> #'double
#<Interpreted-Function C66ACE>
```

we get the actual object created by the definition above. Though its printed representation will vary from implementation to implementation, a Common Lisp function is a first-class object, with all the same rights as more familiar objects like numbers and strings. So we can pass this function as an argument, return it, store it in a data structure, and so on:

```
> (eq #'double (car (list #'double)))
T
```

We don't even need `defun` to make functions. Like most Lisp objects, we can refer to them literally. When we want to refer to an integer, we just use the integer itself. To represent a string, we use a series of characters surrounded by double-quotes. To represent a function, we use what's called a *lambda-expression.* A lambda-expression is a list with three parts: the symbol `lambda`, a parameter list, and a body of zero or more expressions. This lambda-expression refers to a function equivalent to `double`:

```
(lambda (x) (* x 2))
```

It describes a function which takes one argument *x*, and returns *2x*.

A lambda-expression can also be considered as the name of a function. If `double` is a proper name, like "Michelangelo," then `(lambda (x) (* x 2))` is a definite description, like "the man who painted the ceiling of the Sistine Chapel." By putting a sharp-quote before a lambda-expression, we get the corresponding function:

```
> #'(lambda (x) (* x 2))
#<Interpreted-Function C674CE>
```

This function behaves exactly like `double`, but the two are distinct objects.

In a function call, the name of the function appears first, followed by the arguments:

```
> (double 3)
6
```

Since lambda-expressions are also names of functions, they can also appear first in function calls:

```
> ((lambda (x) (* x 2)) 3)
6
```

In Common Lisp, we can have a function named `double` and a variable named `double` at the same time.

```
> (setq double 2)
2
> (double double)
4
```

When a name occurs first in a function call, or is preceded by a sharp-quote, it is taken to refer to a function. Otherwise it is treated as a variable name.

It is therefore said that Common Lisp has distinct *name-spaces* for variables and functions. We can have a variable called `foo` and a function called `foo`, and they need not be identical. This situation can be confusing, and leads to a certain amount of ugliness in code, but it is something that Common Lisp programmers have to live with.

If necessary, Common Lisp provides two functions which map symbols to the values, or functions, that they represent. The function `symbol-value` takes a symbol and returns the value of the corresponding special variable:

```
> (symbol-value 'double)
2
```

while `symbol-function` does the same for a globally defined function:

```
> (symbol-function 'double)
#<Interpreted-Function C66ACE>
```

Note that, since functions are ordinary data objects, a variable could have a function as its value:

```
> (setq x #'append)
#<Compiled-Function 46B4BE>
> (eq (symbol-value 'x) (symbol-function 'append))
T
```

Beneath the surface, `defun` is setting the `symbol-function` of its first argument to a function constructed from the remaining arguments. The following two expressions do approximately the same thing:

```
(defun double (x) (* x 2))

(setf (symbol-function 'double)
      #'(lambda (x) (* x 2)))
```

So `defun` has the same effect as procedure definition in other languages—to associate a name with a piece of code. But the underlying mechanism is not the same. We don't need `defun` to make functions, and functions don't have to be stored away as the value of some symbol. Underlying `defun`, which resembles procedure definition in any other language, is a more general mechanism: building a function and associating it with a certain name are two separate operations. When we don't need the full generality of Lisp's notion of a function, `defun` makes function definition as simple as in more restrictive languages.

2.3 Functional Arguments

Having functions as data objects means, among other things, that we can pass them as arguments to other functions. This possibility is partly responsible for the importance of bottom-up programming in Lisp.

A language which allows functions as data objects must also provide some way of calling them. In Lisp, this function is `apply`. Generally, we call `apply` with two arguments: a function, and a list of arguments for it. The following four expressions all have the same effect:

```
(+ 1 2)

(apply #'+ '(1 2))

(apply (symbol-function '+) '(1 2))

(apply #'(lambda (x y) (+ x y)) '(1 2))
```

In Common Lisp, `apply` can take any number of arguments, and the function given first will be applied to the list made by consing the rest of the arguments onto the list given last. So the expression

```
(apply #'+ 1 '(2))
```

is equivalent to the preceding four. If it is inconvenient to give the arguments as a list, we can use `funcall`, which differs from `apply` only in this respect. This expression

