5. Functions

After the rules of syntax and semantics, the three most basic components of all Lisp programs are functions, variables and macros. You used all three while building the database in Chapter 3, but I glossed over a lot of the details of how they work and how to best use them. I'll devote the next few chapters to these three topics, starting with functions, which--like their counterparts in other languages--provide the basic mechanism for abstracting, well, functionality.

The bulk of Lisp itself consists of functions. More than three quarters of the names defined in the language standard name functions. All the built-in data types are defined purely in terms of what functions operate on them. Even Lisp's powerful object system is built upon a conceptual extension to functions, generic functions, which I'll cover in Chapter 16.

And, despite the importance of macros to The Lisp Way, in the end all real functionality is provided by functions. Macros run at compile time, so the code they generate--the code that will actually make up the program after all the macros are expanded--will consist entirely of calls to functions and special operators. Not to mention, macros themselves are also functions, albeit functions that are used to generate code rather than to perform the actions of the program.[1]

Defining New Functions

Normally functions are defined using the **`DEFUN`** macro. The basic skeleton of a **`DEFUN`** looks like this:

```
(defun name (parameter*)
  "Optional documentation string."
  body-form*)
```

Any symbol can be used as a function name.[2] Usually function names contain only alphabetic characters and hyphens, but other characters are allowed and are used in certain naming conventions. For instance, functions that convert one kind of value to another sometimes use `->` in the name. For example, a function to convert strings to widgets might be called `string->widget`. The most important naming convention is the one mentioned in Chapter 2, which is that you construct compound names with hyphens rather than underscores or inner caps. Thus, `frob-widget` is better Lisp style than either `frob_widget` or `frobWidget`.

A function's parameter list defines the variables that will be used to hold the arguments passed to the function when it's called.[3] If the function takes no arguments, the list is empty, written as `()`.

Different flavors of parameters handle required, optional, multiple, and keyword arguments. I'll discuss the details in the next section.

If a string literal follows the parameter list, it's a documentation string that should describe the purpose of the function. When the function is defined, the documentation string will be associated with the name of the function and can later be obtained using the **DOCUMENTATION** function.[4]

Finally, the body of a **DEFUN** consists of any number of Lisp expressions. They will be evaluated in order when the function is called and the value of the last expression is returned as the value of the function. Or the **RETURN-FROM** special operator can be used to return immediately from anywhere in a function, as I'll discuss in a moment.

In Chapter 2 we wrote a `hello-world` function, which looked like this:

```
(defun hello-world () (format t "hello, world"))
```

You can now analyze the parts of this function. Its name is `hello-world`, its parameter list is empty so it takes no arguments, it has no documentation string, and its body consists of one expression.

```
(format t "hello, world")
```

The following is a slightly more complex function:

```
(defun verbose-sum (x y)
  "Sum any two numbers after printing a message."
  (format t "Summing ~d and ~d.~%" x y)
  (+ x y))
```

This function is named `verbose-sum`, takes two arguments that will be bound to the parameters `x` and `y`, has a documentation string, and has a body consisting of two expressions. The value returned by the call to **+** becomes the return value of `verbose-sum`.

Function Parameter Lists

There's not a lot more to say about function names or documentation strings, and it will take a good portion of the rest of this book to describe all the things you can do in the body of a function, which leaves us with the parameter list.

The basic purpose of a parameter list is, of course, to declare the variables that will receive the arguments passed to the function. When a parameter list is a simple list of variable names--as in `verbose-sum`--the parameters are called *required parameters*. When a function is called, it must be supplied with one argument for every required parameter. Each parameter is bound to the corresponding argument. If a function is called with too few or too many arguments, Lisp will signal an error.

However, Common Lisp's parameter lists also give you more flexible ways of mapping the arguments in a function call to the function's parameters. In addition to required parameters, a function can have optional parameters. Or a function can have a single parameter that's bound to a list containing any extra arguments. And, finally, arguments can be mapped to parameters using keywords rather than position. Thus, Common Lisp's parameter lists provide a convenient solution to several common coding problems.

