CSE 341 - Recursion and Applicative Programming

These notes cover the following topics:

1. Recursion Introduction
2. Typical recursion templates
3. Three important classes of recursive list functions: *mapping*, *reducing*, *filtering*.
4. Higher order functions and applicative programming
5. Lexical closures

Simple examples of recursion:

```
(define (factorial n)
  (if (= 0 n)
      1
      (* n (factorial (- n 1)))))


;;  (double-each '(1 3 4))  => (2 6 8)
(define (double-each s)
  (if (null? s)
      ()
      (cons (* 2 (car s))
            (double-each (cdr s)))))
```

Debugging

- `trace` -- watch functions as they call and return
- `(debug)` -- call the debugger if you get an error expression

```
(trace factorial)
```

Rules for writing recursive functions

- Know when to stop (the base case)
- Decide how to take one step towards the base case.
- Phrase the solution in terms of one step, and a smaller version of the original problem.

For numbers, the base case is usually a small integer constant, and a smaller version of the problem is something like n-1.

For lists, the base case is usually the empty list, and a smaller version of the problem is usually the rest or "cdr" of the list. Here is a template for most recursive functions you'll write:

```
 (define (fn args)
  (if base-case
      base-value
    (compute-result (fn (smaller args)))))
```

Recursion Templates

Augmenting Recursion

Many recursive functions build up their answer bit by bit. We'll call these *augmenting* recursive functions.

```
;; general form

(define (func x)
  (if end-test
      end-value
      (augmenting-function augmenting-value (func reduced-x))))
```

Factorial is the classic example:

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

Many Scheme functions operate on lists, so we often see an important special case of augmenting recursion where our augmenting function is `cons` and our base case is the empty list. These are sometimes called *mapping* functions, because they map a function onto every element of a list. Notice that because we are using `cons`, we are actually *constructing* a brand new list and returning it. In other words, we are not changing the original list.

```
(define (double-each s)
  (if (null? s)
      ()
      (cons (* 2 (car s)) (double-each (cdr s)))))
```

With `cons` augmenting recursion, our base value (the value returned when we reach the base case) isn't always ():

```
;; still cons as augmenting-function, but end-value is not ()

(define (my-append x y)
  (if (null? x) y
      (cons (car x) (my-append (cdr x) y))))
```

A final important kind of augmenting recursion over lists is the class of *reducing* functions, because they reduce a list of elements to a single element.

```
;; sum the elements of a list

(define (sumlist x)
  (if (null? x) 0
      (+ (car x) (sumlist (cdr x)))))
```

Tail Recursion

In tail recursion, we don't build up a solution, but rather, just return a recursive call on a smaller version of the problem. Double-test tail recursion is the most common form:

```
;; general form:

(define (func x)
  (cond (end-test-1 end-value-1)
        (end-test-2 end-value-2)
        (else (func reduced-x))))

;; example: are all elements of a list positive?

(define (all-positive x)
  (cond ((null? x) #t)
        ((<= (car x) 0) #f)
        (else (all-positive (cdr x)))))
;; (all-positive '(3 5 6))  => #t
;; (all-positive '(3 5 -6))  => #f
```

Some tail recursive functions will have more than one argument. All but one of the arguments are passed along unaltered.

```
;; example: member:

(define (my-member e x)
  (cond ((null? x) #f)
        ((equal? e (car x)) #t)
        (else (my-member e (cdr x)))))
```

A less commonly seen form is *single-test tail recursion*.

Scheme compilers handle tail recursion very efficiently, as efficiently as a program that just uses loops instead of recursion. (In particular, tail recursive functions don't use stack space for every recursive call.)

Simultaneous Recursion on Several Variables

```
;; test whether two lists have the same length

(define (same-length x y)
  (cond ((and (null? x) (null? y)) #t)
        ((null? x) #f)
        ((null? y) #f)
        (else (same-length (cdr x) (cdr y)))))

;; do these two objects have the same shape?
(define (same-shape x y)
  (cond ((and (pair? x) (pair? y))
         (and (same-shape (car x) (car y))
              (same-shape (cdr x) (cdr y))))
        ((or (pair? x) (pair? y)) #f)
        (else #t)))
```

Conditional augmentation (like augmenting recursion)

