Recursion: Five Steps to a Recursive Function

1. **Strategy:** How to reduce the problem?
2. **Header:**
   - What info needed as input and output?
   - Write the function header.
     Use a noun phrase for the function name.
3. **Spec:** Write a method specification in terms of the parameters and return value.
   Include preconditions.
4. **Base Cases:**
   - When is the answer so simple that we know it without recursing?
   - What is the answer in these base case(s)?
   - Write code for the base case(s).
5. **Recursive Cases:**
   - Describe the answer in the other case(s) in terms of the answer on smaller inputs.
   - Simplify if possible.
   - Write code for the recursive case(s).

Recursive Scheme Procedures: Sum-N

Parameter: integer n $\geq$ 0.

Result: sum of integers from 0 to n.

```
(define (sum-n n)

  (cond (                              )

        (else                          )

   )
 )
```


Recursive Scheme Procedures: Length

```
(define (length x)

   ))
```

This is called “cdr-recursion.”

Note: There is a built-in `length` procedure.

Length (cont.)

```
1 ]=> (trace length)

;No value

1 ]=> (length '(a b c))

[Entering #[compound-procedure 5 length]
    Args: (a b c)]
[Entering #[compound-procedure 5 length]
    Args: (b c)]
[Entering #[compound-procedure 5 length]
    Args: (c)]
[Entering #[compound-procedure 5 length]
    Args: ()]
[0
      <== #[compound-procedure 5 length]
    Args: ()]
[1
      <== #[compound-procedure 5 length]
    Args: (c)]
[2
      <== #[compound-procedure 5 length]
    Args: (b c)]
[3
      <== #[compound-procedure 5 length]
    Args: (a b c)]
;Value: 3
```


Recursive Scheme Procedures: Abs-List

- `(abs-list '(1 -2 -3 4 0))` ⇒ `(1 2 3 4 0)`
- `(abs-list '())` ⇒ `()`

```
(define (abs-list lst)

)
```


Recursive Scheme Procedures: Append

```
(append '(1 2) '(3 4 5)) ⇒ (1 2 3 4 5)
(append '(1 2) '(3 (4) 5)) ⇒ (1 2 3 (4) 5)
(append '() '(1 4 5)) ⇒ (1 4 5)
(append '(1 4 5) '()) ⇒ (1 4 5)
(append '() '()) ⇒ ()

(define (append x y)

)
```

Note: There is a built-in `append` procedure.

Lists Revisited

Recall the Cons Cell Representation:

The *pair* or *cons cell* is the most fundamental of Scheme's structured object types.

A **list** is a sequence of **pairs**; each pair's cdr is the next pair in the sequence.

The cdr of the last pair in a **proper list** is the empty list. Otherwise the sequence of pairs forms an **improper list**. I.e., an empty list is a proper list, and and any pair whose cdr is a proper list is a proper list.

An improper list is printed in **dotted-pair notation** with a period (dot) preceding the final element of the list. A pair whose cdr is not a list is often called a **dotted pair**

Creating lists

`quote`: `'(1 (2 3) ()) => (1 (2 3) ())`
  or `(quote (1 (2 3) ())) => (1 (2 3) ())`

`list`: `(list 1 '(2 3) ()) => (1 (2 3) ())`

`cons`: Build it, piece by piece.

```
(cons 1 (cons (cons 2 (cons 3 ()))
              (cons () ())))
```

`append`: Appending lists

```
(append '(1) '(4 5)) => (1 4 5)
```

cons vs. list: The procedure `cons` actually builds *pairs*, and there is no reason that the cdr of a pair must be a list.

The procedure `list` is similar to `cons`, except that it takes an arbitrary number of arguments and always builds a proper list.

E.g., `(list 'a 'b 'c)` → `(a b c)`

Testing for Equality

- `(eq? a b)`: Returns `#t` iff `a` and `b` are the same Scheme object. (Don't use `eq?` with numbers!)
- `(= a b)`: Returns `#t` iff `a` and `b` are numerically equal. Pre: `a` and `b` must evaluate to numbers.
- `(eqv? a b)`: Similar to `eq?`, but works for numbers and characters. More expensive than `eq?`, however.
- `(equal? a b)`: Returns `#t` iff `a` and `b` have the same structure and contents. Thus, `equal?` recursively tests for equality. The most expensive equality predicate.

Recommended Reading:
Dybvig §6.1, 2nd ed. (available online), or
Dybvig §6.2, 3rd ed.

Testing for Equality (cont.)

The `eq?` predicate doesn't work for lists.

Why not?

1. `(cons 'a '())` makes a new list
2. `(cons 'a '())` makes a(nother) new list
3. `eq?` checks if its two args are *the same*
4. `(eq? (cons 'a '()) (cons 'a '()))` evaluates to () (ie, #f)

Lists are stored as pointers to the first element (car) and the rest of the list (cdr).

Symbols are stored uniquely, so `eq?` works on them.

Equality Checking for Lists

For lists, need a comparison procedure to check for the same **structure** in two lists. How might you write such a procedure?

```
(define (myequal? x y)
  (or (and (atom? x) (atom? y) (eq? x y))
      (and (not (atom? x)) (not (atom? y))
           (myequal? (car x) (car y))
           (myequal? (cdr x) (cdr y)))))
```

- `(equal? 'a 'a)` evaluates to `#t`
- `(equal? 'a 'b)` evaluates to ()
- `(equal? '(a) '(a))` evaluates to `#t`
- `(equal? '((a)) '(a))` evaluates to ()

Does this really work? Hint: atoms are numbers, does this work for numbers? Play around with it and with the built-in predicate procedure `equal?`.