```
(funcall #'+ 1 2)
```

has the same effect as those above.

Many built-in Common Lisp functions take functional arguments. Among the most frequently used are the mapping functions. For example, `mapcar` takes two or more arguments, a function and one or more lists (one for each parameter of the function), and applies the function successively to elements of each list:

```
> (mapcar #'(lambda (x) (+ x 10))
          '(1 2 3))
(11 12 13)
> (mapcar #'+
          '(1 2 3)
          '(10 100 1000))
(11 102 1003)
```

Lisp programs frequently want to do something to each element of a list and get back a list of results. The first example above illustrates the conventional way to do this: make a function which does what you want done, and mapcar it over the list.

Already we see how convenient it is to be able to treat functions as data. In many languages, even if we could pass a function as an argument to something like `mapcar`, it would still have to be a function defined in some source file beforehand. If just one piece of code wanted to add 10 to each element of a list, we would have to define a function, called `plus_ten` or some such, just for this one use. With lambda-expressions, we can refer to functions directly.

One of the big differences between Common Lisp and the dialects which preceded it are the large number of built-in functions that take functional arguments. Two of the most commonly used, after the ubiquitous `mapcar`, are `sort` and `remove-if`. The former is a general-purpose sorting function. It takes a list and a predicate, and returns a list sorted by passing each pair of elements to the predicate.

```
> (sort '(1 4 2 5 6 7 3) #'<)
(1 2 3 4 5 6 7)
```

To remember how `sort` works, it helps to remember that if you sort a list with no duplicates by <, and then apply < to the resulting list, it will return true.

If `remove-if` weren't included in Common Lisp, it might be the first utility you would write. It takes a function and a list, and returns all the elements of the list for which the function returns false.

```
> (remove-if #'evenp '(1 2 3 4 5 6 7))
(1 3 5 7)
```

As an example of a function which takes functional arguments, here is a definition of a limited version of `remove-if`:

```
(defun our-remove-if (fn lst)
  (if (null lst)
      nil
      (if (funcall fn (car lst))
          (our-remove-if fn (cdr lst))
          (cons (car lst) (our-remove-if fn (cdr lst))))))
```

Note that within this definition `fn` is not sharp-quoted. Since functions are data objects, a variable can have a function as its regular value. That's what's happening here. Sharp-quote is only for referring to the function named by a symbol—usually one globally defined as such with `defun`.

As Chapter 4 will show, writing new utilities which take functional arguments is an important element of bottom-up programming. Common Lisp has so many utilities built-in that the one you need may exist already. But whether you use built-ins like `sort`, or write your own utilities, the principle is the same. Instead of wiring in functionality, pass a functional argument.

2.4 Functions as Properties

The fact that functions are Lisp objects also allows us to write programs which can be extended to deal with new cases on the fly. Suppose we want to write a function which takes a type of animal and behaves appropriately. In most languages, the way to do this would be with a `case` statement, and we can do it this way in Lisp as well:

```
(defun behave (animal)
  (case animal
    (dog (wag-tail)
         (bark))
    (rat (scurry)
         (squeak))
    (cat (rub-legs)
         (scratch-carpet))))
```

What if we want to add a new type of animal? If we were planning to add new animals, it would have been better to define `behave` as follows:

```
(defun behave (animal)
  (funcall (get animal 'behavior)))
```

and to define the behavior of an individual animal as a function stored, for example, on the property list of its name:

```
(setf (get 'dog 'behavior)
      #'(lambda ()
          (wag-tail)
          (bark)))
```

This way, all we need do in order to add a new animal is define a new property. No functions have to be rewritten.

The second approach, though more flexible, looks slower. It is. If speed were critical, we would use structures instead of property lists and, especially, compiled instead of interpreted functions. (Section 2.9 explains how to make these.) With structures and compiled functions, the more flexible type of code can approach or exceed the speed of versions using `case` statements.

This use of functions corresponds to the concept of a *method* in object-oriented programming. Generally speaking, a method is a function which is a property of an object, and that's just what we have. If we add inheritance to this model, we'll have all the elements of object-oriented programming. Chapter 25 will show that this can be done with surprisingly little code.

One of the big selling points of object-oriented programming is that it makes programs extensible. This prospect excites less wonder in the Lisp world, where extensibility has always been taken for granted. If the kind of extensibility we need does not depend too much on inheritance, then plain Lisp may already be sufficient.

2.5 Scope

Common Lisp is a lexically scoped Lisp. Scheme is the oldest dialect with lexical scope; before Scheme, dynamic scope was considered one of the defining features of Lisp.

The difference between lexical and dynamic scope comes down to how an implementation deals with free variables. A symbol is *bound* in an expression if it has been established as a variable, either by appearing as a parameter, or by variable-binding operators like `let` and `do`. Symbols which are not bound are said to be *free*. In this example, scope comes into play:

