20. The Special Operators

In a way, the most impressive aspect of the condition system covered in the previous chapter is that if it wasn't already part of the language, it could be written entirely as a user-level library. This is possible because Common Lisp's special operators--while none touches directly on signaling or handling conditions--provide enough access to the underlying machinery of the language to be able to do things such as control the unwinding of the stack.

In previous chapters I've discussed the most frequently used special operators, but it's worth being familiar with the others for two reasons. First, some of the infrequently used special operators are used infrequently simply because whatever need they address doesn't arise that often. It's good to be familiar with these special operators so when one of them is called for, you'll at least know it exists. Second, because the 25 special operators--along with the basic rule for evaluating function calls and the built-in data types--provide the foundation for the rest of the language, a passing familiarity with them will help you understand how the language works.

In this chapter, I'll discuss all the special operators, some briefly and some at length, so you can see how they fit together. I'll point out which ones you can expect to use directly in your own code, which ones serve as the basis for other constructs that you use all the time, and which ones you'll rarely use directly but which can be handy in macro-generated code.

Controlling Evaluation

The first category of special operators contains the three operators that provide basic control over the evaluation of forms. They're **`QUOTE`**, **`IF`**, and **`PROGN`**, and I've discussed them all already. However, it's worth noting how each of these special operators provides one fundamental kind of control over the evaluation of one or more forms. **`QUOTE`** prevents evaluation altogether and allows you to get at s-expressions as data. **`IF`** provides the fundamental boolean choice operation from which all other conditional execution constructs can be built.[1] And **`PROGN`** provides the ability to sequence a number of forms.

Manipulating the Lexical Environment

The largest class of special operators contains the operators that manipulate and access the *lexical environment*. **`LET`** and **`LET*`**, which I've already discussed, are examples of special operators that manipulate the lexical environment since they can introduce new lexical bindings for variables. Any construct, such as a **`DO`** or **`DOTIMES`**, that binds lexical variables will have to

expand into a **LET** or **LET*.**[2] The **SETQ** special operator is one that accesses the lexical environment since it can be used to set variables whose bindings were created by **LET** and **LET***.

Variables, however, aren't the only thing that can be named within a lexical scope. While most functions are defined globally with **DEFUN**, it's also possible to create local functions with the special operators **FLET** and **LABELS**, local macros with **MACROLET**, and a special kind of macro, called a *symbol macro*, with **SYMBOL-MACROLET**.

Much like **LET** allows you to introduce a lexical variable whose scope is the body of the **LET**, **FLET** and **LABELS** let you define a function that can be referred to only within the scope of the **FLET** or **LABELS** form. These special operators are handy when you need a local function that's a bit too complex to define inline as a **LAMBDA** expression or that you need to use more than once. Both have the same basic form, which looks like this:

```
(flet (function-definition*)
  body-form*)
```

and like this:

```
(labels (function-definition*)
  body-form*)
```

where each *function-definition* has the following form:

```
(name (parameter*) form*)
```

The difference between **FLET** and **LABELS** is that the names of the functions defined with **FLET** can be used only in the body of the **FLET**, while the names introduced by **LABELS** can be used immediately, including in the bodies of the functions defined by the **LABELS**. Thus, **LABELS** can define recursive functions, while **FLET** can't. It might seem limiting that **FLET** can't be used to define recursive functions, but Common Lisp provides both **FLET** and **LABELS** because sometimes it's useful to be able to write local functions that can call another function of the same name, either a globally defined function or a local function from an enclosing scope.

Within the body of a **FLET** or **LABELS**, you can use the names of the functions defined just like any other function, including with the **FUNCTION** special operator. Since you can use **FUNCTION** to get the function object representing a function defined with **FLET** or **LABELS**, and since a **FLET** or **LABELS** can be in the scope of other binding forms such as **LET**s, these functions can be closures.

Because the local functions can refer to variables from the enclosing scope, they can often be written to take fewer parameters than the equivalent helper functions. This is particularly handy when you need to pass a function that takes a single argument as a functional parameter. For example, in the following function, which you'll see again in Chapter 25, the **FLET**ed function,

`count-version`, takes a single argument, as required by `walk-directory`, but can also use the variable `versions`, introduced by the enclosing **`LET`**:

```
(defun count-versions (dir)
  (let ((versions (mapcar #'(lambda (x) (cons x 0)) '(2 3 4))))
    (flet ((count-version (file)
             (incf (cdr (assoc (major-version (read-id3 file)) versions)))))
      (walk-directory dir #'count-version :test #'mp3-p))
    versions))
```

This function could also be written using an anonymous function in the place of the **`FLET`**ed `count-version`, but giving the function a meaningful name makes it a bit easier to read.

And when a helper function needs to recurse, an anonymous function just won't do.[3] When you don't want to define a recursive helper function as a global function, you can use **`LABELS`**. For example, the following function, `collect-leaves`, uses the recursive helper function `walk` to walk a tree and gather all the atoms in the tree into a list, which `collect-leaves` then returns (after reversing it):

```
(defun collect-leaves (tree)
  (let ((leaves ()))
    (labels ((walk (tree)
               (cond
                 ((null tree))
                 ((atom tree) (push tree leaves))
                 (t (walk (car tree))
                    (walk (cdr tree))))))
      (walk tree))
    (nreverse leaves)))
```

Notice again how, within the `walk` function, you can refer to the variable, `leaves`, introduced by the enclosing **`LET`**.

