11. 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 environment created by the entering of a dynamic contour (i.e. a **lambda**, **do**, **prog**, **progv**, **let**, or any of several other special forms) and then to use that environment elsewhere, even after the contour has been exited.

11.1 What a Closure Is

There is a view of lambda-binding that we will use in this section because it makes it easier to explain what closures do. In this view, when a variable is bound, a new value cell is created for it. The old value cell is saved away somewhere and is inaccessible. Any references to the variable will get the contents of the new value cell, and any **setq**'s will change the contents of the new value cell. When the binding is undone, the new value cell goes away, and the old value cell, along with its contents, is restored.

For example, consider the following sequence of Lisp forms:

```
(setq a 3)

(let ((a 10))
  (print (+ a 6)))

(print a)
```

Initially there is a value cell for **a**, and the **setq** form makes the contents of that value cell be **3**. Then the **lambda**-combination is evaluated. **a** is bound to **10**: the old value cell, which still contains a **3**, is saved away, and a new value cell 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 value cell, namely **10**. So **16** is printed. Then the binding is undone, discarding the new value cell, and restoring the old value cell, which still contains a **3**. The final **print** prints out a **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 value cells of the variables on *var-list* are saved away, and the value cells that those variables had *at the time* **closure** *was called* (that is, at the time the closure was created) are made to be the value cells 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 value cell was when the closure was created. Then when the closure is called as a function, it first temporarily restores the value cells 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 value cells are restored to be as they were before the closure was called.

Now, if we evaluate the form

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

what happens is that a new value cell is created for x, containing a fixnum 3. Then a closure is created, which remembers the function frob, the symbol x, and that value cell. Finally the old value cell of x is restored, and the closure is returned. Notice that the new value cell is still around, because it is still known about by the closure. When the closure is applied, say by doing (funcall a 7), this value cell will be restored and the value of x will be 3 again. If frob uses x as a free variable, it will see 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 lambda expression, as in '(lambda () x), or 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, then close around the definition of the symbol rather than the symbol itself. In the above example, that would be done by

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

Because of the way closures are implemented, the variables to be closed over must not get turned into "local variables" by the compiler. Therefore, all such variables must be declared special. This can be done with an explicit declare (see page 234), with a special form such as defvar (page 19), or with let-closed (page 184). In simple cases, a local-declare around the binding will do the job. 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 local, by which time it is too late to fix things. The compiler will warn you if this happens.

In Zetalisp's implementation of closures, lambda-binding never really allocates any storage to create new value cells. Value cells are created only by the closure function itself, when they are needed. Thus, there is no cost associated with closures when they are not in use.

Zetalisp closures are not closures in the true sense, as they do not save the whole variable-binding environment; however, most of that environment is irrelevant in any given application. The explicit declaration of the variables that are to be closed allows the implementation to have high efficiency. It also allows the programmer to choose explicitly for each variable whether it is to be bound at the point of call or bound at the point of definition (e.g. of creation of the closure), a choice that is not conveniently available in other languages. In addition the program is clearer because the intended effect of the closure is made manifest by listing the variables to be affected.

Closure implementation (which it not usually necessary for you to understand) involves two kinds of value cells. Every symbol has an *internal value cell*, which is where its value is normally stored. When a variable is closed over by a closure, the variable 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 an invisible pointer to the external value cell currently in effect. A symbol will use such an invisible pointer whenever its current value cell is a value cell that some closure is remembering; at other times, there won't be an invisible pointer, and the value will just reside in the internal value cell.

11.2 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 will implement a function **make-list-generator**, which takes a list and returns a generator that will return 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 we can use closures to solve the problem:

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

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 (external) value cell for **l**. Each of these value cells was created when the **make-list-generator** function was entered, and the value cells are remembered by the closures.

The following form uses closures to create an advanced accessing environment:

```
(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 value cells are created for the symbols **a** and **b**, and two closures are created that both point to those value cells. If we do **(funcall x)**, the function **foo** will be run, and it will change the contents of the remembered value cell of **a** to **5**. If we then do **(funcall y)**, the function **bar** will return **(5 . 1)**. This shows that the value cell of **a** seen by the closure **y** is the same value cell seen by the closure **x**. The top-level value cell of **a** is unaffected.

Here is how we can create a function that prints always using base 10:

```
(deff print-in-base-10
      (let ((base 10.))
        (closure '(base) 'print)))
```

11.3 Closure-Manipulating Functions

closure *var-list function*

This 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 10). The typep function will return the symbol closure if given a closure. (typep *x* 'closure) is equivalent to (closurep *x*).

symeval-in-closure *closure symbol*

This returns the binding of *symbol* in the environment of *closure*; that is, it returns what you would get if you restored the value cells 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*

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

locate-in-closure *closure symbol*

This 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* is not "unbound" in *closure*. This is what (boundp *symbol*) would return if executed in *closure*'s saved environment.

makunbound-in-closure *closure symbol*

Makes *symbol* be unbound, inside *closure*. 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 will not change the closure.

The list that is returned may contain "unbound" markers if some of the closed-over variables were unbound in the closure's environment. In this case, printing the value will get an error (accessing a cell that contains an unbound 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 215.

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* *Special Form*
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 as "special" for the compiler. 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

```
(local-declare ((special a b c))
   (let ((a 5) b (c 'x))
      (closure '(a b c)
         (function (lambda () ...)))))
```

11.4 Entities

An entity is almost the same thing as a closure; the data type is nominally different but an entity behaves just like a closure when applied. The difference is that some system functions, such as **print**, operate on them differently. When **print** sees a closure, it prints the closure in a standard way. When **print** sees an entity, it calls the entity to ask the entity to print itself.

To some degree, entities are made obsolete by flavors (see chapter 20, page 321). The use of entities as message-receiving objects is explained in section 20.13, page 357.

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 will return the value of that symbol.

To test whether an object is an entity, use the **entityp** predicate (see page 10). The functions **symeval-in-closure**, **closure-alist**, **closure-function**, etc. also operate on entities.