
More Practise Writing Procedures

; Returns the sum of all elements of *lst*.
; Precondition:

```
(define sumlist
  (lambda (lst)
    (cond ( (null? lst) 0 )
          ( else (+ (car lst)
                    (sumlist (cdr lst))) )
          )
    )
  )
```

30

; Returns a list containing the length of each
; list in *ll*.
; Precondition:

```
(define lengths
  (lambda (ll)
    (cond ( (null? ll) '() )
          ( else (cons (length (car ll))
                      (lengths (cdr ll))) )
          )
    )
  )
```

32

; Returns a list containing the sum for each
; list in *ll*.
; Precondition:

```
(define sums
  (lambda (ll)
    (cond ( (null? ll) '() )
          ( else (cons (sumlist (car ll))
                      (sums (cdr ll))) )
          )
    )
  )
```

31

; Returns a list containing the cdr of each
; list in *ll*.
; Precondition:

```
(define cdrs
  (lambda (ll)
    (cond ( (null? ll) '() )
          ( else (cons (cdr (car ll))
                      (cdrs (cdr ll))) )
          )
    )
  )
```

33

Passing Procedures as Arguments

Example 1: *consall*

We could have different procedures *sums*, *lengths*, and *cdrs*. Each one does a certain thing to every element of a list.

The code is very repetitive. And if we want to do something else to every element of a list, we need to write yet another procedure.

Instead, we can write one general procedure.

```
; Returns a list containing the result of applying fn  
; to each element of lst.  
; Precondition:
```

```
(define consall  
  (lambda (fn lst)  
    (cond ( (null? lst) '() )  
          ( else (cons (fn (car lst))  
                       (consall fn (cdr lst))) )  
          )  
    )  
  )  
)
```

34

Map

Scheme has a built-in procedure that is a more general version of our *consall*; it is called *map*.

Example:

```
1 ]=> (map length '( (1 2 3) (5) () (1 2 3 4) ))  
;Value 7: (3 1 0 4)
```

```
; equivalent to:  
1 ]=> (consall length '( (1 2 3) (5) () (1 2 3 4) ))  
;Value 8: (3 1 0 4)
```

map can also apply a procedure that takes more than one argument; you supply a list of values for the first argument, then a list of values for the second, etc.

```
1 ]=> (map > '(99 2 5 -1) '(88 4 5 3))  
;Value 9: (#t () () ())
```

36

Sample run

```
eddie 1% scheme -load fcnpars.scm  
Scheme Microcode Version ...  
  
;Loading "fcnpars.scm" -- done  
  
1 ]=> (sums '((3 1) (6 29 4) (5) ()))  
;Value 1: (4 39 5 0)  
  
1 ]=> (lengths '((3 1) (6 29 4) (5) ()))  
;Value 2: (2 3 1 0)  
  
1 ]=> (cdrs '((3 1) (6 29 4) (5)))  
;Value 3: ((1) (29 4) ())  
  
1 ]=> (consall sumlist '((3 1) (6 29 4) (5) ()))  
;Value 4: (4 39 5 0)  
  
1 ]=> (consall length '((3 1) (6 29 4) (5) ()))  
;Value 5: (2 3 1 0)  
  
1 ]=> (consall cdf '((3 1) (6 29 4) (5)))  
;Value 6: ((1) (29 4) ())
```

35

Example 2 (of passing procedures): *prune*

Suppose we want a procedure that will test every element of a list and return a list containing only those that pass the test.

We want it to be very general: it should be able to use any test we might give it. How will we tell it what test to apply?

What should a procedure call look like?

Example: Prune out the elements of *myList* that are not atoms.

Now let's write the procedure.

37

```
; Return a new list containing only the elements of lst
; that pass the test.
; Precondition:
```

```
(define prune
  (lambda (lst test)
    (cond ((null? lst) '())
          ((test (car lst))
           (cons (car lst)
                 (prune (cdr lst) test)))
          (else (prune (cdr lst) test))))))
```

Sample run

```
1 ]=> (define (atom? x) (not (pair? x)))
;Value: atom?

1 ]=> (prune '((3 1) 4 (x y z) (x) y ()) atom?)
;Value 12: (4 y ())

1 ]=> (prune '(() (a b c) (1 2) () (()) (x (y w) z)) null?)
;Value 13: (() ())
```

38

Write calls to *prune* that will prune *myList* in these ways:

- Prune out elements that are null.
- (Assume *myList* contains lists of integers.) Prune out elements whose minimum is not at least 50. Hint: there is a built-in *min* procedure.
- (Assume *myList* contains lists.) Prune out elements that themselves have more than 2 elements.

This is becoming tedious. We need to declare a procedure for each possible test we might dream up.

