Polymorphic and Monomorphic Functions

- *Polymorphic* functions can be applied to arguments of many forms
- The function `length` is polymorphic: it works on lists of numbers, lists of symbols, lists of lists, lists of anything
- The function `square` is monomorphic: it only works on numbers

Higher-Order Procedures

Procedures **as input values**:

```
(define (all-num lst)
  (or (null? lst)
      (and (number? (car lst))
           (all-num (cdr lst))))
  )
(define (all-num-f f lst)
  (cond ((all-num lst) (f lst))
        (else 'error))
  )
1 ]=> (all-num-f abs-list '(1 -2 3))
;Value 1: (1 2 3)

1 ]=> (all-num-f abs-list '(1 a))
;Value: error
```


Higher-Order Procedures

Procedures **as returned values**:

```
(define (plus-list x)
  (cond ((number? x)
         (lambda (y) (+ (sum-n x) y)))
        ((list? x)
         (lambda (y) (+ (sum-list x) y)))
        (else (lambda (x) x))
  ))
1 ]=> ((plus-list 3) 4)
;Value: 10

1 ]=> ((plus-list '(1 3 5)) 5)
;Value: 14
```


Built-In Higher-Order Procedures: `map`

There is a built-in procedure `map`. Let's define our own restricted version first....

```
(define (mymap f l)
  (cond ((null? l) '())
        (else (cons (f (car l))
                    (mymap f (cdr l))))
  ))
```

- `mymap` takes two arguments: a function and a list
- `mymap` builds a new list whose elements are the result of applying the function to each element of the (old) list

Higher-order Procedures: `map`

- Example:

  `(mymap abs '(-1 2 -3 4))` ⇒ `(1 2 3 4)`

  `(mymap (lambda (x) (+ 1 x)) '(-1 2 -3))` ⇒ `(0 3 -2)`

- The built-in `map` will produce the same results, but note that the built-in `map` can take <u>more than two</u> arguments:

  `(map cons '(a b c) '((1) (2) (3)))` ⇒ `((a 1) (b 2) (c 3))`

What's Wrong Here??

```
1 ]=>
(define (atomcount s)
  (cond ((null? s) 0)
        ((atom? s) 1)
        (else (+ (map atomcount s)))
  ))
;Value: atomcount
1 ]=> (atomcount '(a b))

;The object (1 1), passed as an argument
;to +, is not the correct type.
..
2 error>
```

Why doesn't this work?

Using `eval` to Correct the Problem

```
(define (atomcount s)
 (cond ((null? s) 0)
       ((atom? s) 1)
       (else
        (eval
         (cons '+ (map atomcount s)) '()))
))
1 ]=> (atomcount '(a b))

;Value: 2

1 ]=> (atomcount '((1) (2 3 (4)) ((((5))))))

;Value: 5
```


Limitations of Using `eval`

BUT: `eval` only works in the current definition of `atomcount` because numbers evaluate to themselves.

```
1 ]=> (+ 1 2 3)
;Value: 6

1 ]=> (cons '+ '(1 2 3))
;Value 12: (+ 1 2 3)

1 ]=> (eval (cons '+ '(1 2 3)) '())
;Value: 6
```

Using `eval` to Evaluate Expressions

```
1 ]=> (append '(a) '(b))
;Value 13: (a b)
1 ]=> (cons 'append '((a) (b)))
;Value 14: (append (a) (b))

1 ]=> (eval (cons 'append '((a) (b))) '())
;Unbound variable: b
...
1 ]=> (cons 'append '( '(a) '(b) ))
;Value 15: (append (quote (a)) (quote (b)))

1 ]=> (eval
        (cons 'append '( '(a) '(b))) '())
;Value 16: (a b)
```

Too complicated!!

Applying Procedures with `apply`

```
1 ]=> (apply + '(1 2 3))
;Value: 6
1 ]=> (apply append '((a) (b)))
;Value 5: (a b)

1 ]=>
(define (atomcount s)
 (cond ((null? s) 0)
       ((atom? s) 1)
       (else
        (apply + (map atomcount s)))))

;Value: atomcount
1 ]=>  (atomcount '(a (b) c))
;Value: 3
```


Higher-order Procedures: `my-reduce`

```
(define (my-reduce op l id)
 (if (null? l)
     id
     (op (car l)
         (my-reduce op (cdr l) id))
))
```

A binary ↦ n-ary procedure.

The `my-reduce` procedure takes a binary operation and applies it right-associatively to a list of an arbitrary number of arguments.

NOTE: `my-reduce` is not equivalent to `apply`.

Higher-order Procedures: `my-reduce`

`(my-reduce + '(1 2 3) 0)` ⇒ 6:

```
(my-reduce + '(1 2 3) 0)
(+ 1 (my-reduce + '(2 3) 0))
(+ 1 (+ 2 (my-reduce + '(3) 0)))
(+ 1 (+ 2 (+ 3 (my-reduce + '() 0))))
(+ 1 (+ 2 (+ 3 0)))
6
```

Note: `(+ 1 2 3)` ⇒ 6

`(my-reduce / '(24 6 2) 1)` ⇒ 8:

```
(my-reduce / '(24 6 2) 1)
(/ 24 (my-reduce / '(6 2) 1))
(/ 24 (/ 6 (my-reduce / '(2) 1)))
(/ 24 (/ 6 (/ 2 (my-reduce / '() 1))))
(/ 24 (/ 6 (/ 2 1)))
8
```

Note: `(/ 24 6 2)` ⇒ 2

Higher-order Procedures: `my-reduce`

Given `union`, which takes two lists representing sets and returns their union:

```
1 ]=> (apply union '((1 3)(2 3 4)))
;Value 21: (1 2 3 4)

1 ]=> (apply union '((1 3)(2 3)(4 5)))
;The procedure #[compound-procedure union]
;has been called with 3 arguments;
;it requires exactly 2 arguments.

1 ]=> (reduce union '((1 3)(2 3)(4 5)) '())
;Value 22: (1 2 3 4 5)
```

Question: How would you have to change `my-reduce` to be able to take `intersection` as its function argument?

Important

Note that Scheme has a built-in higher-order procedure `reduce` that is different from `my-reduce`. You may use `my-reduce` in assignments and tests. In assignments, you would of course have to define it by copying the code provided here. In tests, you may use it without defining it.

Example Practice Procedures

- cdrLists: given a list of lists, form new list giving all elements of the cdr's of the sub-lists.
  ((1 2) (3 4 5) (6)) ⇒ (2 4 5)
- swapFirstTwo: given a list, swap the first two elements of the list.
  (1 2 3 4) ⇒ (2 1 3 4)
- swapTwoInLists: given a list of lists, form new list of all elements in all lists, with first two of each swapped.
  ((1 2 3)(4)(5 6)) ⇒ (2 1 3 4 6 5)
- addSums: given a list of numbers, sum the total of all sums from 0 to each number.
  (1 3 5) ⇒ 22

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) ())
```


`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

set! (cont.)

```
(define cons-count 0)
(define cons
  (let ((old-cons cons))
     (lambda (x y)
        (set! cons-count (+ cons-count 1))
        (old-cons x y)
     )
  )
)

(cons 'a '(b c)) => (a b c)
cons-count => 1
(cons 'a (cons 'b (cons 'c '()))) => (a b c)
cons-count => 4
```

What's the problem?

set! (cont)

```
(define count
  (let ((next 0))
     (lambda ()
        (let (v next))
          (set! next (+ next 1))
          v))))

count => 0
count => 1
```

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`.