Submitted by Frank Yellin, Lucid, Inc.

Closures

This article introduces closures, a powerful feature of Common Lisp that allows you to write succinct code.

A closure is a function together with the environment in which it was created. The principal idea behind closures is that in Common Lisp, functions are "first-class objects," that is, you can write code that takes a function as an argument, and you can write code that returns a function as one of its values.

Closures as Arguments to Functions

Let's look at some simple examples. The first example defines a function that takes a single argument, which must be a function or a symbol with a function definition, and calls that function with the single argument 2:

```
> (defun apply-to-two (func)
    (funcall func 2))
APPLY-TO-TWO
> (apply-to-two '1+)
3
> (apply-to-two #'plusp)
T
```

The function argument does not need to be a named function. The following examples use a **lambda** expression for the function argument:

```
> (apply-to-two
    #'(lambda (x) (+ x 3)))
5
> (apply-to-two
    #'(lambda (x) (+ 1 (* x x x))))
9
```

The expression `#'(lambda (x) (+ x 3))` is a function that takes a single argument and adds 3 to it. The expression `#'(lambda (x) (+ 1 (* x x x)))` is a function that takes a single argument, cubes it, then adds 1.

A number of predefined Common Lisp functions can take a function as one of their arguments. Among the more useful ones are **mapcar**, which applies a function to each element, **remove-if** and **remove-if-not**, which remove elements from a list

or array, and **find-if** and **find-if-not**, which find the first item in an array or list that satisfies a condition.

Functions that Return Closures

A more powerful use of closures is that functions can return other functions as their value. In the following example, the function **make-adder** takes as its argument a single value **x** and returns as its value a function. The returned function takes a single argument, which it calls **y**, and returns the value **(+ x y)**.

```
> (defun make-adder (x)
    (check-type x number)
    #'(lambda (y) (+ x y)))
MAKE-ADDER
> (setq add2 (make-adder 2))
#<Interpreted-Function (LAMBDA (Y) (+ X Y)) 949476>
> (funcall add2 5)
7
> (funcall (make-adder pi) 1)
4.141592653589793
> (funcall add2 1)
3
```

Notice that something very subtle is going on. When we call **(make-adder 2)**, it returns a function. That function references the local variable **x**, which was bound to the integer 2 when the function was created. The function that is returned "remembers" that **x** had the value 2 when the function was created. In Common Lisp terms, the local variable **x** has been ***closed over***; that is, its value is not saved in a global variable somewhere but is intrinsically part of the function that is returned. The resulting function is called a ***closure***.

An important feature of Common Lisp is that every function has access to the environment in which it was created. It can both access and modify the variables in that environment. For example, here is one possible, albeit inefficient, alternative implementation of **cons**, **car**, and **cdr**:

```
> (defun my-cons (x y)
    #'(lambda (option &optional value)
        (case option
          (:car x)
          (:cdr y)
          (:set-car (setf x value))
          (:set-cdr (setf y value)))))
MY-CONS
> (defun my-car (cons) (funcall cons :car))
MY-CAR
> (defun my-cdr (cons) (funcall cons :cdr))
```

```
MY-CDR
> (defun my-set-car (cons value)
    (funcall cons :set-car value))
MY-SET-CAR
> (defun my-set-cdr (cons value)
    (funcall cons :set-cdr value))
> (defsetf my-car my-set-car)
MY-CAR
> (defsetf my-cdr my-set-cdr)
MY-CDR
> (setq cell (my-cons 'a 'b))
#<Interpreted-Function ...  953386>
> (my-car cell)
A
> (my-cdr cell)
B
> (setf (my-car cell) 44)
44
> (my-car cell)
44
```

Every function remembers the lexical environment in which it was created. In addition to modifying and accessing local variables, a function can also, under certain conditions, remember the names of any **block** form in which it was created, and the names of any **tagbody** tags that it can go to:

```
> (defun outer-function ()
    (block outer-block
      (inner-function
        #'(lambda (x) (return-from outer-block x)))
      (error "This piece of code
              will never be executed")))
OUTER-FUNCTION
> (defun inner-function (arg)
    (funcall arg 23))
INNER-FUNCTION
> (outer-function)
23
```

However, the following code would cause an error:

```
> (defun bad-closure ()
      (block outer-block
        #'(lambda (x) (return-from outer-block x))))
BAD-CLOSURE
> (bad-closure)
#<Interpreted-Function
(LAMBDA (X) (RETURN-FROM OUTER-BLOCK X)) C8C40E>
```

```
> (funcall * 23)
>>Error: A RETURN-FROM block OUTER-BLOCK occurred
  from within an out of scope closure
```

Once you've exited from a block or a tagbody, any closures created inside the block or tagbody that explicitly use the block name or tagbody tags can no longer be used.

