PPL Assignment 2

Shlomi Weitzman - 313444895
Oz Elhassid – 311326110

Q1.1

Primitive atomic expression – 1

Non-primitive atomic expression - size

Non-primitive compound expression –

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

average is a non-primitive compound expression

Primitive atomic value - (define size 6) - now on, 6 will be the primitive atomic value of size.

Non-primitive atomic value – (define x size) – now on, size will be the non-primitive atomic value of x.

Non-primitive compound value – (define y average) – now on, average will be the non-primitive compound value of y.

Q1.2

While non-special form expressions evaluated by a general evaluation rule.
For each special form - a special evaluation rule exists. For instance, the 'if' special form, where one of subexpressions is not evaluated, unlike the non-special form in which all expressions are evaluated.

Q1.3

A variable x occurs free in an expression E if and only if there is some use of x in E that is not bound by any declaration of x in E. For instance, the variable y is free in the following: ((lambda (x) x) y).

Q1.4

Symbolic-Expression (s-exp) is a notation defined inductively:

<S-exp> ::= <AtomicSexp> | <CompoundSexp>
<AtomicSexp> ::= <number> | <boolean> | <string> | <symbol>
<CompoundSexp> ::= '(' <S-exp>* ')

For instance, the number 1.

Q1.5

'let' expression is a syntactic abbreviation that can be replaced with a lambda.
For instance:
(let ((x 1) (y 2))
  (+ x y))
is equal to:
((lambda (x y) (+ x y))
  1 2)

'cond' expression is also a syntactic abbreviation. Which can be replaced with an 'if'.
For instance:
(cond ((> 1 2) "first") ((> 2 2) "second") ((> 3 2) "third") (else "otherwise"))
can be replaced with:
(if (> 1 2) "first" (if (> 2 2) "second" (if (> 3 2) "third" "otherwise")))

Q1.6

Every L1 program can be transformed into an L0 program by replacing each variable with the value bound to it.

Q1.7

This is not the case for L2. Consider the following code:

```
(define count
  (lambda (x)
    (if (= x 0) 0 (+ x (count (- x 1))))))
```

The function 'count' is recursive, and therefore cannot be replaced, with a specific value.

Q1.8

PrimOp advantage:
By implementing primitive operators as syntactic expressions we take some of the load off the parser.

Closure advantage:
By defining primitive operators as closures in the global environment, we are making it easier to add more primitive operators in the future, as well as removing some of the interpreter's work.

Q1.9

As long as the given procedure has no side effects, the order does not affect the map function.

As long as the given procedure is commutative, the order does not affect the reduce function.

Q2 contracts:

Q2.1
; Signature: empty? (lst)
; Type: [Any -> Boolean]
; Purpose: To determine whether a given expression is the empty list.
; Pre-conditions: none
; Tests: (empty? '()) ==> #t

Q2.2
; Signature: list? (lst)
; Type: [Any -> Boolean]
; Purpose: To determine whether a given expression is a list.
; Pre-conditions: none
; Tests: (list? '()) ==> #t

Q2.3
; Signature: equal-list? (lst1 lst2)
; Type: [Any * Any -> Boolean]
; Purpose: To determine whether the two given expressions are equal lists.
; Pre-conditions: none
; Tests: (equal-list? '() '()) ==> #t

Q2.4
; Signature: append (lst1 lst2)
; Type: [Any * Any -> List(Any)]
; Purpose: if both arguments are lists, returns the second list appended to the first
; Pre-conditions: none
; Tests: (append '(1 2 3) '(4 5 6)) ==> '(1 2 3 4 5 6)

Q2.5
; Signature: append3 (lst1 lst2 num)
; Type: [Any * Any * Any -> List(Any)]
; Purpose: if both the first arguments are lists, returns the third argument appended to the second list appended to the first
; Pre-conditions: none
; Tests: (append3 '(1 2 3) '(4 5 6) 7) ==> '(1 2 3 4 5 6 7)

Q2.6
; Signature: pascal (num)
; Type: [Number -> List(Number)]
; Purpose: returns the num'th row of Pascal's triangle
; Pre-conditions: num is an integer
; Tests: (pascal 4) ==> '(1 3 3 1)