`FLET` and **`LABELS`** are also useful operations to use in macro expansions--a macro can expand into code that contains a **`FLET`** or **`LABELS`** to create functions that can be used within the body of the macro. This technique can be used either to introduce functions that the user of the macro will call or simply as a way of organizing the code generated by the macro. This, for instance, is how a function such as **`CALL-NEXT-METHOD`**, which can be used only within a method definition, might be defined.

A near relative to **`FLET`** and **`LABELS`** is the special operator **`MACROLET`**, which you can use to define local macros. Local macros work just like global macros defined with **`DEFMACRO`** except without cluttering the global namespace. When a **`MACROLET`** form is evaluated, the body forms are evaluated with the local macro definitions in effect and possibly shadowing global function and macro definitions or local definitions from enclosing forms. Like **`FLET`** and **`LABELS`**, **`MACROLET`** can be used directly, but it's also a handy target for macro-generated code--by wrapping some user-supplied code in a **`MACROLET`**, a macro can provide constructs that can be used only within that code or can shadow a globally defined macro. You'll see an example of this latter use of **`MACROLET`** in Chapter 31.

Finally, one last macro-defining special operator is **SYMBOL-MACROLET**, which defines a special kind of macro called, appropriately enough, a *symbol macro*. Symbol macros are like regular macros except they can't take arguments and are referred to with a plain symbol rather than a list form. In other words, after you've defined a symbol macro with a particular name, any use of that symbol in a value position will be expanded and the resulting form evaluated in its place. This is how macros such as **WITH-SLOTS** and **WITH-ACCESSORS** are able to define "variables" that access the state of a particular object under the covers. For instance, the following **WITH-SLOTS** form:

```
(with-slots (x y z) foo (list x y z)))
```

might expand into this code that uses **SYMBOL-MACROLET**:

```
(let ((#:g149 foo))
  (symbol-macrolet
      ((x (slot-value #:g149 'x))
       (y (slot-value #:g149 'y))
       (z (slot-value #:g149 'z)))
    (list x y z)))
```

When the expression `(list x y z)` is evaluated, the symbols `x`, `y`, and `z` will be replaced with their expansions, such as `(slot-value #:g149 'x)`.[4]

Symbol macros are most often local, defined with **SYMBOL-MACROLET**, but Common Lisp also provides a macro **DEFINE-SYMBOL-MACRO** that defines a global symbol macro. A symbol macro defined with **SYMBOL-MACROLET** shadows other symbol macros of the same name defined with **DEFINE-SYMBOL-MACRO** or enclosing **SYMBOL-MACROLET** forms.

Local Flow of Control

The next four special operators I'll discuss also create and use names in the lexical environment but for the purposes of altering the flow of control rather than defining new functions and macros. I've mentioned all four of these special operators in passing because they provide the underlying mechanisms used by other language features. They're **BLOCK**, **RETURN-FROM**, **TAGBODY**, and **GO**. The first two, **BLOCK** and **RETURN-FROM**, are used together to write code that returns immediately from a section of code--I discussed **RETURN-FROM** in Chapter 5 as a way to return immediately from a function, but it's more general than that. The other two, **TAGBODY** and **GO**, provide a quite low-level goto construct that's the basis for all the higher-level looping constructs you've already seen.

The basic skeleton of a **BLOCK** form is this:

```
(block name
  form*)
```

The *name* is a symbol, and the *forms* are Lisp forms. The forms are evaluated in order, and the value of the last form is returned as the value of the **BLOCK** unless a **RETURN-FROM** is used to

return from the block early. A **RETURN-FROM** form, as you saw in Chapter 5, consists of the name of the block to return from and, optionally, a form that provides a value to return. When a **RETURN-FROM** is evaluated, it causes the named **BLOCK** to return immediately. If **RETURN-FROM** is called with a return value form, the **BLOCK** will return the resulting value; otherwise, the **BLOCK** evaluates to **NIL**.

A **BLOCK** name can be any symbol, which includes **NIL**. Many of the standard control construct macros, such as **DO**, **DOTIMES**, and **DOLIST**, generate an expansion consisting of a **BLOCK** named **NIL**. This allows you to use the **RETURN** macro, which is a bit of syntactic sugar for `(return-from nil ...)`, to break out of such loops. Thus, the following loop will print at most ten random numbers, stopping as soon as it gets a number greater than 50:

```
(dotimes (i 10)
  (let ((answer (random 100)))
    (print answer)
    (if (> answer 50) (return))))
```

Function-defining macros such as **DEFUN**, **FLET**, and **LABELS**, on the other hand, wrap their bodies in a **BLOCK** with the same name as the function. That's why you can use **RETURN-FROM** to return from a function.