Other Uses of Closures

Closures can be used for many different purposes in Common Lisp. For example, you can use closures to get information from a child process:

```
(let ((ready nil))
  (make-process :function 'child-function
                :name "child"
                :args (list
                        #'(lambda (x) (setq ready x))))
  (process-wait "Wait for child"
                #'(lambda () (eq x :ready)))
  ... other code ...)

;;; The child process can be defined as follows:

(defun child-process (closure)
  ... do some work ...
  (funcall closure :ready)   ; tell my parent I'm ready
  ... do some work
  (funcall closure :even-readier)
)
```

The child process can use the closure to communicate directly back to the parent. This method has several advantages over having the child set a global variable to indicate its state.

1. The child doesn't need to know how it is sending information to its parent.
2. Several different parent processes can communicate with several different child processes. Since each parent communicates by using its own copy of a local variable, there is never any interference between processes.

Another interesting use for closures is delayed evaluation. Delayed evaluation can be used to give the illusion of infinitely large objects, even though Lisp only creates as much of the object as the user wants to look at.

Imagine an implementation of streams (an infinite list of items). The positive integers are the stream `1, 2, 3, 4, 5, ...` and the prime numbers are the stream `2, 3, 5, 7, 11, 13, ....`

Obviously, we can't compute all the elements of a stream ahead of time. So, we'll define a stream to be a list of two elements. The first element is the first item in the stream. The second element is an expression whose value we've delayed evaluating. We evaluate that expression to get the stream consisting of all the elements except the first.

Closures make it easy to implement delayed evaluation. You can delay the evaluation of the expression by typing `#'(lambda () `*expression*`)`. The returned value will be a closure. If you call that closure with **funcall**, you get the value of the original expression.

```
;;; First, let's create some stream functions:
> (defun first-stream (stream)
    (first stream))
FIRST-STREAM
> (defun rest-stream (stream)
    (funcall (second stream)))

;;; And then, let's make some functions
;;; to look at the streams we've created:
> (defun nth-stream (n stream)
    "return the nth element of a stream"
    (if (zerop n)
        (first-stream stream)
        (nth-stream (1- n) (rest-stream stream))))
NTH-STREAM
> (defun firstn-stream (n stream)
    "return a list of the first n elements in a stream"
    (if (zerop n)
        ()
        (cons (first-stream stream)
              (firstn-stream (1- n)
                             (rest-stream stream)))))
NTH-STREAM

;;; Finally, let's create some streams:
> (setq zero-stream (list 0 #'(lambda () zero-stream)))
(0 #<Interpreted-Function
(LAMBDA NIL ZERO-STREAM) CE588E>)
> (firstn-stream 10 zero-stream)
(0 0 0 0 0 0 0 0 0 0)
```

Note the use of delayed evaluation. We define the stream `zero-stream` in terms of itself! Similarly, we can define the integers as follows:

```
> (defun integers-from (n)
    (list n #'(lambda () (integers-from (1+ n)))))
INTEGERS-FROM
> (firstn-stream 20 (integers-from 0))
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19)
```

Even though the function `#'integers-from` looks like it should recurse forever, it doesn't. That's because the evaluation of the recursive part is delayed until something else, in this case, the evaluation of `firstn-stream`, explicitly asks us to recurse.

