PPL – Assignment 2

Sagiv Nethanel 203308069

Roy Soldin 204542179

Question 1 - General Terminology

1.1 Special form is a form of expression which a special evaluation rule exists in the language semantics, explaining how to compute them. In special forms not all sub-expressions are evaluated. (define, lambda, if, cond and else are the special operators).

1.2 Atomic expression is an expression which it's semantics is built in the language and cannot be dismantled.

1.3 Compound expression is a syntactic form used to combine expressions into complex expressions.

1.4 Primitive expression is expression whose evaluation is built in in the interpreter and which is not explained by the semantics of the language. These include primitive operations (for example, arithmetic operations on number or comparison operators) and primitive literal values (for examples, numbers or boolean values).

1.5

1.5.1 + is a primitive operator which is a primitive atomic expression.

1.5.2 5 is a literal expression which is a primitive atomic expression.

1.5.3 x is a variable which is an atomic expression.

1.5.4 ((lambda (x) x) 5) is a procedure expression and its value is a closure, which is a special compound expression.

1.6 multiple expressions in the body of a procedure expression (lambda form) is useful mainly when those expressions have a side effect.

1.7 we call an expression a" syntactic abbreviation" of another expression when it replaced by the real syntax during pre-processing. it is a special operator that does not need a special evaluation rule, since it is an alias of another special operator.

1.8 the expression that the syntactic abbreviation:

```
(let ((x (lambda (x) (+ x 1))) (y ((lambda (y) (- y 22)) 23)) (z 6)) (* (x z) y))
```

translates to:

```
( (lambda (x y z)
   (* (x 6) 1))
     (lambda (x) (+ x 1))
     1
     6)
```

1.9 and expressions in Racket do not support shortcut semantics. shortcut semantics are possible only if there is an element that satisfies the predicate (or not in our case), and the result is known without the need to go throw all other elements.
In case of and expression, shortcut semantics can be used, for example:
Predicate = (x>0) ? true : false
Array = [1,-2,3,4,-5]
so after observing Array[1], the answer is known already without the need to go throw all the rest of the Array.
However, in Racket we use reduce function for this purpose, and reduce does not support shortcut semantics, and goes throw all elements in order to evaluate their value.

1.10

1.10.1 foo and goo are functionally equivalent according to class definition because whenever foo(x) is evaluated to a value, goo(x) is evaluated to the same value, if foo(x) throws an exception, so does goo(x), and if foo(x) does not terminate, so does g(x). the only difference between them is that goo(x) has side effect which does not affect equivalent.

1.10.2 foo(x) and goo(x) are not functionally equivalent according to the new definition which considering side-effects, because goo(x) contains the procedure "display", which has side-effect.

Question 2 - Rules of Evaluation

2.1

(define x 12)

((lambda (x) (+ x (+ (/ x 2) x))) x)

evaluate((define x 12) [compound special form]
evaluate(12) [Atomic]
add the binding < x,12> to the GE
return value: void
evaluate ((lambda (x) (+ x (+ (/ x 2) x))) x)) [compound non-special form]
evaluate((lambda (x) (+ x (+ (/ x 2) x))) [compound special form]
return value: <closure (x) (+ x (+ (/ x 2) x))>
evaluate(x) [Atomic]
return value: 12 (GE)
replace (x) with (12): (+ x (+ (/ x 2) x))
evaluate(+ 12 (+ (/ 12 2) 12))
return value: 30

2.2

```
(define last
        (lambda (l)
                (if (empty? (cdr l))
                        (car l)
                        (last (cdr l)))))
```