Optional Parameters

While many functions, like `verbose-sum`, need only required parameters, not all functions are quite so simple. Sometimes a function will have a parameter that only certain callers will care about, perhaps because there's a reasonable default value. An example is a function that creates a data structure that can grow as needed. Since the data structure can grow, it doesn't matter--from a correctness point of view--what the initial size is. But callers who have a good idea how many items they're going to put into the data structure may be able to improve performance by specifying a specific initial size. Most callers, though, would probably rather let the code that implements the data structure pick a good general-purpose value. In Common Lisp you can accommodate both kinds of callers by using an optional parameter; callers who don't care will get a reasonable default, and other callers can provide a specific value.[5]

To define a function with optional parameters, after the names of any required parameters, place the symbol **`&optional`** followed by the names of the optional parameters. A simple example looks like this:

```
(defun foo (a b &optional c d) (list a b c d))
```

When the function is called, arguments are first bound to the required parameters. After all the required parameters have been given values, if there are any arguments left, their values are assigned to the optional parameters. If the arguments run out before the optional parameters do, the remaining optional parameters are bound to the value **`NIL`**. Thus, the function defined previously gives the following results:

```
(foo 1 2)      ==> (1 2 NIL NIL)
(foo 1 2 3)    ==> (1 2 3 NIL)
(foo 1 2 3 4) ==> (1 2 3 4)
```

Lisp will still check that an appropriate number of arguments are passed to the function--in this case between two and four, inclusive--and will signal an error if the function is called with too few or too many.

Of course, you'll often want a different default value than **`NIL`**. You can specify the default value by replacing the parameter name with a list containing a name and an expression. The expression will be evaluated only if the caller doesn't pass enough arguments to provide a value for the optional parameter. The common case is simply to provide a value as the expression.

```
(defun foo (a &optional (b 10)) (list a b))
```

This function requires one argument that will be bound to the parameter `a`. The second parameter, `b`, will take either the value of the second argument, if there is one, or 10.

```
(foo 1 2) ==> (1 2)
(foo 1)   ==> (1 10)
```

Sometimes, however, you may need more flexibility in choosing the default value. You may want to compute a default value based on other parameters. And you can--the default-value expression can refer to parameters that occur earlier in the parameter list. If you were writing a function that returned some sort of representation of a rectangle and you wanted to make it especially convenient to make squares, you might use an argument list like this:

```
(defun make-rectangle (width &optional (height width)) ...)
```

which would cause the `height` parameter to take the same value as the `width` parameter unless explicitly specified.

Occasionally, it's useful to know whether the value of an optional argument was supplied by the caller or is the default value. Rather than writing code to check whether the value of the parameter is the default (which doesn't work anyway, if the caller happens to explicitly pass the default value), you can add another variable name to the parameter specifier after the default-value expression. This variable will be bound to true if the caller actually supplied an argument for this parameter and **`NIL`** otherwise. By convention, these variables are usually named the same as the actual parameter with a "-supplied-p" on the end. For example:

```
(defun foo (a b &optional (c 3 c-supplied-p))
  (list a b c c-supplied-p))
```

This gives results like this:

```
(foo 1 2)   ==> (1 2 3 NIL)
(foo 1 2 3) ==> (1 2 3 T)
(foo 1 2 4) ==> (1 2 4 T)
```

Rest Parameters

Optional parameters are just the thing when you have discrete parameters for which the caller may or may not want to provide values. But some functions need to take a variable number of arguments. Several of the built-in functions you've seen already work this way. **`FORMAT`** has two required arguments, the stream and the control string. But after that it needs a variable number of arguments depending on how many values need to be interpolated into the control string. The **`+`** function also takes a variable number of arguments--there's no particular reason to limit it to summing just two numbers; it will sum any number of values. (It even works with zero arguments, returning 0, the identity under addition.) The following are all legal calls of those two functions:

```
(format t "hello, world")
(format t "hello, ~a" name)
(format t "x: ~d y: ~d" x y)
(+)
(+ 1)
(+ 1 2)
(+ 1 2 3)
```

Obviously, you could write functions taking a variable number of arguments by simply giving them a lot of optional parameters. But that would be incredibly painful--just writing the parameter list would be bad enough, and that doesn't get into dealing with all the parameters in the body of the function. To do it properly, you'd have to have as many optional parameters as the number of arguments that can legally be passed in a function call. This number is implementation dependent but guaranteed to be at least 50. And in current implementations it ranges from 4,096 to 536,870,911.[6] Blech. That kind of mind-bending tedium is definitely *not* The Lisp Way.