TAGBODY and **GO** have a similar relationship to each other as **BLOCK** and **RETURN-FROM**: a **TAGBODY** form defines a context in which names are defined that can be used by **GO**. The skeleton of a **TAGBODY** is as follows:

```
(tagbody
  tag-or-compound-form*)
```

where each *tag-or-compound-form* is either a symbol, called a *tag*, or a nonempty list form. The list forms are evaluated in order and the tags ignored, except as I'll discuss in a moment. After the last form of the **TAGBODY** is evaluated, the **TAGBODY** returns **NIL**. Anywhere within the lexical scope of the **TAGBODY** you can use the **GO** special operator to jump immediately to any of the tags, and evaluation will resume with the form following the tag. For instance, you can write a trivial infinite loop with **TAGBODY** and **GO** like this:

```
(tagbody
 top
   (print 'hello)
   (go top))
```

Note that while the tag names must appear at the top level of the **TAGBODY**, not nested within other forms, the **GO** special operator can appear anywhere within the scope of the **TAGBODY**. This means you could write a loop that loops a random number of times like this:

```
(tagbody
 top
   (print 'hello)
   (when (plusp (random 10)) (go top)))
```

An even sillier example of **TAGBODY**, which shows you can have multiple tags in a single **TAGBODY**, looks like this:

```
(tagbody
 a (print 'a) (if (zerop (random 2)) (go c))
 b (print 'b) (if (zerop (random 2)) (go a))
 c (print 'c) (if (zerop (random 2)) (go b)))
```

This form will jump around randomly printing *a*s, *b*s, and *c*s until eventually the last **RANDOM** expression returns 1 and the control falls off the end of the **TAGBODY**.

TAGBODY is rarely used directly since it's almost always easier to write iterative constructs in terms of the existing looping macros. It's handy, however, for translating algorithms written in other languages into Common Lisp, either automatically or manually. An example of an automatic translation tool is the FORTRAN-to-Common Lisp translator, f2cl, that translates FORTRAN source code into Common Lisp in order to make various FORTRAN libraries available to Common Lisp programmers. Since many FORTRAN libraries were written before the structured programming revolution, they're full of gotos. The f2cl compiler can simply translate those gotos to **GO**s within appropriate **TAGBODY**s.[5]

Similarly, **TAGBODY** and **GO** can be handy when translating algorithms described in prose or by flowcharts--for instance, in Donald Knuth's classic series *The Art of Computer Programming*, he describes algorithms using a "recipe" format: step 1, do this; step 2, do that; step 3, go back to step 2; and so on. For example, on page 142 of *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, Third Edition (Addison-Wesley, 1998), he describes Algorithm S, which you'll use in Chapter 27, in this form:

> Algorithm S (Selection sampling technique). To select n records at random from a set of N, where 0 < n <= N.
>
> S1. [Initialize.] Set t <-- 0, m <-- 0. (During this algorithm, m represents the number of records selected so far, and t is the total number of input records that we have dealt with.)
>
> S2. [Generate U.] Generate a random number U, uniformly distributed between zero and one.
>
> S3. [Test.] If (N - t)U >= n - m, go to step S5.
>
> S4. [Select.] Select the next record for the sample, and increase m and t by 1. If m < n, go to step S2; otherwise the sample is complete and the algorithm terminates.
>
> S5. [Skip.] Skip the next record (do not include it in the sample), increase t by 1, and go back to step S2.

This description can be easily translated into a Common Lisp function, after renaming a few variables, as follows:

```
(defun algorithm-s (n max) ; max is N in Knuth's algorithm
  (let (seen               ; t in Knuth's algorithm
        selected           ; m in Knuth's algorithm
```

```
        u                  ; U in Knuth's algorithm
        (records ()))      ; the list where we save the records selected
    (tagbody
     s1
       (setf seen 0)
       (setf selected 0)
     s2
       (setf u (random 1.0))
     s3
       (when (>= (* (- max seen) u) (- n selected)) (go s5))
     s4
       (push seen records)
       (incf selected)
       (incf seen)
       (if (< selected n)
           (go s2)
           (return-from algorithm-s (nreverse records)))
     s5
       (incf seen)
       (go s2))))
```

It's not the prettiest code, but it's easy to verify that it's a faithful translation of Knuth's algorithm. But, this code, unlike Knuth's prose description, can be run and tested. Then you can start refactoring, checking after each change that the function still works.[6]

After pushing the pieces around a bit, you might end up with something like this:

```
(defun algorithm-s (n max)
  (loop for seen from 0
     when (< (* (- max seen) (random 1.0)) n)
     collect seen and do (decf n)
     until (zerop n)))
```

While it may not be immediately obvious that this code correctly implements Algorithm S, if you got here via a series of functions that all behave identically to the original literal translation of Knuth's recipe, you'd have good reason to believe it's correct.

Unwinding the Stack

Another aspect of the language that special operators give you control over is the behavior of the call stack. For instance, while you normally use **`BLOCK`** and **`TAGBODY`** to manage the flow of control within a single function, you can also use them, in conjunction with closures, to force an immediate nonlocal return from a function further down on the stack. That's because **`BLOCK`** names and **`TAGBODY`** tags can be closed over by any code within the lexical scope of the **`BLOCK`** or **`TAGBODY`**. For example, consider this function:

```
(defun foo ()
  (format t "Entering foo~%")
  (block a
    (format t " Entering BLOCK~%")
    (bar #'(lambda () (return-from a)))
    (format t " Leaving BLOCK~%"))
  (format t "Leaving foo~%"))
```

The anonymous function passed to `bar` uses **`RETURN-FROM`** to return from the **`BLOCK`**. But that **`RETURN-FROM`** doesn't get evaluated until the anonymous function is invoked with

