

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

```
(letrec ((name1 exp1)
        (name2 exp2)
        ...
        (namen expn))
  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
  ((param1 exp1)
   (param2 exp2)
   ...
   (paramn expn))
  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 (param1 ...
                     paramn)
                   body)))
  (name val1 ... valn))
```

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.

Copyright © 2006 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.