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

2.3

```
(define last
         (lambda (l)
                 (if (empty? (cdr l))
                         (car l)
                         (last (cdr l)))))
(last '(1 2))
```

```
evaluate((define last (lambda (l) (if (empty? (cdr l)) (car l) (last (cdr l))))))          [compound special form]
        return value: <Closure (l) (if (empty? (cdr l)) (car l) (last (cdr l))))>
        add the binding < last ,<Closure (l) (if (empty? (cdr l)) (car l) (last (cdr l))))>> to the GE
evaluate(last '(1 2))                                                                      [compound non-special form]
        evaluate(last)                                                                     [Atomic]
        return value: <Closure (l) (if (empty? (cdr l)) (car l) (last (cdr l))))>
        evaluate('(1 2))                                                                   [compound non-special form]
        return value: '(1 2)
        replace (l) with '(1 2): (if (empty? (cdr '(1 2))) (car '(1 2)) (last (cdr '(1 2)))))
        evaluate (if (empty? (cdr '(1 2))) (car '(1 2)) (last (cdr '(1 2)))))              [compound special form]
                evaluate(empty? (cdr '(1 2)))                                              [compound non-special form]
                        evaluate(empty?)                                                   [Atomic]
                        return value: #<procedure:>
                        evaluate(cdr '(1 2))                                               [compound non-special form]
                                evaluate(cdr)                                              [Atomic]
                                return value: #<procedure:>
                                evaluate('(1 2))                                           [compound non-special form]
                                return value: '(1 2)
                        return value: '(2)
                return value: #f
                evaluate (last (cdr '(2)))                                          [compound non-special form]
                        evaluate(last)                                                     [Atomic]
                        return value: <Closure (l) (if (empty? (cdr l)) (car l) (last (cdr l))))>
                        evaluate(cdr '(2))                                                 [compound non-special form]
                                evaluate(cdr)                                              [Atomic]
                                return value: #<procedure:>
                                evaluate('(2))                                             [compound non-special form]
                                return value: '(2)
                        return value: '()
                        replace (l) with '(2): (if (empty? (cdr '(2))) (car '(2)) (last (cdr '(2)))))
                        evaluate (if (empty? (cdr '(2))) (car '(2)) (last (cdr '(2)))))    [compound special form]
                                evaluate(empty? (cdr '(2)))                                [compound non-special form]
                                        evaluate(empty?)                                   [Atomic]
                                        return value: #<procedure:>
                                        evaluate(cdr '(2))                                 [compound non-special form]
                                                evaluate(cdr)                              [Atomic]
                                                return value: #<procedure:>
                                                evaluate('(2))                             [compound non-special form]
                                                return value: '(2)
                                        return value: '()
                                return value: #t
                                evaluate((car '(2)))                                       [compound non-special form]
                                        evaluate(car)                                      [Atomic]
```

return value: #<procedure:>

evaluate('(2)) [compound non-special form]

return value: '(2)

return value: 2

Question 3 – Scopes

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

Question 5

```
;; Scheme Parser
;;
;; L2 extends L1 with support for IfExp and ProcExp
;; L3 extends L2 with support for:
;; - Pair and List datatypes
;; - Compound literal expressions denoted with quote
;; - Primitives: cons, car, cdr, list?
;; - The empty-list literal expression
;; - The Let abbreviation is also supported.

;; <program> ::= (L3 <exp>+) // Program(exps:List(Exp))
;; <exp> ::= <define> | <cexp>              / DefExp | CExp
;; <define> ::= ( define <var> <cexp> )     / DefExp(var:VarDecl, val:CExp)
;; <var> ::= <identifier>                   / VarRef(var:string)
;; <cexp> ::= <number>                      / NumExp(val:number)
;;         |  <boolean>                     / BoolExp(val:boolean)
;;         |  <string>                      / StrExp(val:string)
;;         |  ( lambda ( <var>* ) <cexp>+ ) / ProcExp(params:VarDecl[], body:CExp[]))
;;         |  ( if <cexp> <cexp> <cexp> )   / IfExp(test: CExp, then: CExp, alt: CExp)
;;         |  ( let ( binding* ) <cexp>+ )  / LetExp(bindings:Binding[], body:CExp[]))
        |  ( let* ( binding* ) <cexp>+ )  / LetStarExp(bindings:Binding[], body:CExp[]))
;;         |  ( quote <sexp> )              / LitExp(val:SExp)
;;         |  ( <cexp> <cexp>* )            / AppExp(operator:CExp, operands:CExp[]))
;; <binding>  ::= ( <var> <cexp> )           / Binding(var:VarDecl, val:Cexp)
;; <prim-op>  ::= + | - | * | / | < | > | = | not |  eq? | string=?
;;                  | cons | car | cdr | list? | number?
;;                  | boolean? | symbol? | string?      ##### L3
;; <num-exp>  ::= a number token
;; <bool-exp> ::= #t | #f
;; <var-ref>  ::= an identifier token
;; <var-decl> ::= an identifier token
;; <sexp>     ::= symbol | number | bool | string | ( <sexp>* )             ##### L3
*/
```