More Practice Procedures

- addToEnd: add an element to the end of a list.
  (addToEnd 'a '(a b c)) ⇒ (a b c a)

- revLists: given a list of lists, form new list consisting of all elements of the sublists in reverse order.
  ((1 2) (3 4 5) (6)) ⇒ (6 5 4 3 2 1)

- revListsAll: given a list of lists, form new list from reversal of elements of each list.
  ((1 2) (3 4 5) (6)) ⇒ (2 1 5 4 3 6)

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.

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

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

Sample run

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

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

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

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.

Passing Anonymous Procs

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

1 ]=> (prune (lambda (x) (not (null? x))) myList)
;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 (lambda (x) (> (apply min x) 50)) myList)
;Value 5: ((85 70 88 56))


1 ]=> (define myList '((23 34) (10 1 3 4) () (2 3 4)))
;Value: mylist

1 ]=> (prune (lambda (x) (<= (length x) 2)) myList)
;Value 6: ((23 34) ())
```

A2 Digression

Some things that may help w/ A2

- defining procedures w/ successive arguments
- continuations
- `set!`

Procs w/ Successive Args

You can use *curry-ing* (named after Haskell Curry) to define procedures to take *successive arguments* rather than simultaneous ones.

For example, take the function `f(x,y)` (or `(f x y)` in Scheme notation). Our objective is to define a procedure `curry` such that:

```
(((curry f) x) y)  =  (f x y)}
```

We do so using nested lambda expressions.

```
(define curry
  (lambda (f)
    (lambda (x)
      (lambda (y)
        (f x y)))
  ))

(define times (curry *))
(define plus (curry +))
(define double (times 2))
(define triple (times 3))
```

Now: `((time 2) 3) => 6`

vs. `(* 2 3) => 6`

I.e., `((times 2) 3) = (((curry *) 2) 3)`

Another Curry-ing Example

Successive Arguments:

```
(define ltplus3
   (lambda (x)
      (lambda (y) (< x (+ y 3)))
))

((ltplus3 5) 3)   => #t
((ltplus3 6) 3)   => ()
```

Simultaneous Arguments:

```
(define ltplus3_2
   (lambda (x y)
      (< x (+ y 3))))

(ltplus3_2 5 3)   => #t
(ltplus3_2 6 3)   => ()
(ltplus3 5 3)     => ERROR
((ltplus3_2 6) 3) => ERROR
```

Scheme: continuations

What is a continuation?

- The current continuation at any point in the execution of a program is an abstraction of the rest of the program.
- A continuation of the evaluation of an expression E in a surrounding context C represents the entire future of the computation, which waits for the value of E.

Context C and expression E	**Intuitive continuation of E in C**
(+ 5 **(* 4 3)**)	*The adding of 5 to the value of **E***
(cons 1 (cons 2 (cons 3 **'()**)))	*The consing of 3, 2 and 1 to the value of **E***
(define x 5) (if (= 0 x) 'undefined (remainder (* **(+ x 1)** (- x 1)) x))	*The multiplication of **E** by x - 1 followed by a division by x*

Scheme: continuations (cont)

What is a continuation?

- A continuation of the evaluation of an expression E in a surrounding context C represents the entire future of the computation, which waits for the value of E

A more precise notation of the continuation of E:

Context C and expression E	**Continuation of E in C**
(+ 5 **(* 4 3)**)	*(lambda (e) (+ 5 e))*
(cons 1 (cons 2 (cons 3 **'()**)))	*(lambda (e) (cons 1 (cons 2 (cons 3 e))))*
(define x 5) (if (= 0 x) 'undefined (remainder (* **(+ x 1)** (- x 1)) x))	*(lambda (e) (remainder (* e (- x 1)) x))*

set!

Global Assignment (Generally EVIL!)

When an assignment statement is applied to variables (i.e., memory locations) that are:

- maintained AFTER the procedure call is completed.
- are used for their values in this or other procedures.

it **violates referential transparency** and destroys the ability to statically analyze source code (formally and intuitively).

E.g.,

```
(define g 10)            ; define global variable g

(define (func a)
   (set! g (* g g))      ; globally assign g=g*g
   (+ a g)
)