```
(let ((y 7))
  (defun scope-test (x)
    (list x y)))
```

Within the `defun` expression, `x` is bound and `y` is free. Free variables are interesting because it's not obvious what their values should be. There's no uncertainty about the value of a bound variable—when `scope-test` is called, the value of `x` should be whatever is passed as the argument. But what should be the value of `y`? This is the question answered by the dialect's scope rules.

In a dynamically scoped Lisp, to find the value of a free variable when executing `scope-test`, we look back through the chain of functions that called it. When we find an environment where `y` was bound, that binding of `y` will be the one used in `scope-test`. If we find none, we take the global value of `y`. Thus, in a dynamically scoped Lisp, `y` would have the value it had in the calling expression:

```
> (let ((y 5))
    (scope-test 3))
(3 5)
```

With dynamic scope, it means nothing that `y` was bound to `7` when `scope-test` was defined. All that matters is that `y` had a value of `5` when `scope-test` was called.

In a lexically scoped Lisp, instead of looking back through the chain of calling functions, we look back through the containing environments at the time the function was *defined.* In a lexically scoped Lisp, our example would catch the binding of `y` where `scope-test` was defined. So this is what would happen in Common Lisp:

```
> (let ((y 5))
    (scope-test 3))
(3 7)
```

Here the binding of `y` to `5` at the time of the call has no effect on the returned value.

Though you can still get dynamic scope by declaring a variable to be `special`, lexical scope is the default in Common Lisp. On the whole, the Lisp community seems to view the passing of dynamic scope with little regret. For one thing, it used to lead to horribly elusive bugs. But lexical scope is more than a way of avoiding bugs. As the next section will show, it also makes possible some new programming techniques.

2.6 Closures

Because Common Lisp is lexically scoped, when we define a function containing free variables, the system must save copies of the bindings of those variables at the time the function was defined. Such a combination of a function and a set of variable bindings is called a *closure.* Closures turn out to be useful in a wide variety of applications.

Closures are so pervasive in Common Lisp programs that it's possible to use them without even knowing it. Every time you give `mapcar` a sharp-quoted lambda-expression containing free variables, you're using closures. For example, suppose we want to write a function which takes a list of numbers and adds a certain amount to each one. The function `list+`

```
(defun list+ (lst n)
  (mapcar #'(lambda (x) (+ x n))
          lst))
```

will do what we want:

```
> (list+ '(1 2 3) 10)
(11 12 13)
```

If we look closely at the function which is passed to `mapcar` within `list+`, it's actually a closure. The instance of `n` is free, and its binding comes from the surrounding environment. Under lexical scope, every such use of a mapping function causes the creation of a closure.[1]

○ Closures play a more conspicuous role in a style of programming promoted by Abelson and Sussman's classic *Structure and Interpretation of Computer Programs.* Closures are functions with local state. The simplest way to use this state is in a situation like the following:

```
(let ((counter 0))
  (defun new-id ()   (incf counter))
  (defun reset-id () (setq counter 0)))
```

These two functions share a variable which serves as a counter. The first one returns successive values of the counter, and the second resets the counter to `0`. The same thing could be done by making the counter a global variable, but this way it is protected from unintended references.

It's also useful to be able to return functions with local state. For example, the function `make-adder`

```
(defun make-adder (n)
  #'(lambda (x) (+ x n)))
```

takes a number, and returns a closure which, when called, adds that number to its argument. We can make as many instances of adders as we want:

[1]Under dynamic scope the same idiom will work for a different reason—so long as neither of `mapcar`'s parameter is called `x`.

```
> (setq add2  (make-adder 2)
        add10 (make-adder 10))
#<Interpreted-Function BF162E>
> (funcall add2 5)
7
> (funcall add10 3)
13
```

In the closures returned by `make-adder`, the internal state is fixed, but it's also possible to make closures which can be asked to change their state.