FUNCALL or **APPLY**. Now suppose `bar` looks like this:

```
(defun bar (fn)
  (format t "  Entering bar~%")
  (baz fn)
  (format t "  Leaving bar~%"))
```

Still, the anonymous function isn't invoked. Now look at `baz`.

```
(defun baz (fn)
  (format t "   Entering baz~%")
  (funcall fn)
  (format t "   Leaving baz~%"))
```

Finally the function is invoked. But what does it mean to **RETURN-FROM** a block that's several layers up on the call stack? Turns out it works fine--the stack is unwound back to the frame where the **BLOCK** was established and control returns from the **BLOCK**. The **FORMAT** expressions in `foo`, `bar`, and `baz` show this:

```
CL-USER> (foo)
Entering foo
 Entering BLOCK
  Entering bar
   Entering baz
Leaving foo
NIL
```

Note that the only "Leaving . . ." message that prints is the one that appears after the **BLOCK** in `foo`.

Because the names of blocks are lexically scoped, a **RETURN-FROM** always returns from the smallest enclosing **BLOCK** in the lexical environment where the **RETURN-FROM** form appears even if the **RETURN-FROM** is executed in a different dynamic context. For instance, `bar` could also contain a **BLOCK** named `a`, like this:

```
(defun bar (fn)
  (format t "  Entering bar~%")
  (block a (baz fn))
  (format t "  Leaving bar~%"))
```

This extra **BLOCK** won't change the behavior of `foo` at all--the name `a` is resolved lexically, at compile time, not dynamically, so the intervening block has no effect on the **RETURN-FROM**. Conversely, the name of a **BLOCK** can be used only by **RETURN-FROM**s appearing within the lexical scope of the **BLOCK**; there's no way for code outside the block to return from the block except by invoking a closure that closes over a **RETURN-FROM** from the lexical scope of the **BLOCK**.

TAGBODY and **GO** work the same way, in this regard, as **BLOCK** and **RETURN-FROM**. When you invoke a closure that contains a **GO** form, if the **GO** is evaluated, the stack will unwind back to the appropriate **TAGBODY** and then jump to the specified tag.

BLOCK names and **TAGBODY** tags, however, differ from lexical variable bindings in one important way. As I discussed in Chapter 6, lexical bindings have indefinite extent, meaning the bindings can stick around even after the binding form has returned. **BLOCK**s and **TAGBODY**s, on the other hand, have dynamic extent--you can **RETURN-FROM** a **BLOCK** or **GO** to a **TAGBODY** tag only while the **BLOCK** or **TAGBODY** is on the call stack. In other words, a closure that captures a block name or **TAGBODY** tag can be passed *down* the stack to be invoked later, but it can't be returned *up* the stack. If you invoke a closure that tries to **RETURN-FROM** a **BLOCK**, after the **BLOCK** itself has returned, you'll get an error. Likewise, trying to **GO** to a **TAGBODY** that no longer exists will cause an error.[7]

It's unlikely you'll need to use **BLOCK** and **TAGBODY** yourself for this kind of stack unwinding. But you'll likely be using them indirectly whenever you use the condition system, so understanding how they work should help you understand better what exactly, for instance, invoking a restart is doing.[8]

CATCH and **THROW** are another pair of special operators that can force the stack to unwind. You'll use these operators even less often than the others mentioned so far--they're holdovers from earlier Lisp dialects that didn't have Common Lisp's condition system. They definitely shouldn't be confused with `try/catch` and `try/except` constructs from languages such as Java and Python.

CATCH and **THROW** are the dynamic counterparts of **BLOCK** and **RETURN-FROM**. That is, you wrap **CATCH** around a body of code and then use **THROW** to cause the **CATCH** form to return immediately with a specified value. The difference is that the association between a **CATCH** and **THROW** is established dynamically--instead of a lexically scoped name, the label for a **CATCH** is an object, called a *catch tag*, and any **THROW** evaluated within the dynamic extent of the **CATCH** that throws that object will unwind the stack back to the **CATCH** form and cause it to return immediately. Thus, you can write a version of the `foo`, `bar`, and `baz` functions from before using **CATCH** and **THROW** instead of **BLOCK** and **RETURN-FROM** like this:

```
(defparameter *obj* (cons nil nil)) ; i.e. some arbitrary object

(defun foo ()
  (format t "Entering foo~%")
  (catch *obj*
    (format t " Entering CATCH~%")
    (bar)
    (format t " Leaving CATCH~%"))
  (format t "Leaving foo~%"))

(defun bar ()
  (format t "  Entering bar~%")
  (baz)
  (format t "  Leaving bar~%"))

(defun baz ()
  (format t "   Entering baz~%")
  (throw *obj* nil)
  (format t "   Leaving baz~%"))
```

Notice how it isn't necessary to pass a closure down the stack--`baz` can call **`THROW`** directly. The result is quite similar to the earlier version.

```
CL-USER> (foo)
Entering foo
 Entering CATCH
  Entering bar
   Entering baz
Leaving foo
NIL
```

However, **`CATCH`** and **`THROW`** are almost *too* dynamic. In both the **`CATCH`** and the **`THROW`**, the tag form is evaluated, which means their values are both determined at runtime. Thus, if some code in `bar` reassigned or rebound `*obj*`, the **`THROW`** in `baz` wouldn't throw to the same **`CATCH`**. This makes **`CATCH`** and **`THROW`** much harder to reason about than **`BLOCK`** and **`RETURN-FROM`**. The only advantage, which the version of `foo`, `bar`, and `baz` that use **`CATCH`** and **`THROW`** demonstrates, is there's no need to pass down a closure in order for low-level code to return from a **`CATCH`**--any code that runs within the dynamic extent of a **`CATCH`** can cause it to return by throwing the right object.

