Assignment 2

Question 1 (18 points)

1.1 - T (3 points) What is a special form?
Answer: A special form is an expressions which has a special evaluation rule.
For example: Conditionals like if and cond, definitions (define), bindings like let, lambdas, etc.

1.2 - T (3 points) Are all atomic expressions primitive? if your answer is 'no', provide an example.
Answer: No. Example: (define x '(1 2 3)), now x is an atomic, but x is not primitive.
Explanation: atomic is all that is not a pair and not null. Therefore x is an atomic. On the other hand x is not primitive because it's evaluation process is not built into the language tools.

1.3 - T (3 points) In what cases multiple expressions in the body of a procedure expression (lambda form) are useful?
Answer: If we want the lambda to have side effects other then just returning the return value.
For example: displaying symbols to screen in order to test your code, changing values of variables, throwing errors, etc.

1.4 - T (3 points) What is a syntactic abbreviation?
Answer: A way to infer a certain expression with another expression which is semantically equivalent. For example: `(let (l (list 1 2 3)) <body that uses l>)` will allow us to infer the value '(1 2 3) with the symbol l in the body of the let. This is achieved by evaluating the let expression into lambda which receives the list '(1 2 3).

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))
(add5 (addsquares (add5 x) (add5 240))))
```

Answer:
```
( ( 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.
Answer: Yes.
They return the same values for the same inputs, and enters infinite loop for the same input.

Proof:
Lets consider only argi to be true, and the rest of the args to be false.
The or will return the first value that is not false, therefore argi will be return.
The if will return the first argi that satisfies the condition, so argi will be returned as well.
At the other case: all args are false. The or will return false, and the nested if's will be performed till the end, which #f is returned.
These two procedures can enter infinite loop.
Let argi a value that causes infinite loop. The or will return the first value which is not false, so by evaluating argi, infinite loop will be caused.
The if will check the condition "argi", so infinite loop will be caused as well.

1.6.2 - T Are the two expressions functionally equivalent when considering side-effects as well? Prove your answer.
Answer: No. Proof: If our argument has side effect, for example arg=display 'X. The or will evaluate our arg value once (As seen at line .1 provided below), and the if will evaluate it twice: on the 'condition' and on the 'then'.(As seen at line .2 below).
Code:
1. (or #f (display 'X))
2. (if #f #f (if (display 'X) (display 'X) #f))

Recall our definition of shortcut semantics.
1.6.3 - T Do or expressions in Racket support shortcut semantics? Prove your answer.
Answer: Yes. The or will return the first argument which is not false, and will not evaluate the rest. For example: (or #f 5 (display 'X)) will return 5, and 'X will not be displayed, because the evaluating of (display 'X) will not be performed.

1.6.4 - T Does the translated if expression above support shortcut semantics? Prove your answer.
Answer: Yes. The if will return the first argument which satisfies the condition, and will not evaluate the rest. For example: (if #t #t (if 5 5 (display 'X))). #t will be returned and 'X will not be displayed.

Question 2 (9 points)

This question refers to the language L3 as seen in the lectures. Show the steps of evaluating the following expressions according to the evaluation rules for L3 as seen in class. At each step, describe what expression is being evaluated, what type of evaluation is it (atomic/compound special form/compound non-special form), and what is the resulting value. If the return value is received from the global environment, state that fact next to the return value. You are provided with an example and its' solution in the next page.

2.1 - T (3 points)

Answer:

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 - T (3 points)

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 non-special form]
        evaluate (cons) [atomic]
        return value: #<procedure:cons>
        evaluate(1) [atomic]
        return value: 1
        evaluate (cons 2 '()) [compound non-special form]
            evaluate (cons) [atomic]
            return value: #<procedure:cons>
            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) [atomic]
        return value: '(1 2) (GE)
        replace (l) with (mylist): (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]
            …

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:cdr>
evaluate(‘(1 2)) [compound literal expression]
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]
…
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: 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:cdr>
evaluate (‘(2)) [compound literal expression]
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]
…
return value: #t
evaluate (0) [atomic]
return value: 0
return value: 0
return value: 0 ;(for this eval: evaluate (length (cdr '(2))))
return value: 1 ;(for this eval: evaluate (+ 1 (length (cdr ‘(2)))))
return value: 1 ;(for this eval: evaluate ((if (empty? ‘(2)) 0 (+ 1 (length (cdr ‘(2)))))))
return value: 1 ;(for this eval: evaluate (length (cdr '(1 2))))
return value: 2 ;(for this eval: evaluate (+ 1 (length (cdr '(1 2)))))
return value: 2 ;(for this eval: evaluate ((if (empty? '(1 2)) 0 (+ 1 (length (cdr '(1 2)))))))
return value: 2 ;(for this eval: evaluate (length mylist))

2.3 - T (3 points)
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))
        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))
        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 - T (3 points)

```
(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	4,6
n	1	Lambda body (1)	2,3,4
y	5	Universal Scope	6

Free variable occurrences: =, +, -.

3.2 - T (3 points)

```
(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)	3
y	2	Lambda body (2)	3
z	3	Lambda body (3)	3

Free variable occurrences: +.