Assignment 2

Question 1 (18 points)

1.1 - T (3 points) What is a special form?

Special form is a form with a special operator in the beginning of the expression.

1.2 - T (3 points) Are all atomic expressions primitive? if your answer is 'no', provide an example.

No, not all atomic expressions are primitive, for example 'y':

>(define y 1)

1.3 - T (3 points) In what cases multiple expressions in the body of a procedure expression (lambda form) are useful?

Having several expressions in a body is useful only when their evaluations have side effects.

1.4 - T (3 points) What is a syntactic abbreviation?

Syntactic abbreviation is a special operator that does not need a special evaluation rule, since it is an alias of another special operator.

1.5 - T (3 points) Recall that the *let* construct in Scheme is implemented as a syntactic abbreviation. Write the syntactic form the following expression is translated to (you do not have to compute the value of the expression, only show the resulting translated expression):

```
(let ((addsquares (lambda (x y) (+ (* x x) (* y y))))
      (add5 (lambda (x) (+ x 5)))
      (x 120))
```

```
((lambda (addsquares add5 x)
        (add5 (addsquares (add5 x) (add5 240))))
        (lambda (x y) (+ (*x x) (* y y)))
        (lambda (x) (+ x 5))
        120)
```

1.6 - T (3 points) Consider the following conversion of `or` expressions to nested `if` expressions:

```
(or arg1 arg2 ... argN) ->
    (if arg1
        arg1
        (if arg2
            arg2
            ...
            (if argN
                argN
                #f)))
```

Recall our definition of *functional equivalence*. The concept as we defined it does not include the concept of side-effects.

1.6.1 - T Are the two expressions *functionally equivalent* according to our definition in class? Prove your answer.

Yes, they both answer true if one of the args return true, by the order of the args.

1.6.2 - T Are the two expressions *functionally equivalent* when considering side-effects as well? Prove your answer.

No, - 'or' evaluates all the arguments, and 'if' evaluates only the arguments that it reaches. If there was a side effect in an argument that if did not reach, 'if' wouldn't do the side effect while 'or' would.

1.6.3 - T Do `or` expressions in `Racket` support *shortcut semantics*? Prove your answer.

Yes, example -

```
(or (let((num 5))
     (eq? num 5))
   (eq? 1 2)
   #f)
```

1.6.4 - T Does the translated `if` expression above support *shortcut semantics*? Prove your answer.

Yes, example –

```
(if (let((num 5))
     (eq? num 3))
   1
   2)
```

Question 2 (9 points)

2.1

```
(define length
  (lambda (l)
    (if (empty? l)
        0
        (+ 1 (length (cdr l))))))
```

evaluate((define length (lambda (l) (if (empty? l) 0 (+ 1 (length (cdr l)))))) [compound special form]
    evaluate (lambda (l) (if (empty? l) 0 (+ 1 (length (cdr l)))))[compound special form]
    return value: <Closure (l) (if (empty? l) 0 (+ 1 (length (cdr l))))>
    add the binding <<length>, <Closure (l) (if (empty? l) 0 (+ 1 (length (cdr l))))>> to the GE
return value: void

2.2

```
(define length
  (lambda (l)
    (if (empty? l)
        0
        (+ 1 (length (cdr l))))))
(define mylist (cons 1 (cons 2 '())))
(length mylist)
```