In older Lisp dialects that didn't have anything like Common Lisp's condition system, **`CATCH`** and **`THROW`** were used for error handling. However, to keep them manageable, the catch tags were usually just quoted symbols, so you *could* tell by looking at a **`CATCH`** and a **`THROW`** whether they would hook up at runtime. In Common Lisp you'll rarely have any call to use **`CATCH`** and **`THROW`** since the condition system is so much more flexible.

The last special operator related to controlling the stack is another one I've mentioned in passing before--**`UNWIND-PROTECT`**. **`UNWIND-PROTECT`** lets you control what happens as the stack unwinds--to make sure that certain code always runs regardless of how control leaves the scope of the **`UNWIND-PROTECT`**, whether by a normal return, by a restart being invoked, or by any of the ways discussed in this section.[9] The basic skeleton of **`UNWIND-PROTECT`** looks like this:

```
(unwind-protect protected-form
  cleanup-form*)
```

The single *protected-form* is evaluated, and then, regardless of how it returns, the *cleanup-forms* are evaluated. If the *protected-form* returns normally, then whatever it returns is returned from the **`UNWIND-PROTECT`** after the cleanup forms run. The cleanup forms are evaluated in the same dynamic environment as the **`UNWIND-PROTECT`**, so the same dynamic variable bindings, restarts, and condition handlers will be visible to code in cleanup forms as were visible just before the **`UNWIND-PROTECT`**.

You'll occasionally use **`UNWIND-PROTECT`** directly. More often you'll use it as the basis for `WITH-` style macros, similar to **`WITH-OPEN-FILE`**, that evaluate any number of body forms in a context where they have access to some resource that needs to be cleaned up after they're done, regardless of whether they return normally or bail via a restart or other nonlocal exit. For

example, if you were writing a database library that defined functions `open-connection` and `close-connection`, you might write a macro like this:[10]

```
(defmacro with-database-connection ((var &rest open-args) &body body)
  `(let ((,var (open-connection ,@open-args)))
     (unwind-protect (progn ,@body)
       (close-connection ,var))))
```

which lets you write code like this:

```
(with-database-connection (conn :host "foo" :user "scott" :password "tiger")
  (do-stuff conn)
  (do-more-stuff conn))
```

and not have to worry about closing the database connection, since the **`UNWIND-PROTECT`** will make sure it gets closed no matter what happens in the body of the `with-database-connection` form.

Multiple Values

Another feature of Common Lisp that I've mentioned in passing--in Chapter 11, when I discussed **`GETHASH`**--is the ability for a single form to return multiple values. I'll discuss it in greater detail now. It is, however, slightly misplaced in a chapter on special operators since the ability to return multiple values isn't provided by just one or two special operators but is deeply integrated into the language. The operators you'll most often use when dealing with multiple values are macros and functions, not special operators. But it is the case that the basic ability to get at multiple return values is provided by a special operator, **`MULTIPLE-VALUE-CALL`**, upon which the more commonly used **`MULTIPLE-VALUE-BIND`** macro is built.

The key thing to understand about multiple values is that returning multiple values is quite different from returning a list--if a form returns multiple values, unless you do something specific to capture the multiple values, all but the *primary value* will be silently discarded. To see the distinction, consider the function **`GETHASH`**, which returns two values: the value found in the hash table and a boolean that's **`NIL`** when no value was found. If it returned those two values in a list, every time you called **`GETHASH`** you'd have to take apart the list to get at the actual value, regardless of whether you cared about the second return value. Suppose you have a hash table, `*h*`, that contains numeric values. If **`GETHASH`** returned a list, you couldn't write something like this:

```
(+ (gethash 'a *h*) (gethash 'b *h*))
```

because **+** expects its arguments to be numbers, not lists. But because the multiple value mechanism silently discards the secondary return value when it's not wanted, this form works fine.

There are two aspects to using multiple values--returning multiple values and getting at the nonprimary values returned by forms that return multiple values. The starting points for

returning multiple values are the functions **VALUES** and **VALUES-LIST**. These are regular functions, not special operators, so their arguments are passed in the normal way. **VALUES** takes a variable number of arguments and returns them as multiple values; **VALUES-LIST** takes a single list and returns its elements as multiple values. In other words:

```
(values-list x) === (apply #'values x)
```

The mechanism by which multiple values are returned is implementation dependent just like the mechanism for passing arguments into functions is. Almost all language constructs that return the value of some subform will "pass through" multiple values, returning all the values returned by the subform. Thus, a function that returns the result of calling **VALUES** or **VALUES-LIST** will itself return multiple values--and so will another function whose result comes from calling the first function. And so on.[11]