```
(defun make-adderb (n)
  #'(lambda (x &optional change)
      (if change
          (setq n x)
          (+ x n))))
```

This new version of `make-adder` returns closures which, when called with one argument, behave just like the old ones.

```
> (setq addx (make-adderb 1))
#<Interpreted-Function BF1C66>
> (funcall addx 3)
4
```

However, when the new type of adder is called with a non-nil second argument, its internal copy of `n` will be reset to the value passed as the first argument:

```
> (funcall addx 100 t)
100
> (funcall addx 3)
103
```

It's even possible to return a group of closures which share the same data objects. Figure 2.1 contains a function which creates primitive databases. It takes an assoc-list (`db`), and returns a list of three closures which query, add, and delete entries, respectively.

Each call to `make-dbms` makes a new database—a new set of functions closed over their own shared copy of an assoc-list.

```
> (setq cities (make-dbms '((boston . us) (paris . france))))
(#<Interpreted-Function 8022E7>
 #<Interpreted-Function 802317>
 #<Interpreted-Function 802347>)
```

```
(defun make-dbms (db)
  (list
    #'(lambda (key)
        (cdr (assoc key db)))
    #'(lambda (key val)
        (push (cons key val) db)
        key)
    #'(lambda (key)
        (setf db (delete key db :key #'car))
        key)))
```

Figure 2.1: Three closures share a list.

The actual assoc-list within the database is invisible from the outside world—we can't even tell that it's an assoc-list—but it can be reached through the functions which are components of `cities`:

```
> (funcall (car cities) 'boston)
US
> (funcall (second cities) 'london 'england)
LONDON
> (funcall (car cities) 'london)
ENGLAND
```

Calling the `car` of a list is a bit ugly. In real programs, the access functions might instead be entries in a structure. Using them could also be cleaner—databases could be reached indirectly via functions like:

```
(defun lookup (key db)
  (funcall (car db) key))
```

However, the basic behavior of closures is independent of such refinements.

In real programs, the closures and data structures would also be more elaborate than those we see in `make-adder` or `make-dbms`. The single shared variable could be any number of variables, each bound to any sort of data structure.

Closures are one of the distinct, tangible benefits of Lisp. Some Lisp programs could, with effort, be translated into less powerful languages. But just try to translate a program which uses closures as above, and it will become evident how much work this abstraction is saving us. Later chapters will deal with closures in more detail. Chapter 5 shows how to use them to build compound functions, and Chapter 6 looks at their use as a substitute for traditional data structures.

2.7 Local Functions

When we define functions with lambda-expressions, we face a restriction which doesn't arise with `defun`: a function defined in a lambda-expression doesn't have a name and therefore has no way of referring to itself. This means that in Common Lisp we can't use `lambda` to define a recursive function. ○

If we want to apply some function to all the elements of a list, we use the most familiar of Lisp idioms:

```
> (mapcar #'(lambda (x) (+ 2 x))
          '(2 5 7 3))
(4 7 9 5)
```

What about cases where we want to give a recursive function as the first argument to `mapcar`? If the function has been defined with `defun`, we can simply refer to it by name:

```
> (mapcar #'copy-tree '((a b) (c d e)))
((A B) (C D E))
```

But now suppose that the function has to be a closure, taking some bindings from the environment in which the `mapcar` occurs. In our example `list+`,

```
(defun list+ (lst n)
  (mapcar #'(lambda (x) (+ x n))
          lst))
```

the first argument to `mapcar`, `#'(lambda (x) (+ x n))`, must be defined within `list+` because it needs to catch the binding of `n`. So far so good, but what if we want to give `mapcar` a function which both needs local bindings *and* is recursive? We can't use a function defined elsewhere with `defun`, because we need bindings from the local environment. And we can't use `lambda` to define a recursive function, because the function will have no way of referring to itself.

Common Lisp gives us `labels` as a way out of this dilemma. With one important reservation, `labels` could be described as a sort of `let` for functions. Each of the binding specifications in a `labels` expression should have the form

`(`⟨name⟩ ⟨parameters⟩ `.` ⟨body⟩`)`

Within the `labels` expression, ⟨name⟩ will refer to a function equivalent to:

`#'(lambda` ⟨parameters⟩ `.` ⟨body⟩`)`

So for example:

