More Practise Writing Procedures

; Returns the sum of all elements of \texttt{lst}.
; Precondition:

\begin{verbatim}
(define sumlist
  (lambda (lst)
    (cond ( (null? lst) 0 )
          ( else (+ (car lst)
                    (sumlist (cdr lst))))
    )
  )
\end{verbatim}

; Returns a list containing the sum for each list in \texttt{ll}.
; Precondition:

\begin{verbatim}
(define sums
  (lambda (ll)
    (cond ( (null? ll) '() )
          ( else (cons (sumlist (car ll))
                     (sums (cdr ll))) )
    )
  )
\end{verbatim}

; Returns a list containing the length of each list in \texttt{ll}.
; Precondition:

\begin{verbatim}
(define lengths
  (lambda (ll)
    (cond ( (null? ll) '() )
          ( else (cons (length (car ll))
                     (lengths (cdr ll))) )
    )
  )
\end{verbatim}

; Returns a list containing the cdr of each list in \texttt{ll}.
; Precondition:

\begin{verbatim}
(definecdrs
  (lambda (ll)
    (cond ( (null? ll) '() )
          ( else (cons (cdr (car ll))
                     (cdrs (cdr ll))) )
    )
  )
\end{verbatim}
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 ist.
; Precondition:
(define consall
  (lambda (fn ist)
    (cond ((null? ist) '())
          (else (cons (fn (car ist))
                      (consall fn (cdr ist)))))
    )
)

Sample run

eddie 1% scheme -load fcnpars.scm
Scheme Microcode Version ...

; Loading "fcnpars.scm" -- done
1 ]=> (cons all '((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) ()))

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) '((8 4 5 3))
;Value 9: (#t () () )

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.
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.

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 '((69 72 40) (85 70 88 56))) ;Value: mylist
1 ]=> (prune mylist (lambda (x) (not (null? x))))) ;Value 4: ((a b c) (1 2) () (()) (x (y w) z))

1 ]=> (define mylist '((69 72 40) (85 70 88 56))) ;Value: mylist
1 ]=> (prune mylist (lambda (x) (> (apply min x) 60))) ;Value 5: ((85 70 88 56))

1 ]=> (define mylist '((69 72 40) (85 70 88 56))) ;Value: mylist
1 ]=> (prune mylist (lambda (x) (> (apply min x) 60))) ;Value 6: ((85 70 88 56))

---

**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) )
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.

Example 3 (of passing procedures):

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

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

; 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 (cadr lst) smaller? (- n 1)))
          )
          (else (cons (cadr lst)
                      (bubbleFirstN (cons (car lst) (cadr 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?

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 4 5)
1 ]=> (bubblesort
   '(((a b c) (a) (1 2 3 4)) (z z) (y y))
   (lambda (z y) (< (length x) (length y))))
;Value 2: ((a) (y y) (z z) (a b c) (1 2 3 4))
1 ]=> (trace helper)
;No value
1 ]=> (trace bubbleFirstN)
;No value
```

42

43

44

45
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`.

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

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

  )
  )
```
Redefining the if-statement in Scheme

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

```
(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)
  )
)
```

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)
  )
)
```

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 ()))))
)
```

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

\[
\text{(define myif} \\
\quad \text{(lambda} \ (\text{test then myelse env}) \\
\quad \quad \text{(cond} \ (\text{test} \ (\text{eval} \ \text{then env})) \\
\quad \quad \quad \quad \text{(else} \ (\text{eval} \ \text{myelse env})) \\
\quad \quad \)) \\
\text{)}
\]

\[
\text{(define len} \\
\quad \text{(lambda} \ (\text{x}) \\
\quad \quad \text{(myif} \ (\text{list? x}) \\
\quad \quad \quad \quad \text{'(length} \ \text{x}) \\
\quad \quad \quad \quad \quad \quad '-1 \\
\quad \quad \quad \quad \text{(the-environment}) \\
\quad \quad \)) \\
\text{)}
\]

Conclusion: Good thing if is a syntactic form!