12. Closures

A *closure* is a type of Lisp functional object useful for implementing certain advanced access and control structures. Closures give you more explicit control over the environment by allowing you to save the dynamic bindings of specified variables and then to refer to those bindings later, even after the construct (let, etc.) which made the bindings has been exited.

12.1 What a Closure Is

There is a view of dynamic variable binding that we use in this section because it makes it easier to explain what closures do. In this view, when a variable is bound dynamically, a new binding is created for it. The old binding is saved away somewhere and is inaccessible. Any references to the variable then get the contents of the new binding, and any setq's change the contents of the new value cell. Eventually the new binding goes away, and the old binding, along with its contents, becomes current again.

For example, consider the following sequence of Lisp forms:

```
(defvar a 3)              ; a becomes 3.

(let ((a 10))             ; a rebound to 10.
  (print (+ a 6)))        ; 16 is printed.

(print a)                 ; 3 is printed.
```

Initially there is a binding for a, and the setq form makes the contents of that binding be 3. Then the lambda-combination is evaluated. a is bound to 10: the old binding, which still contains 3, is saved away, and a new binding is created with 10 as its contents. The reference to a inside the lambda expression evaluates to the current binding of a, which is the contents of its current binding, namely 10. So 16 is printed. Then the newer binding is discarded and the old binding, which still contains a 3, is restored. The final print prints 3.

The form (closure *var-list function*), where *var-list* is a list of variables and *function* is any function, creates and returns a closure. When this closure is applied to some arguments, all of the bindings of the variables on *var-list* are saved away, and the bindings that those variables had *at the time* closure *was called* (that is, at the time the closure was created) are made to be the bindings of the symbols. Then *function* is applied to the arguments. (This paragraph is somewhat complex, but it completely describes the operation of closures; if you don't understand it, come back and read it again after reading the next two paragraphs.)

Here is another, lower-level explanation. The closure object stores several things inside it. First, it saves the *function*. Secondly, for each variable in *var-list*, it remembers what that variable's binding was when the closure was created. Then when the closure is called as a function, it first temporarily restores the bindings it has remembered inside the closure, and then applies *function* to the same arguments to which the closure itself was applied. When the function returns, the bindings are restored to be as they were before the closure was called.

Now, if we evaluate the form

```
(setq a
      (let ((x 3))
        (declare (special x))
        (closure '(x) 'frob)))
```

what happens is that a new binding is created for x, containing a fixnum 3. Then a closure is created, which remembers the function frob, the symbol x, and that binding. Finally the old binding of x is restored, and the closure is returned. Notice that the new binding is still around, because it is still known about by the closure. When the closure is applied, say by doing (funcall a 7), this binding is temporarily restored and the value of x is 3 again. If frob uses x as a free variable, it sees 3 as the value.

A closure can be made around any function, using any form that evaluates to a function. The form could evaluate to a compiled function, as would (function (lambda () x)). In the example above, the form is 'frob and it evaluates to the symbol frob. A symbol is also a good function. It is usually better to close around a symbol that is the name of the desired function, so that the closure points to the symbol. Then, if the symbol is redefined, the closure will use the new definition. If you actually prefer that the closure continue to use the old definition that was current when the closure was made, use function, as in:

```
(closure '(x) (function frob))
```

Explicit closures made with closure record only the dynamic bindings of the specified variables. Another closure mechanism is activated automatically to record lexical bindings whenever function is used around an explicit lambda expression, but closure itself has no interaction with lexical bindings.

It is the user's responsibility to make sure that the bindings that the closure is intended to record are dynamic bindings, either by means of special declarations (see page 51) as shown above or by making the variables globally special with defvar or equivalent. If the function closed over is an explicit lambda expression, it is occasionally necessary to use declarations within it to make sure that the variables are considered special there. But this is not needed if the variables are globally special or if a special declaration is lexically visible where closure is called.

Usually the compiler can tell when a special declaration is missing, but when making a closure the compiler detects this only after acting on the assumption that the variable is lexical, by which time it is too late to fix things. The compiler warns you if this happens.

