Scheme Basics

```
> (butfirst '(help!))
()
```

[The butfirst of a *sentence* containing one word is all but that word, i.e., the empty sentence. (BUTFIRST 'HELP!) without the inner parentheses would be butfirst of a word, and would return ELP!, the most common wrong answer. The second most common wrong answer, although we didn't take off for it, was '() -- that is, the correct answer with a quote in front. If you are reading '() in programs as if it were the name of the empty sentence, then you don't understand quoting.]

```
> ((if 3 * +) 4 5)
20
```

[Since 3 is true, the value of the inner parenthesized expression is the multiplication function.]

```
> (let ((+ -))(+ 8 2))
6
```

[Names of primitives are just ordinary variables that happen to have a predefined value, but that can be overridden by a local binding. Notice that that isn't true about names of special forms; you can't say (LET ((DEFINE +))) etc.]

```
> (let ((+ -)(- +))(- 8 2))
10
```

[Names of primitives are just ordinary variables that happen to have a predefined value, but that can be overridden by a local binding. Notice that that isn't true about names of special forms; you can't say (LET ((DEFINE +)) ...) etc. Since all the bindings in a LET are done at once, not in sequence, the local value for the name "-" is the *original* meaning of +, not the local meaning.]

```
> ((lambda (z) (- z z)) (random 10))
0
```

[Some people said "0 if applicative order, some random value if normal order." That's true, but the point of the question was to see if you know what Scheme does!]

Applicative vs. Normal Order

If an expression produces an error, just say "error": if it returns a procedure, just say "procedure."

Given the following definitions:

```
(define (marco x) 'done)

(define (polo) (polo))
```

(a) What will be the result of the expression `(marco (polo))`

in normal order? done

in applicative order? Error

(b) What will be the result of the expression `(marco polo)`

in normal order? done

in applicative order? done

Iterative vs Recursive

Is the following procedure iterative or recursive:

```
(define (foo sent)
   (define (foo-helper sent prev)
       (cond ((empty? (bf sent)) (* (first sent) prev))
             (else (se (* (first sent) prev)
                       (foo-helper (bf sent) (first sent))))))
   (foo-helper sent 1))
```

If recursive, rewrite as iterative. If iterative, rewrite as recursive.

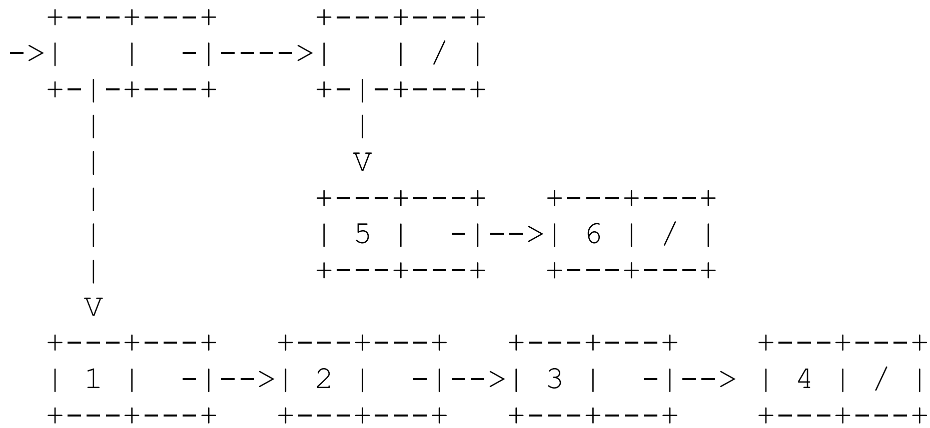
This is recursive. The iterative version looks like:

```
(define (foo sent)
     (define (foo-iter sent prev so-far)
          (cond ((empty? sent) so-far)
                (else (foo-iter (bf sent) (first sent)
                         (se so-far (* (first sent) prev))))))
     (foo-iter sent 1 '()))
```

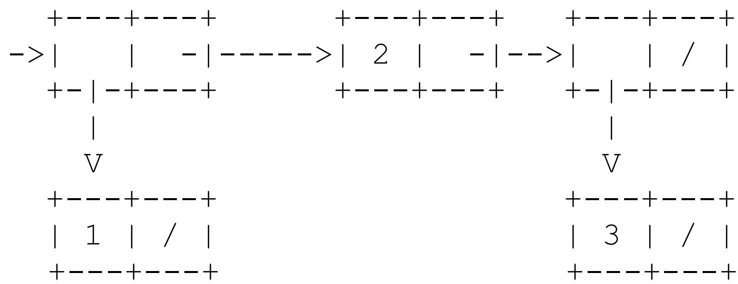
Boxes and Pointers

Draw the box and pointer diagrams for the following expressions

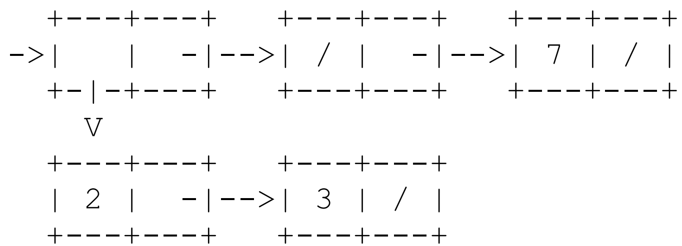
```
>(list (append (list 1 2) (cons 3 (cons 4 '( )))) (cons 5 6))
```




```
>(cons (cons 1 '( )) (list 2 (list 3)))
```




```
(define x (list (list 1 2 3) (list 5) (cons 6 7)))
(define y (map cdr x))
```



Orders of Growth

1.

```
(define (garply n)
  (if (< n 2)
      n
      (garply (garply (- n 1)))))
```

Solution:

garply is a procedure that takes linear O(n) time. When garply is called with a positive number as argument, garply will call itself with progressively smaller arguments, and so (garply n) will yield (garply (garply (garply (garply...(garply 1))...)))) in n time. (garply 1) takes constant time to evaluate and returns 1, and so each containing procedure call will receive 1 as argument and return 1. The chain collapses to the value 1, and since there are n procedure calls in the chain, each of which takes constant time to evaluate, garply takes 2n total time, or O(n) time, to evaluate.