```
> (labels ((inc (x) (1+ x)))
    (inc 3))
4
```

However, there is an important difference between `let` and `labels`. In a `let` expression, the value of one variable can't depend on another variable made by the same `let`—that is, you can't say

```
(let ((x 10) (y x))
  y)
```

and expect the value of the new `y` to reflect that of the new `x`. In contrast, the body of a function *f* defined in a `labels` expression may refer to any other function defined there, including *f* itself, which makes recursive function definitions possible.

Using `labels` we can write a function analogous to `list+`, but in which the first argument to `mapcar` is a recursive function:

```
(defun count-instances (obj lsts)
  (labels ((instances-in (lst)
             (if (consp lst)
                 (+ (if (eq (car lst) obj) 1 0)
                    (instances-in (cdr lst)))
                 0)))
    (mapcar #'instances-in lsts)))
```

This function takes an object and a list, and returns a list of the number of occurrences of the object in each element:

```
> (count-instances 'a '((a b c) (d a r p a) (d a r) (a a)))
(1 2 1 2)
```

2.8 Tail-Recursion

A recursive function is one that calls itself. Such a call is *tail-recursive* if no work remains to be done in the calling function afterwards. This function is not tail-recursive

```
(defun our-length (lst)
  (if (null lst)
      0
      (1+ (our-length (cdr lst)))))
```

because on returning from the recursive call we have to pass the result to `1+`. The following function is tail-recursive, though

```
(defun our-find-if (fn lst)
  (if (funcall fn (car lst))
      (car lst)
      (our-find-if fn (cdr lst))))
```

because the value of the recursive call is immediately returned.

Tail-recursion is desirable because many Common Lisp compilers can transform tail-recursive functions into loops. With such a compiler, you can have the elegance of recursion in your source code without the overhead of function calls at runtime. The gain in speed is usually great enough that programmers go out of their way to make functions tail-recursive.

A function which isn't tail-recursive can often be transformed into one that is by embedding in it a local function which uses an *accumulator*. In this context, an accumulator is a parameter representing the value computed so far. For example, `our-length` could be transformed into

```
(defun our-length (lst)
  (labels ((rec (lst acc)
             (if (null lst)
                 acc
                 (rec (cdr lst) (1+ acc)))))
    (rec lst 0)))
```

where the number of list elements seen so far is contained in a second parameter, `acc`. When the recursion reaches the end of the list, the value of `acc` will be the total length, which can just be returned. By accumulating the value as we go down the calling tree instead of constructing it on the way back up, we can make `rec` tail-recursive.

Many Common Lisp compilers can do tail-recursion optimization, but not all of them do it by default. So after writing your functions to be tail-recursive, you may also want to put

```
(proclaim '(optimize speed))
```

at the top of the file, to ensure that the compiler can take advantage of your efforts.[2]

Given tail-recursion and type declarations, existing Common Lisp compilers can generate code that runs as fast as, or faster than, C. Richard Gabriel gives as ○ an example the following function, which returns the sum of the integers from `1` to `n`:

[2]The declaration `(optimize speed)` ought to be an abbreviation for `(optimize (speed 3))`. However, one Common Lisp implementation does tail-recursion optimization with the former, but not the latter.

```
(defun triangle (n)
  (labels ((tri (c n)
             (declare (type fixnum n c))
             (if (zerop n)
                 c
                 (tri (the fixnum (+ n c))
                      (the fixnum (- n 1))))))
    (tri 0 n)))
```

This is what fast Common Lisp code looks like. At first it may not seem natural to write functions this way. It's often a good idea to begin by writing a function in whatever way seems most natural, and then, if necessary, transforming it into a tail-recursive equivalent.

2.9 Compilation

Lisp functions can be compiled either individually or by the file. If you just type a `defun` expression into the toplevel,

```
> (defun foo (x) (1+ x))
FOO
```

many implementations will create an interpreted function. You can check whether a given function is compiled by feeding it to `compiled-function-p`:

```
> (compiled-function-p #'foo)
NIL
```

We can have `foo` compiled by giving its name to `compile`

```
> (compile 'foo)
FOO
```

which will compile the definition of `foo` and replace the interpreted version with ○ a compiled one.