Instead, Lisp lets you include a catchall parameter after the symbol **`&rest`**. If a function includes a **`&rest`** parameter, any arguments remaining after values have been doled out to all the required and optional parameters are gathered up into a list that becomes the value of the **`&rest`** parameter. Thus, the parameter lists for **`FORMAT`** and **`+`** probably look something like this:

```
(defun format (stream string &rest values) ...)
(defun + (&rest numbers) ...)
```

Keyword Parameters

Optional and rest parameters give you quite a bit of flexibility, but neither is going to help you out much in the following situation: Suppose you have a function that takes four optional parameters. Now suppose that most of the places the function is called, the caller wants to provide a value for only one of the four parameters and, further, that the callers are evenly divided as to which parameter they will use.

The callers who want to provide a value for the first parameter are fine--they just pass the one optional argument and leave off the rest. But all the other callers have to pass some value for between one and three arguments they don't care about. Isn't that exactly the problem optional parameters were designed to solve?

Of course it is. The problem is that optional parameters are still positional--if the caller wants to pass an explicit value for the fourth optional parameter, it turns the first three optional parameters into required parameters for that caller. Luckily, another parameter flavor, keyword parameters, allow the caller to specify which values go with which parameters.

To give a function keyword parameters, after any required, **`&optional`**, and **`&rest`** parameters you include the symbol **`&key`** and then any number of keyword parameter specifiers,

which work like optional parameter specifiers. Here's a function that has only keyword parameters:

```
(defun foo (&key a b c) (list a b c))
```

When this function is called, each keyword parameters is bound to the value immediately following a keyword of the same name. Recall from Chapter 4 that keywords are names that start with a colon and that they're automatically defined as self-evaluating constants.

If a given keyword doesn't appear in the argument list, then the corresponding parameter is assigned its default value, just like an optional parameter. Because the keyword arguments are labeled, they can be passed in any order as long as they follow any required arguments. For instance, `foo` can be invoked as follows:

```
(foo)                 ==> (NIL NIL NIL)
(foo :a 1)            ==> (1 NIL NIL)
(foo :b 1)            ==> (NIL 1 NIL)
(foo :c 1)            ==> (NIL NIL 1)
(foo :a 1 :c 3)       ==> (1 NIL 3)
(foo :a 1 :b 2 :c 3) ==> (1 2 3)
(foo :a 1 :c 3 :b 2) ==> (1 2 3)
```

As with optional parameters, keyword parameters can provide a default value form and the name of a supplied-p variable. In both keyword and optional parameters, the default value form can refer to parameters that appear earlier in the parameter list.

```
(defun foo (&key (a 0) (b 0 b-supplied-p) (c (+ a b)))
  (list a b c b-supplied-p))

(foo :a 1)            ==> (1 0 1 NIL)
(foo :b 1)            ==> (0 1 1 T)
(foo :b 1 :c 4)       ==> (0 1 4 T)
(foo :a 2 :b 1 :c 4) ==> (2 1 4 T)
```

Also, if for some reason you want the keyword the caller uses to specify the parameter to be different from the name of the actual parameter, you can replace the parameter name with another list containing the keyword to use when calling the function and the name to be used for the parameter. The following definition of `foo`:

```
(defun foo (&key ((:apple a)) ((:box b) 0) ((:charlie c) 0 c-supplied-p))
  (list a b c c-supplied-p))
```

lets the caller call it like this:

```
(foo :apple 10 :box 20 :charlie 30) ==> (10 20 30 T)
```

This style is mostly useful if you want to completely decouple the public API of the function from the internal details, usually because you want to use short variable names internally but descriptive keywords in the API. It's not, however, very frequently used.

