01/05/18

Liav Rabi, id: 205688328
Benny Peresetsky, id: 310727771

PPL – Assignment 2

Question 1

1.1 A special form is an expression that follows special evaluations such as lambda expression, etc.
1.2 Atomic expression is a statement that has a single effect such as variable assignment or print statements. For example, the number expression "12" or the Boolean expression "true".
1.3 Compound expression is a statement that is build from multiple sub-statements such as if-then-else construct or for-loop construct.
1.4 Primitive expressions are expressions whose evaluation is built in in the interpreter and which are 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 atomic and primitive
1.5.2 atomic and primitive
1.5.3 primitive
1.5.4 compound

1.6 side-effects
1.7 semantically equivalent syntactic structure
1.8 ((lambda (x y z) (* (x z) y)
(lambda (x) (+ x 1))
((lambda (y) (- y 22)) 23))
(6))

1.9 Yes. Example: (and (> 1 2) (display "working")) => #f. Because "and" expression supports shortcut semantics and the return value of the evaluation of (> 1 2) is #f, (display "working") will not be performed. Otherwise it would have printed "working" on the screen.
1.10.1 Yes. According to the definition of functional equivalence: two procedures considered functional equivalent if they halt on the same inputs and return the same value.
1.10.2 No. According to the practical session display/printing to the screen it is a side-effect and function goo includes also an expression of display.

Question 2

2.1

Evaluate((define x 12)) [compound special form]
Evaluate(12) [atomic]
Return value: 12
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)

```
Reduce: (+ x (+ (/ x 2) x))
Evaluate((+ x (+ (/ x 2) x)) [compound non-special form]
        Evaluate(+) [atomic]
        Return value: #<procedure:+>
        Evaluate(x) [atomic]
        Return value: 12 (GE)
                Evaluate((+ (/ x 2) x)) [compound non-special form]
                        Evaluate(+) [atomic]
                        Return value: #<procedure:+>
                        Evaluate((/ x 2)) [compound non-special form]
                                Evaluate(/) [atomic]
                                Return value: #<procedure:/>
                                Evaluate(x) [atomic]
                                Return value: 12 (GE)
                                Evaluate(2) [atomic]
                                Return value: 2
                        Return value: 6
                        Evaluate(x) [atomic]
                        Return value: 12 (GE)
                Return value: 18
        Return value: 30
```

2.2

```
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 <last, <closure,(l),(if (empty? (cdr l)) (car l) (last (cdr l))))>> to the GE
Return value: void
```

2.3

```
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 <last, <closure,(l),(if (empty? (cdr l)) (car l) (last (cdr l))))>> to the GE
Return value: void
Evaluate((last '(1 2))) [compound special form]
        Evaluate(last) [atomic]
        Return value: <closure,(l),(if (empty? (cdr l)) (car l) (last (cdr l))))> (GE)
        Evaluate('(1 2)) [compound literal]
```

```
        Return value: '(1 2)
        Apply process: (<closure,(l),(if (empty? (cdr l)) (car l) (last (cdr l))))>  '(1 2))
        Substitute: ((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: empty?>
                        Evaluate(cdr '(1 2)) [compound non-special form]
                                Evaluate(cdr) [atomic]
                                Return value: #<procedure: cdr>
                                Evaluate('(1 2)) [compound literal]
                                Return value: '(1 2)
                        Return value: '(2)
                Return value: #f
                Evaluate(last (cdr '(1 2))) [compound non-special form]
                        Evaluate(last) [atomic]
                        Return value: <closure,(l),(if (empty? (cdr l)) (car l) (last (cdr l))))> (GE)
                        Evaluate (cdr '(1 2)) [compound non-special form]
                                Evaluate(cdr) [atomic]
                                Return value: #<procedure: cdr>
                                Evaluate('(1 2)) [compound literal]
                                Return value: '(1 2)
                        Return value: '(2)
                Substitute(last '(2))
                Evaluate(last) [atomic]
        Return value: <closure,(l),(if (empty? (cdr l)) (car l) (last (cdr l))))> (GE)
                Evaluate('(2)) [compound literal]
                Return value: '(2)
                Apply process: (<closure,(l),(if (empty? (cdr l)) (car l) (last (cdr l))))> '(2))
                Substitute: ((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: empty?>
                                Evaluate(cdr '(2)) [compound non-special form]
                                        Evaluate(cdr) [atomic]
                                        Return value: #<procedure: cdr>
                                        Evaluate('(2)) [compound literal]
                                        Return value: '(2)
                                Return value: '()
                        Return value: #t
                        Evaluate(car '(2)) [compound non-special form]
                                Evaluate(car) [atomic]
                                Return value: #<procedure: car>
                                Evaluate(2) [compound literal]
                                Return value: 2
                        Return value: 2
                Return value: 2
        Return value: 2
Return value: 2
Return value: 2
```

Question 3

3.1

Binding Instance	Appears first in 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 variables: None.

3.2

Binding Instance	Appears first in 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 variables: None.

Question 5

```
;; <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
*/
```