Other Useful Predicates

- `(null? a)`: Returns `#t` iff `a` is the empty list (or `#f`, depending on the implementation).
- `(pair? a)`: Returns `#t` iff `a` is a pair, *i.e.*, a cons cell.
- `(number? a)`: Returns `#t` iff `a` is a number.
- `(min list)`: Returns the minimum of a list of numbers.
- `(max list)`: Returns the maximum of a list of numbers.
- `(even? a)`: Returns `#t` iff `a` is even.

Lots more in Dybvig §6.

Recursive Procedures: Counting

```
(define (atomcount x)
  (cond ((null? x) 0)
        ((atom? x) 1)
        (else (+ (atomcount (car x))
                 (atomcount (cdr x))))))
```

- (atomcount '(1 2)) ⇒ 2
- (atomcount '(1 (2 (3)) (5))) ⇒ 4:

```
(at '(1 (2 (3)) (5)))
(+ (at 1) (at ((2 (3)) (5))))
(+ 1 (+ (at (2 (3))) (at ((5)))))
(+ 1 (+ (+ (at 2) (at ((3)))) (+ (at (5)) (at ()))))
(+ 1 (+ (+ 1 (+ (at (3)) (at ()))) (+ (+ (at 5) (at ())) 0)))
(+ 1 (+ (+ 1 (+ (+ (at 3) (at ())) 0)) (+ (+ 1 0) 0)))
(+ 1 (+ (+ 1 (+ (+ 1 0) 0)) (+ 1  0)))
(+ 1 (+ (+ 1 (+ 1 0)) 1))
(+ 1 (+ (+ 1 1) 1))
(+ 1 (+ 2 1))
(+ 1 3)
4
```

This is called "car-cdr-recursion.'

Efficiency Issues

Problem: Evaluating the same expression twice.

Example:

```
(define (longest-nonzero x y)
  (cond ((and (null? x) (null? y)) -1)
        ((> (length x) (length y))
         (length x))
        (else (length y))

  ))
```

What can you do if there is no assignment statement?

Efficiency Issues

Solution 1: Bind values to parameters in a helper procedure.

```
(define (maximum x y)
  (cond ((> x y) x)
        (else y)
  ))

(define (longest-nonzero x y)
  (cond ((and (null? x) (null? y))  -1)
        (else
         (maximum (length x) (length y)))
  ))
```

Note: There is a built-in `max` function.

Note 2: Helper procedures are an important and useful tool!

Efficiency Issues

Solution 2: Use a `let` or `let*` construct, to create *local* variables and to bind them to expression results. The scope of these variables is limited to the scope of the let statement.

```
(let ((var1 expr1)
      ...
      (varn exprn))
      body
)
```

The variables can only be used within the body of the let.
Evaluation: expr1, ... exprn are evaluated in some **undefined order**, saved, and then assigned to var1,..varn. In our interpreter, they have the appearance of being evaluated **in parallel**.

```
(let* ((var1 expr1)
        ...
       (varn exprn))
       body
)
```

Again, the variables can only be used within the body of let*.

Evaluation: evaluation and binding is **sequential**, i.e., the evaluation of `expr1` is bound to `var1`, the evaluation of `expr2` is then bound to `var2`, etc.

Let and let* Example

```
(define a 100) (define b 200) (define c 300)

(let ((a 5)
      (b (+ a a))
      (c (+ a b)))
     (list a b c)
)
```

What does this return? What are a, b, c bound to now? (Answer: still 100, 200, 300)

```
(let* ((a 5)
      (b (+ a a))
      (c (+ a b)))
     (list a b c)
)
```

What does this return?

Note that `let*` can be simulated by nested `let`s.

```
(let ((a 5))
   (let ((b (+ a a)))
      (let ((c (+ a b)))
         (list a b c)
      )
   )
)
```


Lambda Expressions

We have often been defining procedures using the shorthand:

```
(define (square x)
    (* x x))
```

But recall that this is just shorhand for binding the variable `square` to the lambda expression `(* x x)`.

```
(define square
    (lambda (x)
       (* x x)
    )
)
```

It is often very useful to define procedures without naming them. These **anonymous procedures** can be passed as arguments, returned as arguments, bound to local variable names using let, etc. We will see further applications later when we cover higher-order procedures.

Lambda Expressions Examples

Establishing a procedure as the value of a local variable.

```
(let ((square-it (lambda (x) (* x x))))
   (list (square-it (+ 1 3))
         (square-it (* 2 5))
         (square-it 7)))       => (16 100 49)
```

`square-it` is defined only within the scope of the `let` statement.

Recall that procedures can have multiple arguments, and that we can even have **procedures as arguments** to procedures.

```
(let ((double-any (lambda (f x) (f x x))))
   (list (double-any + 25)
         (double-any cons 'a))) => (100 (a.a))
```

Dybvig §2.5 is a good reference to this material (available online). I **strongly** recommend that you read it. §4.2 may also be useful.

Lambda Expressions Examples (cont.)

The following examples are taken from Dybvig §2.5:

```
(let ((x 'a))
  (let ((f (lambda (y) (list x y))))
    (f 'b)))        returns     (a b)
```

Note that x is bound in the outer `let`. It is a *free variable* in the lambda expression. A variable that occurs free in a lambda expression should be bound by an enclosing lambda or let expression, unless the variable is (like the names of primitive procedures) bound at top level, as we discuss in the following section.

```
(let ((f (let ((x 'a))
           (lambda (y) (cons x y)))))
  (let ((x 'i-am-not-a))
    (f 'b)))  (a . b)
```

In both cases, the value of x within the procedure named f is a.

Interestingly, a let expression is just an application of a lambda expression to a set of argument expressions. I.e., the following two expressions are equivalent:

```
(let ((x 'a))
  (cons x x))

((lambda (x) (cons x x))
 'a)
```