A final important template is conditional augmentation, where we don't necessarily augment on every step. These functions are sometimes called *filtering* functions, because they remove elements from a list that don't pass some test.

```
(define (func x)
  (cond (end-test end-value)
        (aug-test (augmenting-function augmenting-value (func reduced-x)))
```

```
        (else (func reduced-x))))

;; example:  remove all non-positive numbers from a list

(define (positive-numbers x)
  (cond ((null? x) ())
        ((> (car x) 0) (cons (car x) (positive-numbers (cdr x))))
        (else (positive-numbers (cdr x)))))
```

Example: Insertion Sort

```
;; variation on list-consing recursion
(define (insert x s)
   (cond ((null? s) (list x))
         ((< x (car s)) (cons x s))
         (else (cons (car s) (insert x (cdr s))))))

;; augmenting recursion
(define (isort s)
   (if (null? s) ()
       (insert (car s) (isort (cdr s)))))
```

Applicative Programming

Applicative programming uses *higher order functions* -- functions that take other functions as arguments, or that return functions as the result. Scheme gets much of its expressiveness and capacity for abstraction by supporting an applicative programming style. Scheme supports functions as *first class citizens*.

Mapping a function over a list

Let's start with an example:

```
;; here is a general function that adds N to every element of a list

(define (addN-to-list alist n)
  (cond ((null? alist) ())
        (else (cons (+ n (car alist)) (addN-to-list (cdr alist) n)))))

;; here is a function that takes the cdr of every element of a list
;; (the argument must be a list of lists)
(define (getcdrs alist)
  (cond ((null? alist) ())
        (else (cons (cdr (car alist)) (getcdrs (cdr alist))))))
```

If we think about what these functions are doing at an abstract level, we see that they are applying some function to every element of a list, and returning the list of resulting values. Scheme provides us with a built in function to support this idiom.

Map is a built in Scheme function that takes a function and a list as an argument, and returns the list that results by applying that function to every element of the list.

```
(map function list)             ;; general form

(map null? '(3 () () 5))        => (() T T ())
```

```
(map round '(3 3.3 4.6 5))      => (3 3 5 5)

(map cdr '((1 2) (3 4) (5 6)))  => ((2) (4) (6))
```

Lambda

We often find ourselves mapping little functions over lists that would be annoying to write a separate definition for. For instance, if we wanted to map the double function, we'd have to do the following:

```
(define (double x) (* 2 x))

(map double '(3 4 5))         => (6 8 10)
```

Scheme provides us with the special form LAMBDA to define an anonymous function. LAMBDA really is special, because it actually creates a *lexical closure*, which is a bundling of code and a lexical environment. We'll see why this is important later.

```
(map (lambda (x) (* 2 x)) '(3 4 5))

;; the right way to define addN-to-list
(define (addN-to-list alist n)
  (map (lambda (x) (+ n x)) alist))
```

Please note that in the addN-to-list function, the body of the lambda function can refer to the variable n, which is in the lexical scope of the lambda expression. (These lambda closures are like blocks in Smalltalk.)

map can also be used with functions that take more than one argument. Examples:

```
(map + '(1 2 3) '(10 11 12))   => (11 13 15)
(map (lambda (x y) (list x y)) '(a b c) '(j k l))   => ((a j) (b k) (c l))
```

Defining Higher-Order Functions

Suppose we wanted to define the 1-argument version of map ourselves. We can do it like this:

```
(define (my-map func alist)
   (if (null? alist)
       ()
       (cons (func (car alist)) (my-map func (cdr alist)))))
```

Lexical Closures

Now that we have the code for `my-map`, we can talk about what makes LAMBDA special. Remember that a lambda expresssion evaluates to a *lexical closure*, which is a coupling of code and a lexical environment (a scope, essentially). The lexical environment is necessary because the code needs a place to look up the definitions of symbols it references. For example, look again at `addN-to-list` below.

```
(define (addN-to-list alist n)
  (my-map (lambda (x) (+ n x)) alist))
```

When the lambda expression is used in `my-map`, it needs to know where to look up the variable name n. It can get the right value for n, because it retains its lexical environment.

Nested Scopes

You can have arbitrarily nested scopes (scopes within scopes within scopes ...). Further, since function names are bound like any other variable, function names also obey the scope rules.

As an example, let's define a simple function `test`:

```
(define (test x)
   (+ x 1))
```

Evaluating `(test 10)` gives `11`.

However, if we evaluate

```
(let ((test (lambda (x) (* x 2)))
     (test 10))
```

we get 20 -- within the `let`, `test` is rebound to a different function.