39

Back to Unnamed Procedures

Exercise: What is the value of each of these Scheme expressions?

```
( (lambda (x) (cons x ())) 'y )
```

```
( (lambda (x y) (> (length x) (length y)))
  '(a b c) '(d) )
```

```
( (lambda (x) (list? x)) '(lambda (x) (list? x)) )
```

```
( (lambda (x y) (append x y)) '(1 2) '(3 4 5) )
```

40

Using unnamed procedures when calling *prune*

```
1 ]=> (define myList
      '(() (a b c) (1 2) () (()) (x (y w) z)))
;Value: myList
```

```
1 ]=> (prune myList (lambda (x) (not (null? x))))
;Value 4: ((a b c) (1 2) (()) (x (y w) z))
```

```
1 ]=> (define myList '((59 72 40) (85 70 88 56)))
;Value: myList
```

```
1 ]=> (prune myList (lambda (x) (> (apply min x) 50)))
;Value 5: ((85 70 88 56))
```

```
1 ]=> (define myList '((59 72 40) (85 70 88 56)))
;Value: myList
```

```
1 ]=> (prune myList (lambda (x) (> (apply min x) 50)))
;Value 6: ((85 70 88 56))
```

41

Uses of unnamed lambda-expressions

Example: Suppose we have tables of data (represented using Scheme lists), and procedures that can do things like select out the rows of a given table that pass some test.

Suppose we want the *user* to be able to specify any criterion they might want. Examples:

- Retrieve students where `gpa > 3.0`
- Retrieve courses where `classSize < 100`
- Retrieve profs where `building = SF`

It would be tedious to write a named procedure for every single criterion that the user might specify.

Instead, we can have the *program* construct an appropriate lambda-expression, based on the user's query.

42

Example 3 (of passing procedures):

bubblesort

```
; Precondition: smaller? is a procedure that can be
; applied to any two elements of lst. It should return
; #t iff the first argument is "smaller" than the second.
```

```
(define bubblesort
  (lambda (lst smaller?)
    (helper lst smaller? (- (length lst) 1))
  )
)
```

```
; Bubblesorts the first n elements of lst. Returns a
; new list with the first n elements of lst sorted,
; followed by the rest of lst unchanged.
; Precondition: n < (length list).
```

```
(define helper
  (lambda (lst smaller? n)
    (if (<= n 0)
        lst
        (helper (bubbleFirstN lst smaller? n)
                 smaller?
                 (- n 1))
    )
  )
)
```

43

```
; Does a single "bubble run".
; Precondition: n < (length lst)
(define bubbleFirstN
  (lambda (lst smaller? n)
    (cond ((= n 0) lst)
          ((smaller? (car lst) (cadr lst))
           (cons (car lst)
                  (bubbleFirstN (cdr lst)
                                smaller?
                                (- n 1))
           )
          (else (cons (cadr lst)
                      (bubbleFirstN (cons (car lst)
                                           (cdr lst))
                                    smaller?
                                    (- n 1))
                      )
          )
    )
  )
)
```

Is our *bubblesort* procedure $O(n^2)$, where *n* is the length of the original list, as it should be?

Can you improve the style and/or comments?

44

Sample run of procedure bubblesort

```
eddie 1% scheme
Scheme Microcode Version ...

1 ]=> (load "sort.scm")

;Loading "sort.scm" -- done
;Value: bubblesort

1 ]=> (bubblesort '(3 4 1 5 0 2 3) <)

;Value 1: (0 1 2 3 3 4 5)

1 ]=> (bubblesort
      '((a b c) (a) (1 2 3 4) () (z z z) (y y))
      (lambda (x y) (< (length x) (length y))))
;Value 2: ((()) (a) (y y) (z z z) (a b c) (1 2 3 4))

1 ]=> (trace helper)
;No value

1 ]=> (trace bubbleFirstN)
;No value
```