In Zetalisp's implementation of closures, lambda-binding never really allocates any storage to create new bindings. Bindings receive separate storage only when the closure function itself finds they need it. Thus, there is no cost associated with closures when they are not in use.

Zetalisp closures differ from the closures of Lisp 1.5 (which were made with function) in that they save specific variables rather than the entire variable-binding environment. For their intended applications, this is an advantage. The explicit declaration of the variables in closure permits higher efficiency and more flexibility. In addition the program is clearer because the intended effect of the closure is made manifest by listing the variables to be affected. Lisp 1.5 closures are more similar to Zetalisp's automatic handling of lexical variables.

Closure implementation (which it not usually necessary for you to understand) involves two kinds of value cells. Every symbol has an *internal value cell*, part of the symbol itself, which is where its dynamic value is normally stored. When a variable is closed over, it gets an *external value cell* to hold its value. The external value cells behave according to the lambda-binding model used earlier in this section. The value in the external value cell is found through the usual access mechanisms (such as evaluating the symbol, calling **symeval**, etc.), because the internal value cell is made to contain a forwarding pointer to the external value cell that is current. Such a forwarding pointer is present in a symbol's value cell whenever its current binding is being remembered by some closure; at other times, there won't be an invisible pointer, and the value resides directly in the symbol's internal value cell.

12.2 Examples of the Use of Closures

One thing we can do with closures is to implement a *generator*, which is a kind of function which is called successively to obtain successive elements of a sequence. We implement a function **make-list-generator**, which takes a list and returns a generator that returns successive elements of the list. When it gets to the end it should return **nil**.

The problem is that in between calls to the generator, the generator must somehow remember where it is up to in the list. Since all of its bindings are undone when it is exited, it cannot save this information in a bound variable. It could save it in a global variable, but the problem is that if we want to have more than one list generator at a time, they will all try to use the same global variable and get in each other's way.

Here is how to solve this problem using closures:

```
(defun make-list-generator (l)
    (declare (special l))
    (closure '(l)
             #'(lambda ()
                 (prog1 (car l)
                        (setq l (cdr l))))))
```

(make-list-generator '(1 2 3)) returns a generator which, on successive calls, returns 1, 2, 3, and **nil**.

Now we can make as many list generators as we like; they won't get in each other's way because each has its own binding for l. Each of these bindings was created when the **make-list-generator** function was entered, and the bindings are remembered by the closures.

The following example uses closures which share bindings:

```
(defvar a)
(defvar b)

(defun foo () (setq a 5))

(defun bar () (cons a b))

(let ((a 1) (b 1))
   (setq x (closure '(a b) 'foo))
   (setq y (closure '(a b) 'bar)))
```

When the let is entered, new bindings are created for the symbols a and b, and two closures are created that both point to those bindings. If we do (funcall x), the function foo is be run, and it changes the contents of the remembered binding of a to 5. If we then do (funcall y), the function bar returns (5 . 1). This shows that the binding of a seen by the closure y is the same binding seen by the closure x. The top-level binding of a is unaffected.

Here is how we can create a function that prints always using base 16:

```
(deff print-in-base-16
      (let ((*print-base* 16.))
        (closure '(*print-base*) 'print)))
```

12.3 Closure-Manipulating Functions

closure *var-list function*

Creates and returns a closure of *function* over the variables in *var-list*. Note that all variables on *var-list* must be declared special if the function is to compile correctly.

To test whether an object is a closure, use the closurep predicate (see page 13) or (typep *object* 'closure).

symeval-in-closure *closure symbol*

Returns the binding of *symbol* in the environment of *closure*; that is, it returns what you would get if you restored the bindings known about by *closure* and then evaluated *symbol*. This allows you to "look around inside" a closure. If *symbol* is not closed over by *closure*, this is just like symeval.

symbol may be a locative pointing to a value cell instead of a symbol (this goes for all the whatever-in-closure functions).

