INTERPRETERS AND MACROS

CS 61A GROUP MENTORING

July 30, 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 Interpreters

The following questions refer to the Scheme interpreter. Assume we're using the implementation seen in lecture and in the Scheme project, as well as the Calculator subset seen in discussion.

2. What's the purpose of the read stage in a Read-Eval-Print Loop? For our Scheme interpreter, what does it take in, and what does it return?

3. What are the two components of the read stage? What do they do?

4. Write out the constructor for the Pair object the read stage creates with the input string `(define (foo x) (+ x 1))`

5. For the previous example, imagine we saved that Pair object to the variable `p`. How could we check that the expression is a `define` special form? How would we access the name of the function and the body of the function?

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

```
scm> (+ 1 2)
3
```

```
scm> (if 1 (+ 2 3) (/ 1 0))
5
```

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

```
scm> (define (square x) (* x x))
square
scm> (+ (square 3) (- 3 2))
10
```

```
scm> (define (add x y) (+ x y))
add
scm> (add (- 5 3) (or 0 2))
2
```

3 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  ____________________________))

           (_____________________________________________________)))
```