2.

```
(Define (garply n)
    (if (< n 2)
    n
    (+ (foo n)
       (garply (- n 1)))))
```

Assuming foo is defined somewhere, please circle TRUE or FALSE. and in one sentence explain your choice.

True or False: We have enough information to determine the order of growth of garply.

True or False: NO matter how foo is defined, garply will always have an order of growth greater than or equal to THETA(n)

True or False: garply has an order of growth in Theta(n^2) if foo is defined as follows:

```
(define (foo n)
  (if (< n 100)
      121
      (+ (* n 100) (foo (- n 1))))
```

Solutions: False, True, True

Data Abstraction

Time for some math! We're going to use Scheme to represent polynomials of *arbitrary* degree. We are going to represent a polynomial as a list, where the first element in the list is the coefficient of the zeroth order term, the second element in the list is the coefficient of the first order term, etc. For example, the polynomial 7 – 2x + 3x^3 would be represented as the list (7 -2 0 3).

Assume that we already have the constructor make-polynomial defined as follows: (define (make-poly ls) ls). This seems a little silly but bear with us for now.

Write the following utility procedures:

1. get-n, which takes a polynomial and a number n as argument and returns the coefficient of its nth term (ex: (get-n (make-poly '(7 -2 0 3)) 1) returns -2).
2. degree, which takes a polynomial as argument and returns its degree (ex: (degree (make-poly '(7 -2 0 3))) returns 3).

```
(define (get-n poly n)
          (if (= n 0)
               (car poly)
               (get-n (cdr poly) (- n 1))))

(define (degree poly)
          (- (length poly) 1))
```

Write evaluate-at, which takes a polynomial and a number x as argument and evaluates the polynomial at x. You can use the procedure exp, which takes a number n and another number m and raises n to the m power (ex: (exp 2 3) returns 8).

```
(define (evaluate-at poly x)
      (define (evaluate-helper count poly x result)
            (cond ((> count (degree poly)) result)
                (else (evaluate-helper (+ count 1) poly x
                                               (+ (* (get-n poly count)
                                                     (exp x count))
                                                  result)))))
      (evaluate-helper 0 poly x 0))
```

What is the current run time of degree? Suppose we want to make degree run in constant time. How would we change our representation to make this possible (keep in mind make-poly should still take a list as argument)? Would we have to rewrite get-n and degree?

Degree currently runs in linear time (since it uses length, which is a linear time operation). One way to change our representation is to instead represent a polynomial is a pair, whose car is its degree and whose cdr is a list of its coefficients. Now computing the degree takes a single call to car, which takes constant time. We would have to rewrite get-n and degree, but the key point is that we would **not** have to rewrite evaluate-at.

Higher Order

Write a procedure called product-list that takes in a list of numbers (a1, a2, a3, …) and returns another list (a1, a1a2, a1a2a3, …). You may use higher-order functions and recursion.

```
(define (product-list ls)
  (if (null? ls)
      '()
      (cons (car ls)
            (map (lambda (num) (* num (car ls)))
                 (product-list (cdr ls))))))
```

Deep Member

Write the predicate procedure, deep-member?, which takes in a word and a deep-list as arguments and returns #t if the word appears somewhere within the list and #f otherwise.

```
(define (deep-member? wd ls)
  (cond ((null? ls) #f)
        ((atom? (car ls))
         (or (equal? (car ls) wd))
             (deep-member? wd (cdr ls))))
        (else (or (deep-member? (car ls))
                  (deep-member? (cdr ls))))))
```

Trees

A Tree is considered degenerate if every parent node has at most one child. Write the predicate is-degenerate? which takes a Tree as argument and returns true if the Tree is degenerate, and false otherwise.

```
(define (is-degenerate? Tree)
            (cond ((null? Tree) #t)
                  ((null? (children Tree)) #t)
                  ((> (length (children Tree)) 1) #f)
                  (else (is-degenerate? (car (children Tree))))))
```

Binary Search Trees

Consider a procedure named get-rank for binary search trees. Get-rank takes two arguments, a BST, and a number n, and returns the number of elements in the BST that are smaller than n. Assume that we already have a procedure count-nodes written, which takes a BST and returns the number of nodes in the BST.

```
(define (get-rank Bst n)
    (cond ((null? Bst) 0)
          ((< n (datum Bst)) (get-rank (left-subtree Bst) n))
          (else (+ 1 (get-rank (right-subtree Bst) n)
                     (count-nodes (left-subtree Bst))))))
```

Also count-nodes can be written as:

```
(define (count-nodes Bst)
      (if (null? Bst)
            0
            (+ 1 (count-nodes (left-subtree Bst))
                 (count-nodes (right-subtree Bst)))))
```