evaluate ((define length (lambda (l) (if (empty? l) 0 (+ 1 (length (cdr l)))))) [compound special form]
    evaluate (lambda (l) (if (empty? l) 0 (+ 1 (length (cdr l)))))[compound special form]
    return value: <Closure (l) (if (empty? l) 0 (+ 1 (length (cdr l))))>
    add the binding <<length>, <Closure (l) (if (empty? l) 0 (+ 1 (length (cdr l))))>> to the GE
return value: void
evaluate (define mylist (cons 1 (cons 2 '()))) [compound special form]
    evaluate (cons 1 (cons 2 '())) [compound special form]
        evaluate (cons) [atomic]
        return value: #<procedure:>>
        evaluate (1) [atomic]
        return value: 1
        evaluate(cons 2 '()) [compound special form]
            evaluate (cons) [atomic]
            return value: #<procedure:>>
            evaluate (2) [atomic]
            return value: 2
            evaluate ('()) [atomic]
            return value: '()
        return value: '(2)
    return value: '(1 2)
    add the binding <<mylist>, '(1 2)>> to the GE
return value: void
evaluate (length mylist) [compound non-special form]
    evaluate (length) [atomic]
    return value: <Closure (l) (if (empty? l) 0 (+ 1 (length (cdr l))))> (GE)

```
evaluate (mylist) [compound]
return value: '(1 2) (GE)
replace (l) with '(1 2): (if (empty? '(1 2)) 0 (+ 1 (length (cdr '(1 2)))))
evaluate (if (empty? '(1 2)) 0 (+ 1 (length (cdr '(1 2))))) [compound special form]
        evaluate (empty? '(1 2)) [compound non-special form]
                evaluate (empty?) [atomic]
                return value: #<procedure:>>
                evaluate ('(1 2)) [compound]
                return value: '(1 2)
        return value: #f
        evaluate  (+ 1 (length (cdr '(1 2)))) [compound non-special form]
                evaluate (+) [atomic]
                return value: #<procedure:>>
                evaluate (1) [atomic]
                return value: 1
                evaluate (length (cdr '(1 2))) [compound non-special form]
                        evaluate (length) [atomic]
                        return value: <Closure (l) (if (empty? l) 0 (+ 1 (length (cdr l))))> (GE)
                        evaluate (cdr '(1 2)) [compound non-special form]
                                evaluate (cdr) [atomic]
                                return value: #<procedure:>>
                                evaluate ('(1 2)) [compound]
                                return value: '(1 2)
                        return value: '(2)
                        replace (l) with '(2): (if (empty? '(2)) 0 (+ 1 (length (cdr '(2)))))
                        evaluate (if (empty? '(2)) 0 (+ 1 (length (cdr '(2))))) [compound special
form]
                                evaluate (empty? '(2)) [compound non-special form]
                                        evaluate (empty?) [atomic]
                                        return value: #<procedure:>>
                                        evaluate ('(2)) [compound]
                                        return value: '(2)
                                return value: #f
                                evaluate  (+ 1 (length (cdr '(2)))) [compound non-special form]
                                        evaluate (+) [atomic]
                                        return value: #<procedure:>>
                                        evaluate (1) [atomic]
                                        return value: 1
                                        evaluate (length (cdr '(2))) [compound non-special form]
                                                evaluate (length) [atomic]
                                                return value: <Closure (l) (if (empty? l) 0 (+ 1
(length (cdr l))))> (GE)
                                                evaluate (cdr '(2)) [compound non-special form]
                                                        evaluate (cdr) [atomic]
                                                        return value: #<procedure:>>
                                                        evaluate ('(2)) [compound]
                                                        return value: '(2)
                                                return value: '()
```

replace (l) with '(): (if (empty? '()) 0 (+ 1 (length (cdr '()))))

evaluate (if (empty? '()) 0 (+ 1 (length (cdr '()))))

[compound special form]

evaluate (empty? '()) [compound non-special form]

evaluate (empty?) [atomic]
return value: #<procedure:>>
evaluate ('()) [atomic]
return value: '()
return value: #t
evaluate (0) [atomic]
return value: 0
return value: 0
return value: 0
return value: 1
return value: 1
return value: 1
return value: 2
return value: 2
return value: 2

