Class 19: Local Procedure Bindings

Held: Wednesday, February 21, 2007

Summary: Today we consider techniques for defining local procedures, procedures that are only available to select other procedures.

Related Pages:
- EBoard.
- Lab: Local Procedure Bindings and Recursion.
- Reading: Local Procedure Bindings and Recursion.

Notes:
- Sorry, I won’t be there today. Dr. Davis should replace me.
- Don’t forget tomorrow’s convo.
- There’s also a CS Thursday extra tomorrow.
- Saturday night at 7:00 p.m. in JRC 101, Kumail Nanjiani ’01 and a group of other Chicago comedians will be performing. Extra credit for attending (if you get this form of EC, you can earn 3.5 points of EC, rather than the basic 3).
- Tomorrow’s reading on analyzing algorithms should be ready by Thursday morning.

Overview:

- Why Have Local Procedures.
- Creating Local Procedures.
- An Example: Reverse.
- Lab.

Local Procedure Bindings

- Today’s class will focus not on something new, but on a better way to do something old: Define helper procedures.
- We frequently want to define procedures that are only available to certain other procedures (typically to one or two other procedures).
- We call such procedures local procedures
- Most local procedures can be done with let and let*.
- However, neither let nor let* works for recursive procedures.
- When you want to define a recursive local procedure, use letrec.
- When you want to define only one, you can use a weird variant of let.
**letrec**

- A letrec expression has the format

```
(letrec ((name_1 exp_1)
    (name_2 exp_2)
    ...
    (name_n exp_n))
  body)
```

- A letrec is evaluated using the following series of steps.
  - First, enter `name_1` through `name_n` into the binding table. (Note that no corresponding values are entered.)
  - Next, evaluate `exp_1` through `exp_n`, giving you results `result_1` through `result_n`.
  - Finally, update the binding table (associating `name_i` and `result_i` for each reasonable `i`.

- Not that `its meaning is fairly similar to that of let, except that the order of entry into the binding table is changed.

**Named let**

- Named let is somewhat stranger, but is handy for some problems.
- Named let has the format

```
(let name
  ((param_1 exp_1)
   (param_2 exp_2)
   ...
   (param_n exp_n))
  body)
```

- The meaning is as follows:
  - Create a procedure with formal parameters `param_1 ... param_n` and body `body`.
  - Name that procedure `name`.
  - Call that procedure with actual parameters `exp_1` through `exp_n`.
- Yes, that’s right, we’ve packaged together the procedure definition and the procedure call.
- In effect, we’re just doing

```
(letrec ((name (lambda (param_1 ... param_n)
    body))
  (name val_1 ... val_n))
```
An Example

- As an example, let’s consider the problem of writing `reverse` (which I hope you recall from the exam).
- A first version, without local procedures

```
(define reverse
  (lambda (lst)
    (reverse-kernel lst null)))
(define reverse-kernel
  (lambda (remaining so-far)
    (if (null? remaining)
      so-far
      (reverse-kernel (cdr remaining) (cons (car remaining) so-far)))))
```

- The principle of encapsulation suggests that we should make `reverse-kernel` a local procedure.

```
(define reverse
  (letrec ((kernel
              (lambda (remaining so-far)
                (if (null? remaining)
                  so-far
                  (kernel (cdr remaining) (cons (car remaining) so-far)))))
            (lambda (lst)
                      (kernel lst null))))
```

- The pattern of “create a kernel and call it” is so common that the named let exists simply as a way to write that more concisely.

```
(define reverse
  (lambda (lst)
    (let kernel ((remaining lst)
                 (so-far null))
      (if (null? remaining)
        so-far
        (kernel (cdr remaining) (cons (car remaining) so-far)))))
```

Lab

- Start the lab.
- Finish it on your own time.