Extra topics, Week 08

Overview

- Prelim.
  - Admin.
  - About the quiz.
- Q & A
  - Husks and kernels, using local kernels
  - Relationships between named let and letrec, with some emphasis on the syntax of named let.
  - When do let and letrec give different results?

Admin

- Upcoming extra credit
  - CS table tomorrow
  - Grinnell prize week event
  - Grinnell town hall, noon or 7:30 on Nov. 13

What’s might be on the quiz?

- Recursion.
- Local recursive procedure bindings with letrec or named let.
- Numeric recursion.
- Precondition checking with error.
- Testing.

Husks and Kernels

- Goal of husk and kernel is that
  - kernel does the real work
  - husk protects the kernel
- Not always precise
- Sometimes variants
  - husk sets up additional parameters for kernel

Let's do a problem or problems:

*Extract all of the even elements in a list of numbers*
Procedure:

all-evens

Parameters:

lst, a list of integers

Purpose:

Find all the even numbers in the list

Produces:

evens, a list

Preconditions:

[See parameters]

Postconditions:

if n is in evens, then n is in lst
if n is in lst and is even, then n is in evens

(define all-evens
  (lambda (lst)
    (cond
      [(not (list? lst))
        (error "all-evens: requires a list, given" lst))
      [(not (all-integer? lst))
        (error "all-evens: JC says this requires a list of integers, given" lst)]
      [else
        DO THE REAL WORK]))

Filling in the kernel: Requires another (recursive) procedure

(define kernel
  (lambda (lst)
    (cond
      [(null? lst)
        null]
      [(even? (car lst))
        (cons (car lst) (kernel (cdr lst)))]
      [else
        (kernel (cdr lst))])))

We can put this in with a letrec or a named let.
Letrec looks just like a let, but is for recursive procedure

(define all-evens
  (letrec ([kernel
    (lambda (lst)
      (cond
        [(null? lst)
          null]
        [(even? (car lst))
          (cons (car lst) (kernel (cdr lst)))]
        [else
          (kernel (cdr lst))]])]
    (lambda (lst)
      (cond
        [(not (list? lst))
          (error "all-evens: requires a list, given" lst)]
        [(not (all-integer? lst))
          (error "all-evens: JC says this requires a list of integers, given" lst)]
        [else
          (kernel lst)])))
Named let. Form

Goal: Define a procedure and start the procedure running with a particular set of inputs

(let NAME ([param-0 initial-value]
            [param-1 initial-value])
  body)

(define all-evens
  (lambda (lst)
    (cond
     [(not (list? lst))
      (error "all-evens: requires a list, given" lst))
     [(not (all-integer? lst))
      (error "all-evens: JC says this requires a list of integers, given" lst)]
     [else
      (let kernel ([l lst])
        (cond
         [(null? l)
          null]
         [(even? (car l))
          (cons (car l) (kernel (cdr l))))
         [else
          (kernel (cdr l))])))])

Named Let vs. Letrec

- Named let
  - Puts related things together
  - Can become easier to read
- Letrec
  - Useful if we need more than one recursive procedure
  - May be easier to read

Is there something for which let and letrec give different results?

(define x 2)
(let ([x (* x x)])
  (list x))
(letrec ([x (* x x)])
  (list x))

Utility: image-draw-line! sometimes fails when the y value is > 1million

(let ((image-draw-line! (lambda (image x1 y1 x2 y2) (cond [(> y1 1000000) (error "You’ll crash the gmip")][(!> y2 1000000) (error "You’ll crash the gimpe")][else (image-draw-line! image x1 y1 x2 y2)])))