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.* $F(0) = 0$,$F(1) = 1$, *etc.*

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
; We added some display statements so we could see what was ha
; Of course trace also works!
   (lambda (n)
     (display "entering fib")
     (display n)
     (newline)
     (cond ((= n 0) 0)
           ((= n 1) 1)
           (else (+ (fib (- n 1)) (fib (- n 2))))
     )
   )
)
```

Problem: Procedure is doubly recursive.
Complexity is exponential!

Let's Run Simple Fibonacci

```
1 ]=> (fib 6)
entering fib6
entering fib4
entering fib2
entering fib0
entering fib1
entering fib3
entering fib1
entering fib2
entering fib0
entering fib1
entering fib5
entering fib3
entering fib1
entering fib2
entering fib0
entering fib1
entering fib4
entering fib2
entering fib0
entering fib1
entering fib3
entering fib1
entering fib2
entering fib0
entering fib1
;Value: 8

1 ]=> (fib 3)
entering fib3
entering fib1
entering fib2
entering fib0
entering fib1
;Value: 2
```

Trace of Simple Fibonacci

```
1 ]=> (trace fib)

;Unspecified return value

1 ]=> (fib 3)

[Entering #[compound-procedure 1 fib]
    Args: 3]
entering fib3
[Entering #[compound-procedure 1 fib]
    Args: 1]
entering fib1
[1
      <== #[compound-procedure 1 fib]
    Args: 1]
[Entering #[compound-procedure 1 fib]
    Args: 2]
entering fib2
[Entering #[compound-procedure 1 fib]
    Args: 0]
entering fib0
[0
      <== #[compound-procedure 1 fib]
    Args: 0]
[Entering #[compound-procedure 1 fib]
    Args: 1]
entering fib1
[1
      <== #[compound-procedure 1 fib]
    Args: 1]
[1
      <== #[compound-procedure 1 fib]
    Args: 2]
[2
      <== #[compound-procedure 1 fib]
    Args: 3]
;Value: 2
```

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 ith Fibonacci number
; p2 is the i+1th Fibonnacci 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 ith Fibonacci number
; p2 is the i+1th Fibonnacci number
;
(define fast-fib
   (lambda (p1 p2 i n)
        (display "entering fast-fib ")
        (display i)
        (newline)
        (if (= i n)
            p1
            (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.)

Let's Run Faster Fibonacci

```
1 ]=> (fib 6)
entering fast-fib 0
entering fast-fib 1
entering fast-fib 2
entering fast-fib 3
entering fast-fib 4
entering fast-fib 5
entering fast-fib 6
;Value: 8


1 ]=> (fib 3)
entering fast-fib 0
entering fast-fib 1
entering fast-fib 2
entering fast-fib 3
;Value: 2
```


Trace of Faster Fibonacci

```
1 ]=> (trace fast-fib)

;Unspecified return value

1 ]=> (fib 3)

[Entering #[compound-procedure 1 fast-fib]
    Args: 0
          1
          0
          3]
entering fast-fib 0
[Entering #[compound-procedure 1 fast-fib]
    Args: 1
          1
          1
          3]
entering fast-fib 1
[Entering #[compound-procedure 1 fast-fib]
    Args: 1
          2
          2
          3]
entering fast-fib 2
[Entering #[compound-procedure 1 fast-fib]
    Args: 2
          3
          3
          3]
entering fast-fib 3
[2
```

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

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! It's a syntactic form so it's evaluation is different.

Syntactic Forms (cont.)

(begin *expr1 expr2 ...*)

`begin` evaluates it subexpressions from left to right and returns the value of the last subexpression. `begin` can be used to sequence assignment statements, inputs/outputs, or other operations that cause side effects. E.g.,

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


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`.

When Lazy Evaluation isn't or friend

Problem: Sometimes lazy evaluation works *against* you.

Challege with `validate-bindings` is that it's a predicate procedure, so it must return #t/() depending upon whether the expression has valid bindings, but you must go through the *e*ntire list, even after you generate your first (). How do you do this?

Hint: There is a construct in Scheme that forces evaluation of a series of expressions before performing some operation on it. (there are many, actually!) Let's think of one we've seen in class and use it.