Mixing Different Parameter Types

It's possible, but rare, to use all four flavors of parameters in a single function. Whenever more than one flavor of parameter is used, they must be declared in the order I've discussed them: first the names of the required parameters, then the optional parameters, then the rest parameter, and finally the keyword parameters. Typically, however, in functions that use multiple flavors of parameters, you'll combine required parameters with one other flavor or possibly combine **`&optional`** and **`&rest`** parameters. The other two combinations, either **`&optional`** or **`&rest`** parameters combined with **`&key`** parameters, can lead to somewhat surprising behavior.

Combining **`&optional`** and **`&key`** parameters yields surprising enough results that you should probably avoid it altogether. The problem is that if a caller doesn't supply values for all the optional parameters, then those parameters will eat up the keywords and values intended for the keyword parameters. For instance, this function unwisely mixes **`&optional`** and **`&key`** parameters:

```
(defun foo (x &optional y &key z) (list x y z))
```

If called like this, it works fine:

```
(foo 1 2 :z 3) ==> (1 2 3)
```

And this is also fine:

```
(foo 1)  ==> (1 nil nil)
```

But this will signal an error:

```
(foo 1 :z 3) ==> ERROR
```

This is because the keyword `:z` is taken as a value to fill the optional `y` parameter, leaving only the argument 3 to be processed. At that point, Lisp will be expecting either a keyword/value pair or nothing and will complain. Perhaps even worse, if the function had had two **`&optional`** parameters, this last call would have resulted in the values `:z` and 3 being bound to the two **`&optional`** parameters and the **`&key`** parameter `z` getting the default value **`NIL`** with no indication that anything was amiss.

In general, if you find yourself writing a function that uses both **`&optional`** and **`&key`** parameters, you should probably just change it to use all **`&key`** parameters--they're more flexible, and you can always add new keyword parameters without disturbing existing callers of the function. You can also remove keyword parameters, as long as no one is using them.[7] In general, using keyword parameters helps make code much easier to maintain and evolve--if you need to add some new behavior to a function that requires new parameters, you can add keyword parameters without having to touch, or even recompile, any existing code that calls the function.

You can safely combine **`&rest`** and **`&key`** parameters, but the behavior may be a bit surprising initially. Normally the presence of either **`&rest`** or **`&key`** in a parameter list causes all the values remaining after the required and **`&optional`** parameters have been filled in to be

processed in a particular way--either gathered into a list for a **&rest** parameter or assigned to the appropriate **&key** parameters based on the keywords. If both **&rest** and **&key** appear in a parameter list, then both things happen--all the remaining values, which include the keywords themselves, are gathered into a list that's bound to the **&rest** parameter, and the appropriate values are also bound to the **&key** parameters. So, given this function:

```
(defun foo (&rest rest &key a b c) (list rest a b c))
```

you get this result:

```
(foo :a 1 :b 2 :c 3)  ==> ((:A 1 :B 2 :C 3) 1 2 3)
```

Function Return Values

All the functions you've written so far have used the default behavior of returning the value of the last expression evaluated as their own return value. This is the most common way to return a value from a function.

However, sometimes it's convenient to be able to return from the middle of a function such as when you want to break out of nested control constructs. In such cases you can use the **RETURN-FROM** special operator to immediately return any value from the function.

You'll see in Chapter 20 that **RETURN-FROM** is actually not tied to functions at all; it's used to return from a block of code defined with the **BLOCK** special operator. However, **DEFUN** automatically wraps the whole function body in a block with the same name as the function. So, evaluating a **RETURN-FROM** with the name of the function and the value you want to return will cause the function to immediately exit with that value. **RETURN-FROM** is a special operator whose first "argument" is the name of the block from which to return. This name isn't evaluated and thus isn't quoted.