]=> (func 7)
107

]=> (func 7)
10007                    ; BAD!
```

set! (cont.)

`(set! <var> <expr>)`

alters the value of an existing binding for *var*. Evaluates *expr* then assigns *var* to *expr*.

Useful for implementing counters, state change or for caching values.

References: Dybvig

Summary: Functional Pgming

- Pure functional languages:
  - Referential transparency
  - No assignment
  - No iteration, only recursion
  - Implicit storage management (garbage collection)
  - Functions are values
- λ-calculus
- LISP, Common LISP, Scheme
- Built-In Procedures
- Lists (cons cells, proper/improper)
- Read-eval-print loop
- Inhibiting + Activating evaluation (quote, eval)
- Procedure definition and lambda expressions

- Conditionals (if, cond)
- Equality Checking (eq?, =, equal?, eqv?)
- Recursion (practice, practice)
- Efficiency Concerns
  - helper procedures
  - let, let*, ...
  - accumulators (did not discuss)
- Higher-order functions (map, apply, reduce)
- Passing Procedures, Returning Procedures
- Anonymous Procedures

Optional Material

The material after this point is optional. I have covered this (and much more) in other years, but because of the reorganization of the course, I was unable to get to it this year.

More on Efficiency

We previously saw that helper procedures and local variables (`let`, `let*`) can improve the efficiency of a Scheme program. A third way of improving efficiency (sometimes) is through the use of an accumulator.

Trace the following two procedures. What is their complexity?

```
(define (rev1 lst)
   (cond ((null? lst) '())
         (else (append
                  (rev1 (cdr lst))
                  (list (car lst)))
          )
    )
)
```

More on Efficiency

Using an **accumulator** `new`.

```
(define (rev2 lst new)
   (cond ((null? lst) new)
         (else (rev2 (cdr lst)
               (cons (car lst) new)))
   )
)
```

A Lesson in (In)efficiency: Fibonacci

Problem: Compute the n^{th} Fibonacci number.

Recall, the *Fibonacci numbers are an infinite sequence of integers 0, 1, 1, 2, 3, 5, 8, etc., in which each number is the sum of the two preceding numbers in the sequence.*

Let's define a simple fibonacci procedure:

```
(define fib
; (fib n) returns the nth Fibonacci number
; Pre: n is a non-negative integer
  (lambda (n)
```

Simple Fibonacci

```
(define fib
; (fib n) returns the nth Fibonacci number
; Pre: n is a non-negative integer
   (lambda (n)
     (cond ((= n 0) 1)
           ((= n 1) 1)
           (else (+ (fib (- n 1)) (fib (- n 2))))
     )
   )
)
```

Problem: Procedure is doubly recursive.
Complexity is exponential!

(fib 4) calls (fib 3) and (fib 2),
(fib 3) calls (fib 2) and (fib 1), etc.

Trace of Simple Fibonacci

```
1 ]=> (fib 3)

[Entering #[compound-procedure 1 fib]
    Args: 3]
[Entering #[compound-procedure 1 fib]
    Args: 1]
[1
      <== #[compound-procedure 1 fib]
    Args: 1]
[Entering #[compound-procedure 1 fib]
    Args: 2]
[Entering #[compound-procedure 1 fib]
    Args: 0]
[1
      <== #[compound-procedure 1 fib]
    Args: 0]
[Entering #[compound-procedure 1 fib]
    Args: 1]
[1
      <== #[compound-procedure 1 fib]
    Args: 1]
[2
      <== #[compound-procedure 1 fib]
    Args: 2]
[3
      <== #[compound-procedure 1 fib]
    Args: 3]
;Value: 3
```

Faster Fibonacci

Hint: Use an **accumulator** (or two!) to store intermediate values.

```
; (fast-fib p1 p2 i n) returns the nth Fibonacci number
; Pre:  n>=0 is an integer, 0<=i<=n is an integer,
; p1 is the (i-1)th Fib number (or 0 if i is 0), and
; p2 is the ith Fib number.
(define fast-fib
   (lambda (p1 p2 i n)
```

Faster Fibonacci (cont.)

```
; (fast-fib p1 p2 i n) returns the nth Fibonacci number
; Pre:  n>=0 is an integer, 0<=i<=n is an integer,
; p1 is the (i-1)th Fib number (or 0 if i is 0), and
; p2 is the ith Fib number.
(define fast-fib
   (lambda (p1 p2 i n)
          (if (= i n)
               p2
               (fast-fib p2 (+ p1 p2) (+ i 1) n))
   )
)

; (fib n) returns the nth Fibonacci number
; Pre: n is a non-negative integer
(define fib
    (lambda (n)
       (fast-fib 0 1 0 n)
    )
)
```

Time complexity of this fib procedure is linear!

Lesson: Accumulators are useful for writing efficient code. (e.g., factorial, reverse, etc.)

Trace of Faster Fibonacci

```
1 ]=> (fib 3)

