

Class 31: Naming Local Procedures

Held: Wednesday, March 17, 2010

Summary: We explore why and how one writes local recursive procedures.

Related Pages:

- EBoard.
- Lab: Local Procedure Bindings.
- Reading: Local Procedure Bindings.

Notes:

- Penguins!
- Reading for Friday: Randomized Drawing.
- Are there any final questions on Exam 2?
- I regret to inform you that there will not be an assignment due on the Wednesday after break.
- EC for Thursday's CS Extra (4:30 in 3821).
- EC for Friday's CS Table (noon in JRC PDR).
- Today's outline has much more text than I plan to cover.

Overview:

- Why have local procedures.
- Creating local procedures with `letrec`.
- Creating local procedures with named `let`.
- An example: `reverse`.

Husk and Kernel Programming

- Particularly for recursive procedures, it is inefficient to check preconditions at every recursive call
 - If the preconditions were met for the first call, they should be met for every subsequent call.
- Hence, programmers tend to use what I refer to as “Iowa’s Great Contribution to Programming”: The Husk-and-Kernel approach
 - The husk checks the preconditions and, if all preconditions are met, calls the kernel.
 - The kernel does the real work.
- Corn serves as the metaphor: The husk protects the kernel, and the kernel is the valuable part.
 - And no, Husk-and-Kernel programming was not invented in Iowa.

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 variant of `let` called "named 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`.
- 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

- Work on the the lab.

Copyright © 2007-10 Janet Davis, Matthew Kluber, Samuel A. Rebelsky, and Jerod Weinman. (Selected materials copyright by John David Stone and Henry Walker and used by permission.) This material is based upon work partially supported by the National Science Foundation under Grant No. CCLI-0633090.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. 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.