The following function uses nested loops to find the first pair of numbers, each less than 10, whose product is greater than the argument, and it uses **RETURN-FROM** to return the pair as soon as it finds it:

```
(defun foo (n)
  (dotimes (i 10)
    (dotimes (j 10)
      (when (> (* i j) n)
        (return-from foo (list i j))))))
```

Admittedly, having to specify the name of the function you're returning from is a bit of a pain--for one thing, if you change the function's name, you'll need to change the name used in the **RETURN-FROM** as well.[8] But it's also the case that explicit **RETURN-FROM**s are used much less frequently in Lisp than `return` statements in C-derived languages, because *all* Lisp expressions, including control constructs such as loops and conditionals, evaluate to a value. So it's not much of a problem in practice.

Functions As Data, a.k.a. Higher-Order Functions

While the main way you use functions is to call them by name, a number of situations exist where it's useful to be able treat functions as data. For instance, if you can pass one function as an argument to another, you can write a general-purpose sorting function while allowing the caller to provide a function that's responsible for comparing any two elements. Then the same underlying algorithm can be used with many different comparison functions. Similarly, callbacks and hooks depend on being able to store references to code in order to run it later. Since functions are already the standard way to abstract bits of code, it makes sense to allow functions to be treated as data.[9]

In Lisp, functions are just another kind of object. When you define a function with **DEFUN**, you're really doing two things: creating a new function object and giving it a name. It's also possible, as you saw in Chapter 3, to use **LAMBDA** expressions to create a function without giving it a name. The actual representation of a function object, whether named or anonymous, is opaque--in a native-compiling Lisp, it probably consists mostly of machine code. The only things you need to know are how to get hold of it and how to invoke it once you've got it.

The special operator **FUNCTION** provides the mechanism for getting at a function object. It takes a single argument and returns the function with that name. The name isn't quoted. Thus, if you've defined a function `foo`, like so:

```
CL-USER> (defun foo (x) (* 2 x))
FOO
```

you can get the function object like this:[10]

```
CL-USER> (function foo)
#<Interpreted Function FOO>
```

In fact, you've already used **FUNCTION**, but it was in disguise. The syntax `#'`, which you used in Chapter 3, is syntactic sugar for **FUNCTION**, just the way `'` is syntactic sugar for **QUOTE**.[11] Thus, you can also get the function object for `foo` like this:

```
CL-USER> #'foo
#<Interpreted Function FOO>
```

Once you've got the function object, there's really only one thing you can do with it--invoke it. Common Lisp provides two functions for invoking a function through a function object: **FUNCALL** and **APPLY**.[12] They differ only in how they obtain the arguments to pass to the function.

FUNCALL is the one to use when you know the number of arguments you're going to pass to the function at the time you write the code. The first argument to **FUNCALL** is the function object to be invoked, and the rest of the arguments are passed onto that function. Thus, the following two expressions are equivalent:

```
(foo 1 2 3) === (funcall #'foo 1 2 3)
```

However, there's little point in using **FUNCALL** to call a function whose name you know when you write the code. In fact, the previous two expressions will quite likely compile to exactly the same machine instructions.

The following function demonstrates a more apt use of **FUNCALL**. It accepts a function object as an argument and plots a simple ASCII-art histogram of the values returned by the argument function when it's invoked on the values from `min` to `max`, stepping by `step`.

```
(defun plot (fn min max step)
  (loop for i from min to max by step do
        (loop repeat (funcall fn i) do (format t "*"))
        (format t "~%")))
```

The **FUNCALL** expression computes the value of the function for each value of `i`. The inner **LOOP** uses that computed value to determine how many times to print an asterisk to standard output.

Note that you don't use **FUNCTION** or `#'` to get the function value of `fn`; you *want* it to be interpreted as a variable because it's the variable's value that will be the function object. You can call `plot` with any function that takes a single numeric argument, such as the built-in function **EXP** that returns the value of *e* raised to the power of its argument.

```
CL-USER> (plot #'exp 0 4 1/2)
*
*
**
****
*******
************
********************
*********************************
**************************************************
NIL
```

FUNCALL, however, doesn't do you any good when the argument list is known only at runtime. For instance, to stick with the `plot` function for another moment, suppose you've obtained a list containing a function object, a minimum and maximum value, and a step value. In other words, the list contains the values you want to pass as arguments to `plot`. Suppose this list is in the variable `plot-data`. You could invoke `plot` on the values in that list like this:

```
(plot (first plot-data) (second plot-data) (third plot-data) (fourth plot-data))
```

This works fine, but it's pretty annoying to have to explicitly unpack the arguments just so you can pass them to `plot`.

