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:

```scheme
(define fib
 (if (<= n 1)
     n
     (+ (fib (- n 2)) (fib (- n 1))))
)
```

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.

---

Simple Fibonacci

```scheme
(define fib
 (if (<= n 1)
     n
     (+ (fib (- n 2)) (fib (- n 1)))))
)
```

Trace of Simple Fibonacci

```scheme
1 => (fib 3)
[Entering #<procedure: fib>]
Arguments: 3
[1] => #<procedure: fib>
Arguments: 3
[2] => #<procedure: fib>
Arguments: 2
[3] => #<procedure: fib>
Arguments: 1
```

Faster Fibonacci

```scheme
(define fast-fib
 (if (<= n 1)
     n
     (+ (fast-fib (- n 2)) (fast-fib (- n 1))))
)
```

Trace of Faster Fibonacci

```scheme
1 => (fast-fib 3)
[Entering #<procedure: fast-fib>]
Arguments: 3
[1] => #<procedure: fast-fib>
Arguments: 3
Arguments: 2
[3] => #<procedure: fast-fib>
Arguments: 1
```

Faster Fibonacci (cont.)

```scheme
(define fast-fib
 (if (<= n 1)
     n
     (+ (fast-fib (- n 2)) (fast-fib (- n 1))))
)
```

Note: Time complexity of this fib procedure is linear.

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

---

Other Useful Scheme Procedures

Global Assignment (Generally Evil!)

When an assignment statement is applied to variables (i.e., memory locations) that are:
- Several layers deep in a procedure call (inside another procedure), and
- are used later in other procedures, it makes reference transparent and prevents the ability to statically analyze source code (formally and intuitively).

Example:

```scheme
(define g 100) ; define global variable g
(let* ((g (+ g g)) (x g)) ; globally assign g to g + g
  (g (* g 2)))
```

```scheme
100
```

```scheme
(let* ((g (+ g g)) (x g)) ; globally assign g to g + g
  (g (* g 2)))
```

```scheme
200
```
Strings

Sequences of characters, written within quotes, e.g., "HI mom"

Useful string predicate procedures:

- `(string? <string1> <string2> ...)`: returns `true` if all arguments are strings
- `(string? <string>)`: returns `true` if `<string>` is a string
- `(string? ...)`

Case-insensitive versions:

- `(string-ci? <string1> <string2> ...)`: returns `true` if all arguments are strings
- `(string-ci? <string>)`: returns `true` if `<string>` is a string
- `(string-ci? ...)`

Other string procedures:

- `(string-length <string>)`: returns the length of the string
- `(string-swapcase <string>)`: returns a case-swapped version of the string
- `(string>list <string>)`: returns a list of characters in the string
- `(list>string <list>)`: returns a string from the list

Other Useful Scheme Procedures

Input and Output

- `(read ...)` : reads and returns an expression
- `(read char ...)` : reads and returns a character
- `(peek char ...)` : returns the next character without advancing the position
- `(char>=syms ...)` : returns `true` if char has been entered
- `(write char ...)` : writes a single character
- `(write obj ...)` : writes an object
- `(display obj ...)` : outputs the object (pretty)
- `(each)` : outputs `each` writing

:: Display a number of objects, with a space between each.

```scheme
(define display-w
  (lambda list
    (cond ((null? list) ())
      ((null? (cdr list)) (display (car list)) (display " "))
      (else (display (car list)) (display " ")
            (apply display-w (cdr list))))))
```

Reading/writing files

```scheme
(open input file) (open output file)
```