set-in-closure *closure symbol x*

Sets the binding of *symbol* in the environment of *closure* to *x*; that is, it does what would happen if you restored the bindings known about by *closure* and then set *symbol* to *x*. This allows you to change the contents of the bindings known about by a closure. If *symbol* is not closed over by *closure*, this is just like set.

locate-in-closure *closure symbol*
Returns the location of the place in *closure* where the saved value of *symbol* is stored. An equivalent form is (locf (symeval-in-closure *closure symbol*)).

boundp-in-closure *closure symbol*
Returns t if *symbol*'s binding in *closure* is not void. This is what (boundp *symbol*) would return if executed in *closure*'s saved environment.

makunbound-in-closure *closure symbol*
Makes *symbol*'s binding in *closure* be void. This is what (makunbound *symbol*) would do if executed in *closure*'s saved environment.

closure-alist *closure*
Returns an alist of (*symbol* . *value*) pairs describing the bindings that the closure performs when it is called. This list is not the same one that is actually stored in the closure; that one contains pointers to value cells rather than symbols, and closure-alist translates them back to symbols so you can understand them. As a result, clobbering part of this list does not change the closure.

The list that is returned may contain void cells if some of the closed-over variables were void in the closure's environment. In this case, printing the value will get an error (accessing a cell that contains a void marker is always an error unless done in a special, careful way) but the value can still be passed around.

closure-variables *closure*
Returns a list of variables closed over in *closure*. This is equal to the first argument specified to the function closure when this closure was created.

closure-function *closure*
Returns the closed function from *closure*. This is the function that was the second argument to closure when the closure was created.

closure-bindings *closure*
Returns the actual list of bindings to be performed when *closure* is entered. This list can be passed to sys:%using-binding-instances to enter the closure's environment without calling the closure. See page 287.

copy-closure *closure*
Returns a new closure that has the same function and variable values as *closure*. The bindings are not shared between the old closure and the new one, so that if the old closure changes some closed variable's values, the values in the new closure do not change.

let-closed ((*variable value*)...) *function* *Macro*
When using closures, it is very common to bind a set of variables with initial values only in order to make a closure over those variables. Furthermore, the variables must be declared special. let-closed is a special form which does all of this. It is best described by example:

```
(let-closed ((a 5) b (c 'x))
   (function (lambda () ...)))
```
macro-expands into
```
(let ((a 5) b (c 'x))
  (declare (special a b c))
  (closure '(a b c)
    (function (lambda () ...))))
```

Note that the following code, which would often be useful, does not work as intended if x is not special outside the let-closed:
```
(let-closed ((x x))
  (function ...))
```
This is because the reference to x as an initialization for the new binding of x is affected by the special declaration that the let-closed produces. It therefore does not see any lexical binding of x. This behavior is unfortunate, but it is required by the Common Lisp specifications. To avoid the problem, write
```
(let ((y x))
  (let-closed ((x y))
    (function ...)))
```
or simply change the name of the variable outside the let-closed to something other than x.

12.4 Entities

An entity is almost the same thing as a closure; an entity behaves just like a closure when applied, but it has a recognizably different data type which allows certain parts of the system such as the printer and describe to treat it differently. A closure is simply a kind of function, but an entity is assumed to be a message-receiving object. Thus, when the Lisp printer (see section 23.1, page 506) is given a closure, it prints a simple textual representation, but when it is handed an entity, it sends the entity a :print-self message, which the entity is expected to handle. The describe function (see page 791) also sends entities messages when it is handed them. So when you want to make a message-receiving object out of a closure, as described on page 407, you should use an entity instead.

To a large degree, entities are made obsolete by flavors (see chapter 21, page 401). Flavors have had considerably more attention paid to their efficiency and to good tools for using them. If what you are doing is flavor-like, it is better to use flavors.

entity *variable-list function*

Returns a newly constructed entity. This function is just like the function closure except that it returns an entity instead of a closure.

The *function* argument should be a symbol which has a function definition and a value. When typep is applied to this entity, it returns the value of that symbol.

To test whether an object is an entity, use the entityp predicate (see page 13). The functions symeval-in-closure, closure-alist, closure-function, etc. also operate on entities.