Map and apply stuff below is from <http://courses.cs.washington.edu/courses/cse341/02sp/scheme/apply-eval.html>

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

**CSE 341 -- apply and eval**

These notes are meant to be read on the web. "Hidden" solutions will be printed on the page by most browsers.

**map review**

general form:  
(map *function* *argument-list*)

This evaluates to a list containing the results of applying function to each item in *argument-list*, just like Miranda's map function.

How would you use map to negate the items in the list (1 2 3 4)? (Drag over the space below to select and reveal the solution.)

;; some possible solutions

(map - '(1 2 3 4))

(map (lambda (x) (\* -1 x)) '(1 2 3 4))

(map (lambda (x) (- x)) '(1 2 3 4))

;; You can also make the list differently, e.g., (list 1 2 3 4).

A good review of evaluation rules in Scheme would be to determine what happens when the following expressions are evaluated:

(map (lambda (x) (-x)) '(1 2 3 4)) => ???

(map (lambda (x) -x) '(1 2 3 4)) => ???

hidden description: For the first expression, Scheme thinks -x is a new symbol and tries to interpet it as a function of 0 parameters. The second expression fixes the function application problem, but Scheme still thinks -x is a new symbol.

**apply**

general form:  
(apply *function* *argument-list*)

This evalutes to the result of passing arguments in *argument-list* (all at once) to *function*. (Note that this entails a single call to the function.)

Suppose cube is defined (define (cube x) (expt x 3)) for this example:

(apply cube (list 2)) => 8

(apply cube '(2)) => 8

How would you use apply to define a function sum that works just like the Miranda function of the same name?

(define (sum addends)

(apply + addends))

How would you use map and apply to define a function called apply-many, which takes a list of functions and a single argument, then runs each function on the argument, returning a list of the results? Here's an example usage, to illustrate how it should work:

(apply-many (list - cube (lambda (x) (/ x 2))) 3) => (-3 27 3/2)

hidden solution:

(define (apply-many functions arg)

(map (lambda (function)

(apply function (list arg)))

functions))

Would the following call to apply-many result in the same list as the example call given above? Why or why not? (Note the quoted function list.)

(apply-many '(- cube (lambda (x) (/ x 2))) 3)

hidden description: The first argument (the list) is evaluated into a list of symbols, rather than a list of functions, and the apply-many function definition attempts to apply the symbol - as a function, which results in an error. The first example usage given above works because list evaluates its arguments into functions before passing this list toapply-many.

It's also possible to define apply-many without using apply. How would the function definition differ from the hidden solution above?

(define (apply-many functions arg)

(map (lambda (function)

(function arg))

functions))

**eval**

general form:  
(eval *expression* user-initial-environment)

You use eval to force the interpreter to regard *expression* as an expression in Scheme syntax. The above evaluates to whatever the *expression* would evaluate to if it were called with the current set of global level bindings, as stored in a predefined environment called user-initial-environment.

This example is from Assignment 7:

(define x 5)

(define y 10)

(eval '(\* x (+ y 0)) user-initial-environment) => 50

Since the list (\* x (+ y 0)) is quoted, evaluation before being passed to eval results in a list of symbols, not the result of the multiplication and addition. It's eval that evaluates the symbol list (\* x (+ y 0)), treating it as Scheme code and using the variable bindings given above.

One easy way to remember eval is that it does the opposite of quote (usually written with the character '). If you eval something quoted like '(1 2 3), you end up with the same thing without the quote, (1 2 3).

Below is an extended example that hints at the usefulness of eval. This code is for manipulating polynomials as lists of terms, where each term is represented as a list of two numbers, the first being the coefficient and the second being the exponent.

;; given a term, returns Scheme code for evaluating the term, using

;; x as the variable

(define (term-to-expr-code term)

(list '\* (first term) (list 'expt 'x (second term))))

;; given a term, returns Scheme code for defining an anon. function

;; that takes one argument and returns the value of the term evaluated

;; for that value of x

(define (term-to-lambda-code term)

(list 'lambda '(x) (term-to-expr-code term)))

;; given a term, returns an anon. function as described above

(define (term-to-lambda term)

(eval (term-to-lambda-code term) user-initial-environment))

;; given a polynomial (list of terms), returns an anon. function that

;; takes an argument and returns the value of the represented

;; polynomial for that value of x

(define (poly-to-lambda termlist)

(let ((term-lambdas (map term-to-lambda termlist)))

(lambda (x)

(sum (apply-many term-lambdas x)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; example usage

(define term1 '(2 1)) ; 2x

(define term2 '(-3 2)) ; -x^2

(define term3 '(1 3)) ; x^3

(define poly1 (list term1 term2 term3)) ; 2x - x^2 + x^3

(term-to-expr-code term1) ; => (\* 2 (expt x 1))

(term-to-lambda-code term1) ; => (lambda (x) (\* 2 (expt x 1)))

((term-to-lambda term1) 4) ; => 8

((poly-to-lambda poly1) 4) ; => 24

;; You could imagine functions that factor, multiply, etc.,

;; polynomials in the form of term lists. Then, you could use

;; poly-to-lambda to actually evaluate the polynomials in your

;; program.