Local Procedure Bindings and Recursion

Summary: We consider `letrec`, a variant of `let` that permits us to write recursive local procedures.

Contents:
- Introduction
- Local Procedure Bindings
- A Problem: Recursive Procedure Bindings
- A Solution: `letrec`
- Husk-and-Kernel with Local Kernels
- An Alternative: The Named `let`

Introduction

As you have probably noted, we often find it useful to write helper procedures that accompany our main procedures. For example, we might use a helper to recurse on part of a larger structure or to act as the kernel of a husk-and-kernel procedure.

One issue with this technique is that it often makes little sense for other procedures to use the helper, so we should restrict access to the helper procedure. In particular, only the procedure that uses the helper (unless it’s a very generic helper) should be able to access the helper. We know how to restrict access to variables (using `let` and `let*`). Can we do the same for procedures?

Local Procedure Bindings

Yes. As the reading on local bindings suggested, it is possible for a `let`-expression to bind an identifier to a non-recursive procedure:

```scheme
> (let ((square (lambda (n) (* n n))))
  (square 12))
144
```

Like any other binding that is introduced in a `let`-expression, this binding is local. Within the body of the `let`-expression, it supersedes any previous binding of the same identifier, but as soon as the value of the `let`-expression has been computed, the local binding evaporates.

A Problem: Recursive Procedure Bindings

However, it is not possible to bind an identifier to a recursively defined procedure in this way. For example, consider the following expression which is intended to recursively create the list of values from 10 to 1.
> (let ((count-down (lambda (n)
    (if (zero? n)
        null
        (cons n (count-down (- n 1))))))
  (count-down 10))
reference to undefined identifier: count-down

The difficulty is that when the lambda-expression is evaluated, the identifier count-down has not yet been bound, so the value of the lambda-expression is a procedure that includes an unbound identifier. Binding this procedure value to the identifier count-down creates a new environment, but does not affect the behavior of procedures that were constructed in the old environment. So, when the body of the let-expression invokes this procedure, we get the unbound-identifier error.

Changing let to let* wouldn’t help in this case, since even under let* the lambda-expression would be completely evaluated before the binding is established.

A Solution: letrec

What we need is some variant of let that binds the identifier to some kind of a place-holder and adds the binding to the environment first, then computes the value of the lambda-expression in the new environment, and then finally substitutes that value for the place-holder. This will work in Scheme, so long as the procedure is not actually invoked until we get into the body of the expression.

Fortunately, the designers of Scheme decided to let us write local procedures using that technique. The keyword associated with this “recursive binding” variant of let is named letrec:

> (letrec ((count-down (lambda (n)
    (if (zero? n)
        null
        (cons n (count-down (- n 1))))))
  (count-down 10))
(10 9 8 7 6 5 4 3 2 1)

A letrec-expression constructs all of its place-holder bindings simultaneously (in effect), then evaluates all of the lambda-expressions simultaneously, and finally replaces all of the place-holders simultaneously. This makes it possible to include binding specifications for mutually recursive procedures (which invoke each other) in the same binding list. Here’s a particularly silly example, which takes a list of numbers and alternately adds and subtracts them.

> (letrec ((up-sum (lambda (ls)
    (if (null? ls)
        0
        (+ (down-sum (cdr ls)) (car ls)))))
  (down-sum (lambda (ls)
    (if (null? ls)
        0
        (- (up-sum (cdr ls)) (car ls))))))
(up-sum (list 1 2 3 6 12 7)))
-21
; which is 1 - 23 + 6 - 12 + 7.
Husk-and-Kernel with Local Kernels

We can use letrec expressions to separate the husk and the kernel of a recursive procedure without having to define two procedures.

;;; Procedure:
;;;   list.index
;;; Parameters:
;;;   stuff, a list.
;;;   sought, a value.
;;; Purpose:
;;;   Find the position of a given value in a given list.
;;; Produces:
;;;   pos
;;; Preconditions:
;;;   None.
;;; Postconditions:
;;;   pos is either (1) #f, if sought is not in stuff or (2) a nonnegative integer, if sought is in stuff.
;;;   Affects neither sought nor stuff.
;;;   If pos is #f, then sought is not in stuff.
;;;   If pos is a nonnegative integer, then (list-ref stuff pos) is equal to sought.
(define list.index
  (lambda (stuff sought)
    (list.index-kernel stuff sought 0)))

;;; Kernel:
;;;   list.index-kernel
;;; Parameters:
;;;   rest, a list (a sublist of stuff, described above)
;;;   sought, a value
;;;   bypassed, an integer that counts how many values have been bypassed in stuff
;;; Purpose:
;;;   To keep looking for sought in part of the list.
(define list.index-kernel
  (lambda (rest sought bypassed)
    (cond ((null? rest) #f)
          ((equal? (car rest) sought) bypassed)
          (else (list.index-kernel (cdr rest) sought (+ bypassed 1))))))