That's where **APPLY** comes in. Like **FUNCALL**, the first argument to **APPLY** is a function object. But after the function object, instead of individual arguments, it expects a list. It then applies the function to the values in the list. This allows you to write the following instead:

```
(apply #'plot plot-data)
```

As a further convenience, **APPLY** can also accept "loose" arguments as long as the last argument is a list. Thus, if `plot-data` contained just the min, max, and step values, you could still use **APPLY** like this to plot the **EXP** function over that range:

```
(apply #'plot #'exp plot-data)
```

APPLY doesn't care about whether the function being applied takes **&optional**, **&rest**, or **&key** arguments--the argument list produced by combining any loose arguments with the final list must be a legal argument list for the function with enough arguments for all the required parameters and only appropriate keyword parameters.

Anonymous Functions

Once you start writing, or even simply using, functions that accept other functions as arguments, you're bound to discover that sometimes it's annoying to have to define and name a whole separate function that's used in only one place, especially when you never call it by name.

When it seems like overkill to define a new function with **DEFUN**, you can create an "anonymous" function using a **LAMBDA** expression. As discussed in Chapter 3, a **LAMBDA** expression looks like this:

```
(lambda (parameters) body)
```

One way to think of **LAMBDA** expressions is as a special kind of function name where the name itself directly describes what the function does. This explains why you can use a **LAMBDA** expression in the place of a function name with `#'`.

```
(funcall #'(lambda (x y) (+ x y)) 2 3) ==> 5
```

You can even use a **LAMBDA** expression as the "name" of a function in a function call expression. If you wanted, you could write the previous **FUNCALL** expression more concisely.

```
((lambda (x y) (+ x y)) 2 3) ==> 5
```

But this is almost never done; it's merely worth noting that it's legal in order to emphasize that **LAMBDA** expressions can be used anywhere a normal function name can be.[13]

Anonymous functions can be useful when you need to pass a function as an argument to another function and the function you need to pass is simple enough to express inline. For instance, suppose you wanted to plot the function *2x*. You *could* define the following function:

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

which you could then pass to `plot`.

```
CL-USER> (plot #'double 0 10 1)

**
****
```

```
******
********
**********
************
**************
****************
******************
********************
NIL
```

But it's easier, and arguably clearer, to write this:

```
CL-USER> (plot #'(lambda (x) (* 2 x)) 0 10 1)

**
****
******
********
**********
************
**************
****************
******************
********************
NIL
```

The other important use of **`LAMBDA`** expressions is in making *closures*, functions that capture part of the environment where they're created. You used closures a bit in Chapter 3, but the details of how closures work and what they're used for is really more about how variables work than functions, so I'll save that discussion for the next chapter.

---

[1]Despite the importance of functions in Common Lisp, it isn't really accurate to describe it as a *functional* language. It's true some of Common Lisp's features, such as its list manipulation functions, are designed to be used in a body-form* style and that Lisp has a prominent place in the history of functional programming--McCarthy introduced many ideas that are now considered important in functional programming--but Common Lisp was intentionally designed to support many different styles of programming. In the Lisp family, Scheme is the nearest thing to a "pure" functional language, and even it has several features that disqualify it from absolute purity compared to languages such as Haskell and ML.

[2]Well, almost any symbol. It's undefined what happens if you use any of the names defined in the language standard as a name for one of your own functions. However, as you'll see in Chapter 21, the Lisp package system allows you to create names in different namespaces, so this isn't really an issue.

[3]Parameter lists are sometimes also called *lambda lists* because of the historical relationship between Lisp's notion of functions and the lambda calculus.