45

; Note: #[compound-procedure ... fn] has been changed
 ; to #[fn] and the spacing has been reduced to make
 ; the slide more readable.

```
1 ]=> (bubblesort '(3 4 1 5 0 2 3) <)

[Entering #[helper]           Args: (3 4 1 5 0 2 3) #[<] 6]
[Entering #[bubblefirstn]     Args: (3 4 1 5 0 2 3) #[<] 6]
...
[(3 1 4 0 2 3 5)
  <= #[bubblefirstn] Args: (3 4 1 5 0 2 3) #[<] 6]
[Entering #[helper]           Args: (3 1 4 0 2 3 5) #[<] 5]
[Entering #[bubblefirstn]     Args: (3 1 4 0 2 3 5) #[<] 5]
...
[(1 3 0 2 3 4 5)
  <= #[bubblefirstn] Args: (3 1 4 0 2 3 5) #[<] 5]
[Entering #[helper]           Args: (1 3 0 2 3 4 5) #[<] 4]
[Entering #[bubblefirstn]     Args: (1 3 0 2 3 4 5) #[<] 4]
...
[(1 0 2 3 3 4 5)
  <= #[bubblefirstn] Args: (1 3 0 2 3 4 5) #[<] 4]
[Entering #[helper]           Args: (1 0 2 3 3 4 5) #[<] 3]
[Entering #[bubblefirstn]     Args: (1 0 2 3 3 4 5) #[<] 3]
...
[(0 1 2 3 3 4 5)
  <= #[bubblefirstn] Args: (1 0 2 3 3 4 5) #[<] 3]
[Entering #[helper]           Args: (0 1 2 3 3 4 5) #[<] 2]
[Entering #[bubblefirstn]     Args: (0 1 2 3 3 4 5) #[<] 2]
...
```

Eval

eval is a built-in procedure of two arguments:
 (*eval expr env*).

eval evaluates *expr* in the environment *env* and returns the result. We can use '()' or (*the-environment*) to evaluate *expr* in the current environment.

Before *eval* is called, *expr* already gets evaluated once, so in effect *eval* cause *expr* to be evaluated twice.

Note: Dybvig describes a version of *eval* that precedes the current Scheme standard. The examples work if you add () as a second argument to *eval*.

```
[(0 1 2 3 3 4 5)
  <= #[bubblefirstn] Args: (0 1 2 3 3 4 5) #[<] 2]
[Entering #[helper]           Args: (0 1 2 3 3 4 5) #[<] 1]
[Entering #[bubblefirstn]     Args: (0 1 2 3 3 4 5) #[<] 1]
...
[(0 1 2 3 3 4 5)
  <= #[bubblefirstn] Args: (0 1 2 3 3 4 5) #[<] 1]
[Entering #[helper]           Args: (0 1 2 3 3 4 5) #[<] 0]
[(0 1 2 3 3 4 5)
  <= #[helper]           Args: (0 1 2 3 3 4 5) #[<] 0]
[(0 1 2 3 3 4 5)
  <= #[helper]           Args: (0 1 2 3 3 4 5) #[<] 1]
[(0 1 2 3 3 4 5)
  <= #[helper]           Args: (0 1 2 3 3 4 5) #[<] 2]
[(0 1 2 3 3 4 5)
  <= #[helper]           Args: (1 0 2 3 3 4 5) #[<] 3]
[(0 1 2 3 3 4 5)
  <= #[helper]           Args: (1 3 0 2 3 4 5) #[<] 4]
[(0 1 2 3 3 4 5)
  <= #[helper]           Args: (3 1 4 0 2 3 5) #[<] 5]
[(0 1 2 3 3 4 5)
  <= #[helper]           Args: (3 4 1 5 0 2 3) #[<] 6]
;Value 3: (0 1 2 3 3 4 5)
```

Example: Write an expression, to go inside procedure *blah*, that will return the sum of the elements of *lst*.

```
(define blah
  (lambda (lst)
    ))
```

Redefining the if-statement in Scheme

Define procedure *myif*, without using the **if** syntactic form:

```
(myif (> num 0) ; If num is greater than 0
      num       ; then return num
      (* num -1) ; else return num * -1.
)
```

50

Problem: We shouldn't evaluate the arguments for *then* and *myelse* at the time of the call to *myif*.

Solution: Quote them.

```
(define absolute
  (lambda (num)
    (myif (> num 0)
          'num
          '(* num -1))
  )
)
```

```
(define len
  (lambda (x)
    (myif (list? x)
          '(length x)
          '-1)
  )
)
```

52

```
(define myif
  (lambda (test then myelse)
    (cond (test then)
          (else myelse))
  )
)
```

```
(define absolute
  (lambda (num)
    (myif (> num 0)
          num
          (* num -1))
  )
)
```

```
(define len
  (lambda (x)
    (myif (list? x)
          (length x)
          -1)
  )
)
```

51

Problem: The arguments for *then* and *myelse* are echoed back to us, unevaluated.

Solution: Eval them inside *myif*.

```
(define myif
  (lambda (test then myelse)
    (cond (test (eval then ()))
          (else (eval myelse ())))
  )
)
```

53

Problem: It may not make sense to evaluate the arguments for *then* and *myelse* in the context of *myif*.

```
(define myif
  (lambda (test then myelse env)
    (cond (test (eval then env))
          (else (eval myelse env))
          )
    )
  )
```

```
(define len
  (lambda (x)
    (myif (list? x)
          '(length x)
          '-1
          (the-environment)
          )
    )
  )
```

Conclusion: Good thing **if** is a syntactic form!