Here are some other functions we can write:

```
> (defun map-stream (function stream)
    (list (funcall function (first-stream stream))
          #'(lambda ()
              (map-stream function (rest-stream stream)))))
MAP-STREAM
> (defun remove-if-stream (function stream)
    (let ((first-element (first-stream stream)))
      (if (not (funcall function first-element))
          (list first-element
                #'(lambda ()
                    (remove-if-stream
                      function (rest-stream stream))))
          (remove-if-stream function
                            (rest-stream stream)))))
REMOVE-IF-STREAM

;; Get the first 20 elements of the squares
;; of the numbers that are not even.
> (firstn-stream 20
    (map-stream #'(lambda (x) (* x x))
      (remove-if-stream #'evenp
                        (integers-from 0))))
(1 9 25 49 81 121 169 225 289 361 441 529
  625 729 841 961 1089 1225 1369 1521)
> (defun prime-sieve (stream)
  (let ((first-element (first-stream stream)))
    (cons-stream
       first-element
       (prime-sieve
         (remove-if-stream
           #'(lambda (y) (zerop (mod y first-element)))
           (rest-stream stream))))))
PRIME-SIEVE
> (setq primes (prime-sieve (integers-from 2)))
(2 #<Interpreted-Function ...  CF7B46>)
> (firstn-stream 25 primes)
```

```
(2 3 5 7 11 13 17 19 23 29 31 37 41 43
  47 53 59 61 67 71 73 79 83 89 97)
```

Used properly, closures are an important programming paradigm that can be applied to a variety of problems.

Second Puzzle

The second puzzle looks like very simple since we know the value of each term of the application. But now the current continuations are these of the evaluator while evaluating the functional and parametric part of the application. Let us suppose that, as in Lisp, terms are evaluated from left to right. The original form is $_{k_0}(_{k_1}(\texttt{call/cc}_1\ \texttt{call/cc}_2)\ (\texttt{call/cc}_3\ \texttt{call/cc}_4))$ where k_1 is $\lambda\phi\cdot_{k_0}(\phi_{k_2}(\texttt{call/cc}_3\ \texttt{call/cc}_4))$ and k_2 is $\lambda\epsilon\cdot_{k_0}(k_1\ \epsilon)$. The original form becomes $_{k_0}(k_1\ k_2)$ that is to say $_{k_0}(k_2\ k_2'(\texttt{call/cc}_3\ \texttt{call/cc}_4))$ where k_2' is $\lambda\epsilon\cdot_{k_0}(k_2\ \epsilon)$. The new evaluation of the argument leads to $_{k_0}(k_2\ k_2')$ which in turn will lead to $_{k_0}(k_2'\ k_2'')$ ... i.e. a tail recursive endless loop.

One may ponder wether the evaluation order has any influence on this computation. If terms are evaluated now from right to left, the computation would now be $_{k_0}((\texttt{call/cc}_1\ \texttt{call/cc}_2)\ _{k_2}(\texttt{call/cc}_3\ \texttt{call/cc}_4))$ where k_2 is $\lambda\epsilon\cdot_{k_0}(_{k_1}(\texttt{call/cc}_1\ \texttt{call/cc}_2)\ \epsilon)$ and where k_1 is $\lambda\phi\cdot_{k_0}(\phi\ k_2)$. The computation is therefore $_{k_0}(k_1\ k_2)$ i.e. $_{k_0}(k_2\ k_2)$ i.e. $_{k_0}(k_1'(\texttt{call/cc}_1\ \texttt{call/cc}_2)\ k_2)$ i.e. $_{k_0}(k_1'\ k_2)$ which also loops endless. The only difference is that here only one continuation is built by cycle instead of two if the previous evaluation order was followed. This order thus lessens garbage collection.

This result is due to the fact that $_k$(call/cc call/cc) gives the current continuation k to this same continuation k and therefore creates a cycle $(k\ k)$. This fact can be exploited for recursion without `letrec`. Consider for instance

```
(let ((fact nil)
      (r 1) )
   (let ((n (call/cc call/cc)))
      (if fact
          (if (= n 0) r
              (begin (set! r (* n r))
                     (fact (1- n)) ) )
          (begin (set! fact n)
                 (fact 10) ) ) ) )
;;; Returns 10!
```

References

[Danvy & Malmkjær] Olivier Danvy, Karoline Malmkjæer, *Intensions and Extensions in a Reflective Tower*, 1988 ACM Conference on Lisp and Functional Programming, pp 327–341, Snowbird, Utah.

CHRISTIAN QUEINNEC NITSAN SÉNIAK