[4]For example, the following:

```
(documentation 'foo 'function)
```

returns the documentation string for the function `foo`. Note, however, that documentation strings are intended for human consumption, not programmatic access. A Lisp implementation isn't *required* to store them and is allowed to discard them at any time, so portable programs shouldn't depend on their presence. In some implementations an implementation-defined variable needs to be set before it will store documentation strings.

[5]In languages that don't support optional parameters directly, programmers typically find ways to simulate them. One technique is to use distinguished "no-value" values that the caller can pass to indicate they want the default value of a given parameter. In C, for example, it's common to use `NULL` as such a distinguished value. However, such a protocol between the function and its callers is ad hoc--in some functions or for some arguments `NULL` may be the distinguished value while in other functions or for other arguments the magic value may be -1 or some `#defined` constant.

[6]The constant **CALL-ARGUMENTS-LIMIT** tells you the implementation-specific value.

[7]Four standard functions take both **&optional** and **&key** arguments--**READ-FROM-STRING**, **PARSE-NAMESTRING**, **WRITE-LINE**, and **WRITE-STRING**. They were left that way during standardization for backward compatibility with earlier Lisp dialects. **READ-FROM-STRING** tends to be the one that catches new Lisp programmers most frequently--a call such as `(read-from-string s :start 10)` seems to ignore the `:start` keyword argument, reading from index 0 instead of 10. That's because **READ-FROM-STRING** also has two **&optional** parameters that swallowed up the arguments `:start` and 10.

[8]Another macro, **RETURN**, doesn't require a name. However, you can't use it instead of **RETURN-FROM** to avoid having to specify the function name; it's syntactic sugar for returning from a block named **NIL**. I'll cover it, along with the details of **BLOCK** and **RETURN-FROM**, in Chapter 20.

[9]Lisp, of course, isn't the only language to treat functions as data. C uses function pointers, Perl uses subroutine references, Python uses a scheme similar to Lisp, and C# introduces delegates, essentially typed function pointers, as an improvement over Java's rather clunky reflection and anonymous class mechanisms.

[10]The exact printed representation of a function object will differ from implementation to implementation.

[11]The best way to think of **FUNCTION** is as a special kind of quotation. **QUOTE**ing a symbol prevents it from being evaluated at all, resulting in the symbol itself rather than the value of the variable named by that symbol. **FUNCTION** also circumvents the normal evaluation rule but, instead of preventing the symbol from being evaluated at all, causes it to be evaluated as the name of a function, just the way it would if it were used as the function name in a function call expression.

[12]There's actually a third, the special operator **MULTIPLE-VALUE-CALL**, but I'll save that for when I discuss expressions that return multiple values in Chapter 20.

[13]In Common Lisp it's also possible to use a **LAMBDA** expression as an argument to **FUNCALL** (or some other function that takes a function argument such as **SORT** or **MAPCAR**) with no `#'` before it, like this:

```
(funcall (lambda (x y) (+ x y)) 2 3)
```

This is legal and is equivalent to the version with the `#'` but for a tricky reason. Historically **LAMBDA** expressions by themselves weren't expressions that could be evaluated. That is **LAMBDA** wasn't the name of a function, macro, or special operator. Rather, a list starting with the symbol **LAMBDA** was a special syntactic construct that Lisp recognized as a kind of function name.

But if that were still true, then `(funcall (lambda (...) ...))` would be illegal because **FUNCALL** is a function and the normal evaluation rule for a function call would require that the **LAMBDA** expression be evaluated. However, late in the ANSI standardization process, in order to make it possible to implement ISLISP, another Lisp dialect being standardized at the same time, strictly as a user-level compatibility layer on top of Common Lisp, a **LAMBDA** macro was defined that expands into a call to **FUNCTION** wrapped around the **LAMBDA** expression. In other words, the following **LAMBDA** expression:

```
(lambda () 42)
```

exands into the following when it occurs in a context where it evaluated:

```
(function (lambda () 42))    ; or #'(lambda () 42)
```

This makes its use in a value position, such as an argument to **FUNCALL**, legal. In other words, it's pure syntactic sugar. Most folks either always use `#'` before **LAMBDA** expressions in value positions or never do. In this book, I always use `#'`.