This works, but it’s more stylish to construct the kernel procedure inside a letrec expression, so that the extra identifier can be bound to it locally:

(define list.index
  (lambda (stuff sought)
    (letrec ((kernel (lambda (rest sought bypassed)
                           (cond ((null? rest) #f)
                                 ((equal? (car rest) sought) bypassed)
                                 (else (list.index-kernel (cdr rest) sought (+ bypassed 1)))))
                  (kernel stuff 0))))
Of course, now that the recursive kernel procedure is now inside the body of the index procedure, it is not necessary to pass the value of sought to the kernel as a parameter. Instead, the kernel can treat sought as if it were a constant, since its value doesn’t change during any of the recursive calls.

```
(define list.index
  (lambda (stuff sought)
    (letrec ((kernel (lambda (rest bypassed)
        (cond ((null? rest) #f)
          ((equal? (car rest) sought) bypassed)
          (else (kernel (cdr rest) (+ bypassed 1)))))))
      (kernel stuff 0))))
```

The same approach can be used to perform precondition tests efficiently, by placing them with the husk in the body of a letrec-expression and omitting them from the kernel. For instance, here’s how to introduce precondition tests into the spots.leftmost procedure from the reading on preconditions and postconditions:

```
(define spots.leftmost
  (let ((spot.leftmost
        (lambda (spot1 spot2)
          (if (< (spot.col spot1) (spot.col spot2)) spot1 spot2))))
    (lambda (spots)
      (letrec (
        ; all-spots? checks to see if all the values in a list are spots
        (all-spots?
          (lambda (ls)
            (or (null? ls)
                (and (spot? (car ls))
                  (all-spots? (cdr ls))))))
        ; kernel finds the leftmost spot w/o verifying preconditions
        (kernel
          (lambda (spots)
            (if (null? (cdr spots))
                (car spots)
                (spot.leftmost (car spots) (spots.leftmost (cdr spots)))))))
      (cond
        ((null? spots) (throw "spots.leftmost: Argument must be a *non-empty* list of spots")
        ((not (list? spots)) (throw "spots.leftmost: Argument must be a *list* of spots")
        ((not (all-spots? spots)) (throw "spots.leftmost: Argument list must contain only spots")
        (else (kernel spots))))))))
```

Embedding the kernel inside the definition of spots.leftmost rather than writing a separate spots.leftmost-kernel procedure has another advantage: It is impossible for an incautious user to invoke the kernel procedure directly, bypassing the precondition tests. The only way to get at the recursive procedure to which kernel is bound is to invoke the procedure within which the binding is established.
We’ve recycled the name kernel in this example to drive home the point that local bindings in separate procedures don’t interfere with one another. Even if both procedures were active at the same time, the correct kernel procedure would be used in each case because the correct local binding would supersede all others.

Hence, even though both list.index and spots.leftmost have a helper named kernel, we can safely write

(list.index figure (spots.leftmost figure))

to find the index of the leftmost spot in figure.

An Alternative: The Named let

Many programmers use letrec-expressions in writing most of these husk-and-kernel procedures. When there is only one recursive procedure to bind, however, a contemporary Scheme programmer might well use yet another variation of the let-expression -- the “named let”.

The named let has the same syntax as a regular let-expression, except that there is an identifier between the keyword let and the binding list. The named let binds this extra identifier to a kernel procedure whose parameters are the same as the variables in the binding list and whose body is the same as the body of the let-expression. Here’s the basic form

(let name ((param_1 val_1) (param_2 val_2) ... (param_n val_n))
 body)

You can think of this as a more elegant (and, eventually, more readable) shorthand for

(letrec ((name (lambda (param_1 ... param_n) 
 body))
 (name val_1 ... val_n))

So, for example, one might write the list.index procedure as follows:

(define list.index
 (lambda (sought ls)
  (let kernel ((rest ls) (bypassed 0))
   (cond ((null? rest) #f)
    ((equal? (car rest) sought) bypassed)
    (else (kernel (cdr rest) (+ bypassed 1)))))))

When we enter the named let, the identifier rest is bound to the value of ls and the identifier bypassed is bound to 0, just as if we were entering an ordinary let-expression. In addition, however, the identifier kernel is bound to a procedure that has rest and bypassed as parameters and the body of the named let as its body. As we evaluate the cond-expression, we may encounter a recursive call to the kernel procedure -- in effect, we re-enter the body of the named let, with rest now re-bound to
the former value of (cdr rest) and bypassed to the former value of (+ bypassed 1).

As another example, here’s a version of `sum` that uses a named `let`:

```
(define sum
  (lambda (ls)
    (let kernel ((rest ls) (running-total 0))
      (if (null? rest) running-total
       (kernel (cdr rest) (+ (car rest) running-total)))))
```

Scheme programmers seem to be mixed in their reaction to the named `let`. Some find it clear and elegant, others find it murky and too special-purpose. My colleagues like to use it. I’ll admit that I first found it murky, but eventually came to like it. I hope that you will, too.