

Class 20: Introduction to Local Bindings

Held: Wednesday, 27 September 2006

Summary: Today we consider how to bind names to values using Scheme's various kinds of `let` expressions.

Related Pages:

- EBoard.
- Lab: Naming Values with Local Bindings.
- Reading: Naming Values with Local Bindings.

Notes:

- If you are too stressed about current campus events to work on the exam, please talk to me individually and we'll work out a new due date.
- EC: Campus Musician Showcase, 10 p.m. Thursday, Bob's.
- Today's outline has much more information than I plan to cover in class. You may find it useful to review it.

Overview:

- Why name things.
- Naming things with `let`.
- Naming things with `let *`.
- Naming procedures.
- Lab.

The Problem: Naming Values

- As we've seen in many problems, it helps to name the values that we use within our procedure. Why?
 - It can make the code more readable because the name tells us something about the role the value plays.
 - It can make the code more efficient, because it allows us to avoid recomputing a value.
- Consider the inefficient but elegant `closest-to-zero`

```
(define closest-to-zero
  (lambda (lst)
    (cond
      ; If there's only one element in the list, it's closest to zero
      ((null? (cdr lst)) (car lst))
      ; If the current element is closer to zero than the closest
      ; remaining thing, use that
      ((< (abs (car lst)) (abs (closest-to-zero (cdr lst)))))
      (car lst))
      ; Otherwise, use the thing in the remainder closest to zero
      (else (closest-to-zero (cdr lst))))))
```

- Note that `closest-to-zero` may get called repeatedly with the same parameters. (We'll check that with some sample code.)
- Instead of making two calls to `closest-to-zero`, we can make one by naming the result and using it twice. One possibility is to use a helper procedure

```
(define closest-to-zero
  (lambda (lst)
    (if (null? (cdr lst))
        (car lst)
        (closer-to-zero (car lst) (closest-to-zero (cdr lst))))))
```

```
(define closer-to-zero
  (lambda (guess1 guess2)
    (if (< (abs guess1) (abs guess2)) guess1 guess2)))
```

- Another possibility is to *name* the result of the recursive computation.

```
(define closest-to-zero
  (lambda (lst)
    (cond
      ((null? (cdr lst)) (car lst))
      ("Compute (closest-to-zero (cdr lst)) and call it guess"
       (if (< (abs (car lst)) (abs guess)) (car lst) guess))))
```

- Another reason to name things is that we might want to create helper procedures and only make them available to the current procedure.

Naming Things with `let`

- You name things with `let`.
- `let` has the form

```
(let ((name1 exp1)
      (name2 exp2)
      ...
      (namen expn))
  body)
```

- `let` has the meaning:
 - Evaluate all the expressions.
 - Update the binding table to associate each name with the corresponding value.

- Evaluate *body* using the updated binding table.
- Eliminate all the bindings just created.
- You can use `let` in a simple expression:

```
(define values (list 1 4 2 4 1 5 9))
(let ((largest (max values))
      (smallest (min values)))
  (/ (+ largest smallest) 2))
```

- More frequently, we use `let` within a procedure. Here's a new version of `closest-to-zero` that uses `let`.

```
(define closest-to-zero
  (lambda (lst)
    ; If there's only one element in the list, it's closest to zero
    (if (null? (cdr lst)) (car lst)
        ; Otherwise, find the remaining element closest to zero and
        ; call it guess
        (let ((guess (closest-to-zero (cdr lst))))
          ; Choose the closer to zero of the first element and guess
          (if (< (abs (car lst)) (abs guess)) (car lst) guess)))))
```

Sequencing Bindings with `let*`

- If we want to bind some things in sequence, we need to use `let*` rather than `let`.
- `let*` has the form

```
(let* ((name1 exp1)
      (name2 exp2)
      ...
      (namen expn))
  body)
```

- `let*` has the meaning:
 - Evaluate *exp*₁.
 - Update the binding table to associate *name*₁ with that value.
 - Evaluate *exp*₂.
 - Update the binding table to associate *name*₂ with that value.
 - ...
 - Evaluate *exp*_n.
 - Update the binding table to associate *name*_n with that value.
 - Evaluate *body* using the updated binding table.
 - Eliminate all the bindings just created.

Naming Helper Procedures

- You can also use this technique to name helper procedures. However, it does not work for recursive helper procedures.
- We'll return to recursive helper procedures later.
- Here's yet another version of `closest-to-zero` that makes `closer-to-zero` a helper.

```
(define closest-to-zero
  (let ((closer-to-zero
        (lambda (guess1 guess2)
          (if (< (abs guess1) (abs guess2)) guess1 guess2))))
    (lambda (lst)
      (if (null? (cdr lst))
          (car lst)
          (closer-to-zero (car lst) (closest-to-zero (cdr lst)))))))
```

- Here's an example of the use of a procedure that uses a non-recursive helper procedure that checks whether a value of any type is exact

```
;;; Procedure:
;;; exact-average
;;; Parameters:
;;;   num1, an exact number
;;;   num2, an exact number
;;; Purpose:
;;;   Average the two numbers.
;;; Produces:
;;;   average, an exact number
;;; Preconditions:
;;;   num1 is an exact number [Verified]
;;;   num2 is an exact number [Verified]
;;; Postconditions:
;;;   Guess.
(define exact-average
  (lambda (num1 num2)
    (let ((verify?
          (lambda (val) (and (number? val) (exact? val)))))
      (cond
        ((not (verify? num1))
         (error "exact-average" "first parameter is a non-number"))
        ((not (verify? num2))
         (error "exact-average" "second parameter is a non-number"))
        (else (/ (+ num1 num2) 2))))))
```

Lab

- Do the lab.

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.