**What Would Scheme Do?**

**scm**> (define a 1)

**scm**> (define b (list 2 3))

**scm**> ‘(cons a b)

**scm**> ‘(cons ,a ,b)

**scm**> ‘(a ,(cons ,a ,b) b)

**scm**> ‘(a ,(cons a b) b)

**scm**> (eval ‘(cons ,a ,b))

**Why do we need to postpone evaluation?**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**scm>** (define-macro (mystery)

(define class ‘b)

‘class)

mystery

**scm>** (list 6 1 (mystery))

**scm>** (define class ‘a)

class

**scm>** (list 6 1 (mystery))

**and-macro**

Write **and-macro**, which takes in two expressions and evaluates them in order, returning the first false expression or the last expression.

;; **scm>** (and-macro (+ 5 6) (\* 2 3))

6

;; **scm>** (and-macro (begin (print 5) #f) (print 3))

5

#f

**(define-macro (and-macro expr1 expr2)**

`(let (\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_) (\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_))

**)**

**apply-twice**

Implement **apply-twice**, which is a macro that takes in a call expression with a single argument. It should return the result of applying the operator to the operands twice.

;; **scm>** define add-one (lambda (x) (+ x 1)))

;; add-one

;; **scm>** (apply-twice (add-one 1))

;; 3

;; **scm>** (apply-twice (print ‘hi))

;; hi

;; undefined

**(define-macro (apply-twice call-expr)**

`(let ((operator \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)

(operand \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_))

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)

**eval-all-cond**

Implement a macro, **eval-all-cond**, which takes in a cond expression and evaluates all the non-predicate sub-expressions. It should additionally not evaluate any predicate sub-expressions.

;; scm> (eval-all-cond (cond ((= 1 0) (print ‘a))

;; ((print ‘no) (print ‘b))

;; (else (print ‘c))

;; a

;; b

;; c

**(define-macro (eval-all-cond cond-expr)**

(cons \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)

**let-macro**

Implement **let-macro**, a macro that acts just like the let special form. Recall that let takes in a list of bindings and a body expression. It creates a temporary frame containing the given bindings, and returns the result of evaluating the body in this temporary frame. Do not use the let special form in your solution. You may use the provided cadr procedure in your solution.

Hint: The built-in map procedure takes in a one-argument function and a list and

returns the result of mapping the function to every element in the list.

;; **scm>** (define x 3)

;; x

;; **scm>** (let-macro ((x 1) (y 2)) (+ x y))

;; 3

;; **scm>** (let-macro ((x 2) (y x)) (\* x y))

;; 6

**(define-macro (let-macro bindings body)**

(cons \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_))

(define (cadr lst) (car (cdr lst)))

**lambda-macro (Spring 2018 Final)**

Implement **lambda-macro**, a macro that creates anonymous macros. A **lambda-macro** expression has a list of formal parameters and one body expression. It creates a macro with those formal parameters and that body. Assume that the symbol **anon** is not in use anywhere else in a program that contains **lambda-macro**.

;; A lambda-macro expression evaluates to a macro.

;; For example: ((lambda-macro (expr) (car expr)) (+ 1 2)) evaluates to the symbol +

**(define-macro (lambda-macro bindings body)**

‘(begin (\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_) anon)**)**

**infix (Fall 2018 Final)**

Implement **infix**, a Scheme macro that evaluates infix expressions. An infix expression is either a number or a three-element list containing an infix expression, a procedure, and another infix expression. The value of a compound infix expression is the value of its second element applied to the values of its first and third elements. You may use cadr and caddr to solve this problem.

;; A macro to evaluate infix expressions.

;; **scm>** (infix (2 \* 3))

;; 6

;; **scm>** (infix ((1 + 1) \* (1 + 2)))

;; 6

;; **scm>** (infix ((1 + (3 - 2)) \* ((2 + 3) + 2)))

;; 14

**(define-macro (infix e)**

(if (number? e) e

`(\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)))

(define (cadr x) (car (cdr x)))

(define (caddr x) (car (cdr (cdr x))))