But when a form is evaluated in a value position, only the primary value will be used, which is why the previous addition form works the way you'd expect. The special operator **MULTIPLE-VALUE-CALL** provides the mechanism for getting your hands on the multiple values returned by a form. **MULTIPLE-VALUE-CALL** is similar to **FUNCALL** except that while **FUNCALL** is a regular function and, therefore, can see and pass on only the primary values passed to it, **MULTIPLE-VALUE-CALL** passes, to the function returned by its first subform, *all* the values returned by the remaining subforms.

```
(funcall #'+ (values 1 2) (values 3 4))              ==> 4
(multiple-value-call #'+ (values 1 2) (values 3 4)) ==> 10
```

However, it's fairly rare that you'll simply want to pass all the values returned by a function onto another function. More likely, you'll want to stash the multiple values in different variables and then do something with them. The **MULTIPLE-VALUE-BIND** macro, which you saw in Chapter 11, is the most frequently used operator for accepting multiple return values. Its skeleton looks like this:

```
(multiple-value-bind (variable*) values-form
  body-form*)
```

The *values-form* is evaluated, and the multiple values it returns are bound to the *variables*. Then the *body-forms* are evaluated with those bindings in effect. Thus:

```
(multiple-value-bind (x y) (values 1 2)
  (+ x y)) ==> 3
```

Another macro, **MULTIPLE-VALUE-LIST**, is even simpler--it takes a single form, evaluates it, and collects the resulting multiple values into a list. In other words, it's the inverse of **VALUES-LIST**.

```
CL-USER> (multiple-value-list (values 1 2))
(1 2)
CL-USER> (values-list (multiple-value-list (values 1 2)))
1
2
```

However, if you find yourself using **MULTIPLE-VALUE-LIST** a lot, it may be a sign that some function should be returning a list to start with rather than multiple values.

Finally, if you want to assign multiple values returned by a form to existing variables, you can use **VALUES** as a **SETF**able place. For example:

```
CL-USER> (defparameter *x* nil)
*X*
CL-USER> (defparameter *y* nil)
*Y*
CL-USER> (setf (values *x* *y*) (floor (/ 57 34)))
1
23/34
CL-USER> *x*
1
CL-USER> *y*
23/34
```

EVAL-WHEN

A special operator you'll need to understand in order to write certain kinds of macros is **EVAL-WHEN**. For some reason, Lisp books often treat **EVAL-WHEN** as a wizards-only topic. But the only prerequisite to understanding **EVAL-WHEN** is an understanding of how the two functions **LOAD** and **COMPILE-FILE** interact. And understanding **EVAL-WHEN** will be important as you start writing certain kinds of more sophisticated macros, such as the ones you'll write in Chapters 24 and 31.

I've touched briefly on the relation between **LOAD** and **COMPILE-FILE** in previous chapters, but it's worth reviewing again here. The job of **LOAD** is to load a file and evaluate all the top-level forms it contains. The job of **COMPILE-FILE** is to compile a source file into a FASL file, which can then be loaded with **LOAD** such that `(load "foo.lisp")` and `(load "foo.fasl")` are essentially equivalent.

Because **LOAD** evaluates each form before reading the next, the side effects of evaluating forms earlier in the file can affect how forms later in the form are read and evaluated. For instance, evaluating an **IN-PACKAGE** form changes the value of ***PACKAGE***, which will affect the way subsequent forms are read.[12] Similarly, a **DEFMACRO** form early in a file can define a macro that can then be used by code later in the file.[13]

COMPILE-FILE, on the other hand, normally doesn't evaluate the forms it's compiling; it's when the FASL is loaded that the forms--or their compiled equivalents--will be evaluated. However, **COMPILE-FILE** must evaluate some forms, such as **IN-PACKAGE** and **DEFMACRO** forms, in order to keep the behavior of `(load "foo.lisp")` and `(load "foo.fasl")` consistent.

So how do macros such as **IN-PACKAGE** and **DEFMACRO** work when processed by **COMPILE-FILE**? In some pre-Common Lisp versions of Lisp, the file compiler simply knew it

should evaluate certain macros in addition to compiling them. Common Lisp avoided the need for such kludges by borrowing the **EVAL-WHEN** special operator from Maclisp. This operator, as its name suggests, allows you to control when specific bits of code are evaluated. The skeleton of an **EVAL-WHEN** form looks like this:

```
(eval-when (situation*)
  body-form*)
```

There are three possible *situations*--`:compile-toplevel`, `:load-toplevel`, and `:execute`--and which ones you specify controls when the *body-forms* will be evaluated. An **EVAL-WHEN** with multiple situations is equivalent to several **EVAL-WHEN** forms, one per situation, each with the same body code. To explain the meaning of the three situations, I'll need to explain a bit about how **COMPILE-FILE**, which is also referred to as the *file compiler*, goes about compiling a file.

To explain how **COMPILE-FILE** compiles **EVAL-WHEN** forms, I need to introduce a distinction between compiling *top-level* forms and compiling non-top-level forms. A top-level form is, roughly speaking, one that will be compiled into code that will be run when the FASL is loaded. Thus, all forms that appear directly at the top level of a source file are compiled as top-level forms. Similarly, any forms appearing directly in a top-level **PROGN** are compiled as top-level forms since the **PROGN** itself doesn't *do* anything--it just groups together its subforms, which will be run when the FASL is loaded.[14] Similarly, forms appearing directly in a **MACROLET** or **SYMBOL-MACROLET** are compiled as top-level forms because after the compiler has expanded the local macros or symbol macros, there will be no remnant of the **MACROLET** or **SYMBOL-MACROLET** in the compiled code. Finally, the expansion of a top-level macro form will be compiled as a top-level form.