2.3

```
(define x 3)
(define y 0)
(+ x y y)
(lambda (x) (+ x y y))
(lambda (y) (lambda (x) (+ x y y)))
((lambda (x) (+ x y y)) 5 )
((lambda (y) (lambda (x) (+ x y y))) 2)
(((lambda (y) (lambda (x) (+ x y y))) 2) 5)
```

evaluate((define x 3)) [compound special form]
evaluate(3) [atomic]
return value: 3
add the binding <<x>,3> to the GE
return value: void
evaluate((define y 0)) [compound special form]
evaluate(0) [atomic]
return value: 0
add the binding <<y>,0> to the GE
return value: void
evaluate (+ x y y) [compound non-special form]
evaluate (+) [atomic]
return value: #<procedure:>>
evaluate (x) [atomic]
return value: 3 (GE)
evaluate (y) [atomic]
return value: 0 (GE)
evaluate (y) [atomic]
return value: 0 (GE)
return value: 3
evaluate (lambda (x) (+ x y y)) [compound special form]

```
return value: <Closure (x) (+ x y y)>
evaluate (lambda (y) (lambda (x) (+ x y y))) [compound special form]
return value: <Closure (y) (lambda (x) (+ x y y))>
evaluate ((lambda (x) (+ x y y)) 5 ) [compound non-special form]
        evaluate (lambda (x) (+ x y y)) [compound special form]
        return value: <Closure (x) (+ x y y)>
        evaluate (5) [atomic]
        return value: 5
        replace (x) with (5): (+ 5 y y)
        evaluate (+ 5 y y) [compound non-special form]
                evaluate (+) [atomic]
                return value: #<procedure:>>
                evaluate (5) [atomic]
                return value: 5
                evaluate (y) [atomic]
                return value: 0 (GE)
                evaluate (y) [atomic]
                return value: 0 (GE)
        return value: 5
return value: 5
evaluate ((lambda (y) (lambda (x) (+ x y y))) 2) [compound non-special form]
        evaluate (lambda (y) (lambda (x) (+ x y y))) [compound special form]
        return value: <Closure (y) (lambda (x) (+ x y y)))
        evaluate (2) [atomic]
        return value: 2
        replace (y) with (2): (lambda (x) (+ x 2 2))
        evaluate (lambda (x) (+ x 2 2))
        return value: <Closure (x) (+ x 2 2)>
return value: <Closure (x) (+ x 2 2)>
evaluate (((lambda (y) (lambda (x) (+ x y y))) 2) 5) [compound non-special form]
        evaluate ((lambda (y) (lambda (x) (+ x y y))) 2) [compound non-special form]
                evaluate (lambda (y) (lambda (x) (+ x y y))) [compound special form]
                return value: <Closure (y) (lambda (x) (+ x y y)))
                evaluate (2) [atomic]
                return value: 2
                replace (y) with (2): (lambda (x) (+ x 2 2))
                evaluate (lambda (x) (+ x 2 2))
                return value: <Closure (x) (+ x 2 2)>
        return value: <Closure (x) (+ x 2 2)>
        evaluate (5) [atomic]
        return value: 5
        replace (x) with (5): ((+ 5 2 2))
        evaluate (+ 5 2 2)) [compound non-special form]
                evaluate (+) [atomic]
                return value: #<procedure:>>
                evaluate (5) [atomic]
                return value: 5
                evaluate (2) [atomic]
                return value: 2
                evaluate (2) [atomic]
                return value: 2
        return value: 9
return value: 9
```

Question 3 (6 points)

3.1

```
(define fib (lambda (n)                                          ;1
              (cond ((= n 0) 0)                                  ;2
                    ((= n 1) 1)                                  ;3
                    (else (+ (fib (- n 1)) (fib (- n 2)))))))    ;4
(define y 5)                                                     ;5
(fib (+ y y))                                                    ;6
```

Binding Instance	Appears first at line #	Scope	Line #s of bound occurrences
fib	1	Universal Scope	6
n	1	Lambda body (1)	2-4
y	5	Universal Scope	5

Free variable occurrences: =, -, +

3.2

```
(define triple (lambda (x)                          ;1
                 (lambda (y)                        ;2
                   (lambda (z) (+ x y z)))))        ;3
(((triple 5) 6) 7)                                  ;4
```

Binding Instance	Appears first at line #	Scope	Line #s of bound occurrences
triple	1	Universal Scope	4
x	1	Lambda body (1)	none
y	2	Lambda body (2)	none
z	3	Lambda body (3)	3

Free variable occurrences: +, x, y