MACROS, TAIL RECURSION AND INTERPRETERS

COMPUTER SCIENCE MENTORS CS 61A

April 9 to April 11, 2018

1 Let in Scheme

1. **let** is a special form in Scheme which allows you to create local bindings. Consider the example

   ```
   (let ((x 1)) (+ x 1))
   ```

   Here, we assign `x` to `1`, and then evaluate the expression `(+ x 1)` using that binding, returning `2`. However, outside of this expression, `x` would not be bound to anything.
   Each `let` special form has a corresponding lambda equivalent. The equivalent lambda expression for the above example is

   ```
   ((lambda (x) (+ x 1)) 1)
   ```

   The following line of code does not work. Why? Write the lambda equivalent of the `let` expressions.

   ```
   (let ((foo 3)
         (bar (+ foo 2)))
       (+ foo bar))
   ```

2 Macros

1. What will Scheme output?

```
scm> (define x 6)

scm> (define y 1)

scm> '(x y a)

scm> `(,x ,y a)

scm> `(,x y a)

scm> `(,(if (- 1 2) '+ '-) 1 2)

scm> (eval `(,(if (- 1 2) '+ '-) 1 2))

scm> (define (add-expr a1 a2)
              (list '+ a1 a2))

scm> (add-expr 3 4)

scm> (eval (add-expr 3 4))

scm> (define-macro (add-macro a1 a2)
            (list '+ a1 a2))

scm> (add-macro 3 4)
```

2. Implement `if-macro`, which behaves similarly to the `if` special form in Scheme but has some additional properties. Here's how the `if-macro` is called:
   `if <cond1> <expr1> elif <cond2> <expr2> else <expr3>`
   If cond1 evaluates to a truth-y value, expr1 is evaluated and returned. Otherwise, if cond2 evaluates to a truth-y value, expr2 is evaluated and returned. If neither condition is true, expr3 is evaluted and returned.

```
;Doctests
scm> (if-macro (= 1 0) 1 elif (= 1 1) 2 else 3)
2
scm> (if-macro (= 1 1) 1 elif (= 2 2) 2 else 3)
1
scm> (if-macro (= 1 0) (/ 1 0) elif (= 2 0) (/ 1 0) else 3)
3

(define-macro (if-macro cond1 expr1 elif cond2 expr2 else
   expr3)

)
```

3. Could we have implemented `if-macro` using a function instead of a macro? Why or why not?

4. 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 operand twice.

```
;Doctests
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  ____________________________))

           (__________________________________________________)))
```

3 Tail Recursion

1. What is a tail context? What is a tail call? What is a tail recursive function?

2. Why are tail calls useful for recursive functions?

3. Consider the following function:

```
(define (count-instance lst x)
  (cond ((null? lst) 0)
        ((equal? (car lst) x) (+ 1 (count-instance
                                         (cdr lst) x)))
        (else (count-instance (cdr lst) x))))
```

What is the purpose of `count-instance`? Is it tail recursive? Why or why not? Optional: draw out the environment diagram of this sum-list with `lst = (1 2 1)` and `x = 1`.

4. Rewrite count-instance to be tail recursive.

```
(define (count-tail lst x)

)
```

5. Implement `filter`, which takes in a one-argument function `f` and a list `lst`, and returns a new list containing only the elements in `lst` for which `f` returns true. Your function must be tail recursive.
   You may wish to use the built-in append function, which takes in two lists and returns a new list containing the elements of the first list followed by the elements of the second.

```
;Doctests
scm> (filter (lambda (x) (> x 2)) '(1 2 3 4 5))
(3 4 5)

(define (filter f lst)

)
```

4 Interpreters

1. Circle the number of calls to `scheme_eval` and `scheme_apply` for the code below.

```
(+ 1 2)
```

`scheme_eval`	1	3	4	6
`scheme_apply`	1	2	3	4

2. Write the number of calls to `scheme_eval` and `scheme_apply` for the code below.

```
(if 1 (+ 2 3) (/ 1 0))
```

`scheme_eval`	1	3	4	6
`scheme_apply`	1	2	3	4

```
(or #f (and (+ 1 2) 'apple) (- 5 2))
```

`scheme_eval`	6	8	9	10
`scheme_apply`	1	2	3	4

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

(+ (square 3) (- 3 2))
```

`scheme_eval`	2	5	14	24
`scheme_apply`	1	2	3	4

```
(define (add x y) (+ x y))

(add (- 5 3) (or 0 2))
```