Thus, a **DEFUN** appearing at the top level of a source file is a top-level form--the code that defines the function and associates it with its name will run when the FASL is loaded--but the forms within the body of the function, which won't run until the function is called, aren't top-level forms. Most forms are compiled the same when compiled as top-level and non-top-level forms, but the semantics of an **EVAL-WHEN** depend on whether it's being compiled as a top-level form, compiled as a non-top-level form, or simply evaluated, combined with what situations are listed in its situation list.

The situations `:compile-toplevel` and `:load-toplevel` control the meaning of an **EVAL-WHEN** compiled as a top-level form. When `:compile-toplevel` is present, the file compiler will evaluate the subforms at compile time. When `:load-toplevel` is present, it will compile the subforms as top-level forms. If neither of these situations is present in a top-level **EVAL-WHEN**, the compiler ignores it.

When an **EVAL-WHEN** is compiled as a non-top-level form, it's either compiled like a **PROGN**, if the `:execute` situation is specified, or ignored. Similarly, an evaluated **EVAL-WHEN**--which

includes top-level **EVAL-WHEN**s in a source file processed by **LOAD** and **EVAL-WHEN**s evaluated at compile time because they appear as subforms of a top-level **EVAL-WHEN** with the `:compile-toplevel` situation--is treated like a **PROGN** if `:execute` is present and ignored otherwise.

Thus, a macro such as **IN-PACKAGE** can have the necessary effect at both compile time and when loading from source by expanding into an **EVAL-WHEN** like the following:

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (setf *package* (find-package "PACKAGE-NAME")))
```

PACKAGE will be set at compile time because of the `:compile-toplevel` situation, set when the FASL is loaded because of `:load-toplevel`, and set when the source is loaded because of the `:execute`.

There are two ways you're most likely to use **EVAL-WHEN**. One is if you want to write macros that need to save some information at compile time to be used when generating the expansion of other macro forms in the same file. This typically arises with definitional macros where a definition early in a file can affect the code generated for a definition later in the same file. You'll write this kind of macro in Chapter 24.

The other time you might need **EVAL-WHEN** is if you want to put the definition of a macro and helper functions it uses in the same file as code that uses the macro. **DEFMACRO** already includes an **EVAL-WHEN** in its expansion so the macro definition is immediately available to be used later in the file. But **DEFUN** normally doesn't make function definitions available at compile time. But if you use a macro in the same file as it's defined in, you need the macro *and* any functions it uses to be defined. If you wrap the **DEFUN**s of any helper functions used by the macro in an **EVAL-WHEN** with `:compile-toplevel`, the definitions will be available when the macro's expansion function runs. You'll probably want to include `:load-toplevel` and `:execute` as well since the macros will also need the function definitions after the file is compiled and loaded or if you load the source instead of compiling.

Other Special Operators

The four remaining special operators, **LOCALLY**, **THE**, **LOAD-TIME-VALUE**, and **PROGV**, all allow you to get at parts of the underlying language that can't be accessed any other way. **LOCALLY** and **THE** are part of Common Lisp's declaration system, which is used to communicate things to the compiler that don't affect the meaning of your code but that may help the compiler generate better code--faster, clearer error messages, and so on.[15] I'll discuss declarations briefly in Chapter 32.

The other two, **LOAD-TIME-VALUE** and **PROGV**, are infrequently used, and explaining the reason why you might ever *want* to use them would take longer than explaining what they do. So

I'll just tell you what they do so you know they're there. Someday you'll hit on one of those rare times when they're just the thing, and then you'll be ready.

LOAD-TIME-VALUE is used, as its name suggests, to create a value that's determined at load time. When the file compiler compiles code that contains a **LOAD-TIME-VALUE** form, it arranges to evaluate the first subform once, when the FASL is loaded, and for the code containing the **LOAD-TIME-VALUE** form to refer to that value. In other words, instead of writing this:

```
(defvar *loaded-at* (get-universal-time))

(defun when-loaded () *loaded-at*)
```

you can write the following:

```
(defun when-loaded () (load-time-value (get-universal-time)))
```

In code not processed by **COMPILE-FILE**, **LOAD-TIME-VALUE** is evaluated once when the code is compiled, which may be when you explicitly compile a function with **COMPILE** or earlier because of implicit compilation performed by the implementation in the course of evaluating the code. In uncompiled code, **LOAD-TIME-VALUE** evaluates its form each time it's evaluated.

Finally, **PROGV** creates new dynamic bindings for variables whose names are determined at runtime. This is mostly useful for implementing embedded interpreters for languages with dynamically scoped variables. The basic skeleton is as follows:

