Class 21: Local Procedure Bindings

Held: Friday, 29 September 2006

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.

Due
- Exam 1.

Assignments
- Homework 8: Intersection.

Notes:
- Any EC events this weekend?
- The Tuesday extra returns next week with “Installing Linux”.

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

  \[
  \text{letrec } \left( \begin{array}{c}
  (\text{name}_1 \, \text{exp}_1) \\
  (\text{name}_2 \, \text{exp}_2) \\
  \vdots \\
  (\text{name}_n \, \text{exp}_n)
  \end{array} \right) \text{body}
  \]

- A letrec is evaluated using the following series of steps.
  - First, enter \text{name}_1 through \text{name}_n into the binding table. (Note that no corresponding values are entered.)
  - Next, evaluate \text{exp}_1 through \text{exp}_n, giving you results \text{result}_1 through \text{result}_n.
  - Finally, update the binding table (associating \text{name}_i and \text{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

  \[
  \text{let name} \\
  \left( \begin{array}{c}
  (\text{param}_1 \, \text{exp}_1) \\
  (\text{param}_2 \, \text{exp}_2) \\
  \vdots \\
  (\text{param}_n \, \text{exp}_n)
  \end{array} \right) \text{body}
  \]

- The meaning is as follows:
  - Create a procedure with formal parameters \text{param}_1 \ldots \text{param}_n and body \text{body}.
  - Name that procedure \text{name}.
  - Call that procedure with actual parameters \text{exp}_1 through \text{exp}_n.
- Yes, that’s right, we’ve packaged together the procedure definition and the procedure call.
- In effect, we’re just doing

  \[
  \text{letrec } \left( \begin{array}{c}
  (\text{name} \left( \lambda \left( \begin{array}{c}
  \text{param}_1 \\
  \vdots \\
  \text{param}_n
  \end{array} \right) \text{body})
  \end{array} \right) \\
  (\text{name} \, \text{val}_1 \ldots \text{val}_n)
  \]

```
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

```scheme
(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.

```scheme
(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.

```scheme
(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.