A Lesson in (In)efficiency: Fibonacci

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

Recal, 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 Procedures

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.

```
(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

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

count => 0
count => 1
```


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 (cdr lst))    ; a b c d
```

Reading/writing files

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