```
> (compiled-function-p #'foo)
T
```

Compiled and interpreted functions are both Lisp objects, and behave the same, except with respect to `compiled-function-p`. Literal functions can also be compiled: `compile` expects its first argument to be a name, but if you give `nil` as the first argument, it will compile the lambda-expression given as the second argument.

```
> (compile nil '(lambda (x) (+ x 2)))
#<Compiled-Function BF55BE>
```

If you give both the name and function arguments, `compile` becomes a sort of compiling `defun`:

```
> (progn (compile 'bar '(lambda (x) (* x 3)))
         (compiled-function-p #'bar))
T
```

Having `compile` in the language means that a program could build and compile new functions on the fly. However, calling `compile` explicitly is a drastic measure, comparable to calling `eval`, and should be viewed with the same suspicion.[3] When Section 2.1 said that creating new functions at runtime was a routinely used programming technique, it referred to new closures like those made by `make-adder`, not functions made by calling `compile` on raw lists. Calling `compile` is not a routinely used programming technique—it's an extremely rare one. So beware of doing it unnecessarily. Unless you're implementing another language on top of Lisp (and much of the time, even then), what you need to do may be possible with macros.

There are two sorts of functions which you can't give as an argument to `compile`. According to CLTL2 (p. 677), you can't compile a function "defined interpretively in a non-null lexical environment." That is, if at the toplevel you define `foo` within a `let`

```
> (let ((y 2))
    (defun foo (x) (+ x y)))
```

then `(compile 'foo)` will not necessarily work.[4] You also can't call `compile` on a function which is already compiled. In this situation, CLTL2 hints darkly that "the consequences. . .are unspecified."

The usual way to compile Lisp code is not to compile functions individually with `compile`, but to compile whole files with `compile-file`. This function takes a filename and creates a compiled version of the source file—typically with the same base name but a different extension. When the compiled file is loaded, `compiled-function-p` should return true for all the functions defined in the file.

Later chapters will depend on another effect of compilation: when one function occurs within another function, and the containing function is compiled, the inner

[3]An explanation of why it is bad to call `eval` explicitly appears on page 278.

[4]It's ok to have this code in a file and then compile the file. The restriction is imposed on interpreted code for implementation reasons, not because there's anything wrong with defining functions in distinct lexical environments.

function will also get compiled. CLTL2 does not seem to say explicitly that this will happen, but in a decent implementation you can count on it.

The compiling of inner functions becomes evident in functions which return functions. When `make-adder` (page 18) is compiled, it will return compiled functions:

```
> (compile 'make-adder)
MAKE-ADDER
> (compiled-function-p (make-adder 2))
T
```

As later chapters will show, this fact is of great importance in the implementation of embedded languages. If a new language is implemented by transformation, and the transformation code is compiled, then it yields compiled output—and so becomes in effect a compiler for the new language. (A simple example is described on page 81.)

If we have a particularly small function, we may want to request that it be compiled inline. Otherwise, the machinery of calling it could entail more effort than the function itself. If we define a function:

```
(defun 50th (lst) (nth 49 lst))
```

and make the declaration:

```
(proclaim '(inline 50th))
```

then a reference to `50th` within a compiled function should no longer require a real function call. If we define and compile a function which calls `50th`,

```
(defun foo (lst)
  (+ (50th lst) 1))
```

then when `foo` is compiled, the code for `50th` should be compiled right into it, just as if we had written

```
(defun foo (lst)
  (+ (nth 49 lst) 1))
```

in the first place. The drawback is that if we redefine `50th`, we also have to recompile `foo`, or it will still reflect the old definition. The restrictions on inline
○ functions are basically the same as those on macros (see Section 7.9).

2.10 Functions from Lists

In some earlier dialects of Lisp, functions were represented as lists. This gave Lisp programs the remarkable ability to write and execute their own Lisp programs. In Common Lisp, functions are no longer made of lists—good implementations compile them into native machine code. But you can still write programs that write programs, because lists are the input to the compiler.

It cannot be overemphasized how important it is that Lisp programs can write Lisp programs, especially since this fact is so often overlooked. Even experienced Lisp users rarely realize the advantages they derive from this feature of the language. *This* is why Lisp macros are so powerful, for example. Most of the techniques described in this book depend on the ability to write programs which manipulate Lisp expressions.