```
(progv symbols-list values-list
  body-form*)
```

where *symbols-list* is a form that evaluates to a list of symbols and *values-list* is a form that evaluates to a list of values. Each symbol is dynamically bound to the corresponding value, and then the *body-forms* are evaluated. The difference between **PROGV** and **LET** is that because *symbols-list* is evaluated at runtime, the names of the variables to bind can be determined dynamically. As I say, this isn't something you need to do often.

And that's it for special operators. In the next chapter, I'll get back to hard-nosed practical topics and show you how to use Common Lisp's package system to take control of your namespaces so you can write libraries and applications that can coexist without stomping on each other's names.

---

[1]Of course, if **IF** wasn't a special operator but some other conditional form, such as **COND**, was, you could build **IF** as a macro. Indeed, in many Lisp dialects, starting with McCarthy's original Lisp, **COND** was the primitive conditional evaluation operator.

[2]Well, technically those constructs could also expand into a **LAMBDA** expression since, as I mentioned in Chapter 6, **LET** could be defined--and was in some earlier Lisps--as a macro that expands into an invocation of an anonymous function.

[3]Surprising as it may seem, it actually is possible to make anonymous functions recurse. However, you must use a rather esoteric mechanism known as the *Y combinator*. But the Y combinator is an interesting theoretical result, not a practical programming tool,

so is well outside the scope of this book.

[4]It's not required that **WITH-SLOTS** be implemented with **SYMBOL-MACROLET**--in some implementations, **WITH-SLOTS** may walk the code provided and generate an expansion with `x`, `y`, and `z` already replaced with the appropriate **SLOT-VALUE** forms. You can see how your implementation does it by evaluating this form:

```
(macroexpand-1 '(with-slots (x y z) obj (list x y z)))
```

However, walking the body is much easier for the Lisp implementation to do than for user code; to replace `x`, `y`, and `z` only when they appear in value positions requires a code walker that understands the syntax of all special operators and that recursively expands all macro forms in order to determine whether their expansions include the symbols in value positions. The Lisp implementation obviously has such a code walker at its disposal, but it's one of the few parts of Lisp that's not exposed to users of the language.

[5]One version of f2cl is available as part of the Common Lisp Open Code Collection (CLOCC): `http://clocc.sourceforge.net/`. By contrast, consider the tricks the authors of f2j, a FORTRAN-to-Java translator, have to play. Although the Java Virtual Machine (JVM) has a goto instruction, it's not directly exposed in Java. So to compile FORTRAN gotos, they first compile the FORTRAN code into legal Java source with calls to a dummy class to represent the labels and gotos. Then they compile the source with a regular Java compiler and postprocess the byte codes to translate the dummy calls into JVM-level byte codes. Clever, but what a pain.

[6]Since this algorithm depends on values returned by **RANDOM**, you may want to test it with a consistent random seed, which you can get by binding ***RANDOM-STATE*** to the value of `(make-random-state nil)` around each call to `algorithm-s`. For instance, you can do a basic sanity check of `algorithm-s` by evaluating this:

```
(let ((*random-state* (make-random-state nil))) (algorithm-s 10 200))
```

If your refactorings are all valid, this expression should evaluate to the same list each time.

[7]This is a pretty reasonable restriction--it's not entirely clear what it'd mean to return from a form that has already returned--unless, of course, you're a Scheme programmer. Scheme supports *continuations*, a language construct that makes it possible to return from the same function call more than once. But for a variety of reasons, few, if any, languages other than Scheme support this kind of continuation.

[8]If you're the kind of person who likes to know how things work all the way down to the bits, it may be instructive to think about how you might implement the condition system's macros using **BLOCK**, **TAGBODY**, closures, and dynamic variables.

[9]**UNWIND-PROTECT** is essentially equivalent to `try/finally` constructs in Java and Python.

[10]And indeed, CLSQL, the multi-Lisp, multidatabase SQL interface library, provides a similar macro called `with-database`. CLSQL's home page is at `http://clsql.b9.com`.

[11]A small handful of macros don't pass through extra return values of the forms they evaluate. In particular, the **PROG1** macro, which evaluates a number of forms like a **PROGN** before returning the value of the first form, returns that form's primary value only. Likewise, **PROG2**, which returns the value of the second of its subforms, returns only the primary value. The special operator **MULTIPLE-VALUE-PROG1** is a variant of **PROG1** that returns all the values returned by the first form. It's a minor wart that **PROG1** doesn't already behave like **MULTIPLE-VALUE-PROG1**, but neither is used often enough that it matters much. The **OR** and **COND** macros are also not always transparent to multiple values, returning only the primary value of certain subforms.

[12]The reason loading a file with an **IN-PACKAGE** form in it has no effect on the value of ***PACKAGE*** after **LOAD** returns is because **LOAD** binds ***PACKAGE*** to its current value before doing anything else. In other words, something equivalent to the following **LET** is wrapped around the rest of the code in **LOAD**:

```
(let ((*package* *package*)) ...)
```

Any assignment to ***PACKAGE*** will be to the new binding, and the old binding will be restored when **LOAD** returns. It also binds the variable ***READTABLE***, which I haven't discussed, in the same way.

[13]In some implementations, you may be able to get away with evaluating **DEFUN**s that use undefined macros in the function body as long as the macros are defined before the function is actually called. But that works, if at all, only when **LOAD**ing the definitions from source, not when compiling with **COMPILE-FILE**, so in general macro definitions must be evaluated before they're used.

[14]By contrast, the subforms in a top-level **LET** aren't compiled as top-level forms because they're not run directly when the FASL is loaded. They will run, but it's in the runtime context of the bindings established by the **LET**. Theoretically, a **LET** that binds no variables could be treated like a **PROGN**, but it's not--the forms appearing in a **LET** are never treated as top-level forms.

[15]The one declaration that has an effect on the semantics of a program is the **SPECIAL** declaration mentioned in Chapter 6.