PPL-assignment 2

1. Forms whose first expression is special operator. Special forms are evaluated by the special evaluation rules of their special operator.

An example for special form: lambda, define expressions.

2 .No, primitive expression define by scheme,

The variable x is not defined by scheme, the programmer may define it .so it is not primitive

and it is also not compound because it not composed by multiple expressions so it is an atomic expression.

3.Although that if we have more than one expression in the same function the interpreter return only the last expression there are Uses for some expressions inside the body of the function if we want to compute an expression the have side effect , for example

```
(define average (lambda (x y)
(display x)
 (newline)
(display y)
 (newline)
(/ (+ x y) 2)))
```

Display and newline expressions have side effect of printing to screen.

4. A special syntax, introduced for the sake of convenience, as example the let abbreviation, quote,

For example: ‘() is abbreviation of (list)

‘(1 .2) is abbreviation of (cons 1 2)

5. The syntax form the expression is:

((lambda (addsquares add5 x)

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

(lambda (x y) (+ (* x x) (* y y)))

(lambda (x) (+ x 5))

120)

6.1 True, we prove that with induction on N – the number of the arguments.

If the evaluation of all args are #f than both function return false.

Else, exist argi that return #t and arg0-argi-1 return #f , there are 3 cases

1.1If argi evaluate to true the 'or' function return true, also the if will evaluate argi and return the evaluation of argi = true.

1.2 If argi evaluate to false 'or' function will evaluate the next argument, also

the if evaluate argi and continue to the next condition.

1. If argi evaluation does not terminate when the 'or' function try to evaluate argi it will not halt.
   also if try to evaluate argi but will not halt.
2. If argi evaluation throw an error the 'or' function throw an error when argi evaluate, also when the evaluate of argi , it will throw an error as well.

6.2
No, an example
(or #f #f (display 5) (display 6) (display 7) (display 8))

```
(if #f (display 4)
   (if (display 5) (display 5)
      (if (display 6) (display 6)
         (if (display 7)
            (display 7) (#f)))))
```

5
55
>

The if expression compute display 5 twice while the or expression do it only once

1.6.3 yes, the or expression support shortcut semantic, given i such that we get true in the expression of index i , we compute all the expressions before the index i and all the expression after index i won't compute, and if there is infinite loop after the index i we won't compute the expression that will cause the loop so the program will not go into loop

1.6.4 yes, the if expression support shortcut semantic, given i such that i is the first place that we get in the expression true, we compute the "then" expression and return the value of the compute of the expression
we compute all the expressions before the index i and all the expression after the index i wont compute , and if there infinite loop after the index i we won't compute the expression that will cause the loop so the program will not go into loop.

. And that the reason that "or" and the "if" expression support in shortcut semantic because the expressions after the first one that return true don't compute.

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 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<<my list> : ’(1 2)> to the GE

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)

Replace (l) with(‘(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 : empty?>

Evaluate (‘(1 2))[compound literal expression]

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)))

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))

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 : empty?>

Evaluate ('(2))[compound literal expression]

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)))

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 : cdr>

Evaluate('(2))[compound literal expression]

Return value :'(2)

Return value:'()

Replace (l) with (' ())

Evaluate ((if (empty? '( )) 0 (+ 1 (length (cdr '( )))))[compound special form]

Evaluate (empty? '(2))[compound non special form]

Evaluate (empty?) [atomic]

Return value: #<procedure : empty?>

Evaluate ('())[atomic]

Return value: '()

Return value:#t

Evaluate (0)[atomic]

Return value: 0

Return value: 1

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 x 0))[compound special form]

Evaluate (0)[atomic]

return value :0

add the binding <<x>, 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 x y))>

Evaluate((lambda (y) (lambda (x) (+ x y y))))

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)

Evaluate ((+ 5 y y))

Evaluate (+)[atomic]

Return value(+):#<procedure:+>

Evaluate (5)[atomic]

Return value:5

Evaluate (y)[atomic]

Return value:(0) (GE)

Replace (y) with (0)

Evaluate (y)[atomic]

Return value:(0) (GE)

Replace (y) with (0)

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)

Evaluate((lambda (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)))))[compound special form]

Return value:<Closure(y) (lambda (x) (+ x y y)))>

Evaluate (2)[atomic]

Return value:2

Replace (y) with 2

Evaluate((lambda (x) (+ x 2 2))5)))[compound non special form]

Evaluate(((lambda (x) (+ x 2 2)))[compound special form]

Return value<Closure (x) (+ x 2 2)>

Evaluate (5) [atomic]

Return value: 5

Replace (x) with 5

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

Return value :9

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	4, 6
n	1	Lambda body(1)	2-4
y	5	Universal scope	6

Free variable occurrence : =, +, -

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)	3
Y	2	Lambda body(2)	3
Z	3	Lambda body(3)	3

Free variable occurrence:+ ,