[Entering #[compound-procedure 2 fib]
    Args: 3]
[Entering #[compound-procedure 3 fast-fib]
    Args: 0
          1
          0
          3]
[Entering #[compound-procedure 3 fast-fib]
    Args: 1
          1
          1
          3]
[Entering #[compound-procedure 3 fast-fib]
    Args: 1
          2
          2
          3]
[Entering #[compound-procedure 3 fast-fib]
    Args: 2
          3
          3
          3]
[3
      <== #[compound-procedure 3 fast-fib]
    Args: 2
          3
          3
          3]
[3
      <== #[compound-procedure 3 fast-fib]
    Args: 1
          2
          2
          3]
[3
      <== #[compound-procedure 3 fast-fib]
    Args: 1
          1
          1
          3]
[3
      <== #[compound-procedure 3 fast-fib]
    Args: 0
          1
          0
          3]
[3
      <== #[compound-procedure 2 fib]
    Args: 3]
;Value: 3
```

Other Useful Scheme: Strings

Sequences of characters.
Written within double quotes, e.g., "hi mom"

Useful string predicate procedures:

```
(string=? <string1> <string2> ...)
(string<? <string1> <string2> ...)
(string<=? ...
etc.
```

Case-insensitive versions:

```
(string-ci=? <string1> <string2> ...)
(string-ci<? <string1> <string2> ...)
(string-ci<=? ...
```

Other string procedures:

```
(string-length <string>)
(string->symbol <string>)
(symbol->string <symbol>)
(string->list <string>)
(list->string <list>)
```

Other Useful Scheme Procedures

Input and Output

```
(read ...)         ; reads and returns an expression
(read-char ...)    ; reads & returns a character
(peek-char ...)    ; returns next avail char w/o updating
(char-ready? ...) ; returns #t if char has been entered
(write-char ...)   ; outputs a single character
(write <object> ...) ; outputs the object
(display <object> ...) ; outputs the object (pretty)
(newline)              ; outputs end-of-line
```

```
;; Display a number of objects, with a space between each.
(define display-all
   (lambda lst
      (cond ((null? lst) ())
            ((null? (cdr lst)) (display (car lst)) ())
            (else (display (car lst)) (display " ")
                  (apply display-all (cdr lst))))
   )
)

(define lst '(a b c d))
(display-all "List: " lst "\n")  ; List (a b c d) <cr>
(apply display-all lst)    ; a b c d
```

Reading/writing files

```
(open-input-file)
(open-output-file)
```

Syntactic Forms

`if`, `begin`, `or`, `and` are useful **syntactic forms**.

They have *lazy evaluation*, i.e., their subexpressions are not evaluated until required.

Let's look at lazy evaluation and how to exploit it.

```
(if (= n 0)
     (display "oops")
     (/ 1 n))
```

`if` is evaluated left to right. The "else part" is only evaluated as necessary, so `(/ 1 n)` is only evaluated if the conditional expression is false.

Imagine if `if` were implemented as a procedure. We'd be in trouble!

```
(begin
    (display "this is line 1 of the message")
    (display "this is line 2 of the message")
    #f
)
```

`begin` evaluates it subexpressions from left to right and returns the value of the last subexpression.

Syntactic Forms (cont.)

```
(or) => #f
(or (= 0 1) (= 0 2) (= 0 0))  => #t
(or #f) => #f
(or #f #t) => #t
(or #f 'a #f) => a   (treated as #t in a conditional)
```

`or` evaluates its subexpressions from left to right until either (a) one expression is true, or (b) no more expressions are left. In case (a), the value is true, in (b) the value is false.

Important subtlety: Every Scheme object is considered to be either true or false by conditional experssions and by the procedure `not`. Only `#f` (i.e., ()) is considered false; **all other objects are considered true**.

```
(and) => #t
(and (= 0 0) (= 0 1) (= 0 2)) => #f
(and #f) => #f
(and #t #t) => #t
(and #t #f) => #f
(and 'a 'b 'c) => c  (treated as #t in a conditional)
```

`and` evaluates its subexpressions from left to right until (a) one expression is false, or (b) no more expressions are left. In case (a), the value is false, in (b) the value is true.

Clever Exploitation of Syntactic Forms and Lazy Evaluation

```
(define (validate-bindings expr bindings)
    (cond ((...) ...)
          ((...) ...)
          ((symbol? expr)
             (debug-display "Symbol:" expr)
             (or (get-binding expr bindings)
                 (builtin? expr)
                 (begin
                    (display-error 'unbound expr)
                     #f
                 )
             )
         ((...) ...)

   etc.
)
```

As soon as one of the conditions in the `or` statement is `true`, Scheme stops evaluating. This can be used to advantage. Similarly with `and` and evaluation to `false`.