PPL – Assignment 3

Sagiv Nethanel 203308069

Roy Soldin 204542179

Part 1 – Concept Questions

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. furthermore, every special form eval function uses the current environment. (define, lambda, if, cond and else are the special operators). In case of primitive operator, in order to evaluate the expression, all sub-expressions have to be evaluated first and the evaluation doesn't use the environment.

isPrimOp(exp) ? exp :

isVarRef(exp) ? applyEnv(env, exp.var) :

2.

A)

```
eval(<OR-exp exp>,env) =>
       isEmpty(exp.args)? false:
       first(exp.args)? true:
       eval(makeORexp(rest(args)))
```

B)

```
Import * as R from 'ramda'
eval(<OR-exp exp>,env) =>
       {R.reduce((acc,curr)=>(acc | curr), false, exp.args)}
```

3. Maintaining the language with primitive operators represented by VarRef is easier, because modification of the language only consist an addition of new binding to the GE and implementation of the Primproc interface. No modification of applyPrimitive is needed.

4. The reasons that would justify switching from applicative order to normal order evaluation are:

- To avoid unnecessary calculations.
  Example:

```
(define test
  (lambda (x y)
    (if (= x 0)
        0
```

```
            y)))

    (define zero-div
      (lambda (n)
        (/ n 0))) ; division by zero!

    (test 0 (zero-div 5))


normal-eval[(test 0 (zero-div 5))]
    normal-eval[test] ==> <closure (lambda (x y) (if (= x 0) 0 y))>

(if (= x 0) 0 y))°{x = 0} ==> (if (= 0 0) 0 y))

(if (= 0 0) 0 y))°{y = (zero-div 5)} ==> (if (= 0 0) 0 (zero-div 5) ))

reduce:

normal-eval[(if (= 0 0) 0 (zero-div 5)))]
    normal-eval[(= 0 0)]
        normal-eval[ = ] ==>#<primitive-procedure =>
    normal-eval[ 0 ] ==>0
    normal-eval[ 0 ] ==>0
    ==> #t
    normal-eval[0] ==> 0
==>0
```

We didn't calculate (zero-div 5)

5. The reasons that would justify switching from normal order to applicative order evaluation are:

- To avoid recalculation of expressions.

  Example:

```
(define square (lambda (x) (* x x)))
(define sum-of-squares (lambda (x y) (+ (square x) (square y))))
(define f (lambda (a) (sum-of-squares (+ a 1) (* a 2)))
(f 5) ;; 136

normal-eval[ (f 5) ] ==>
    normal-eval[f] ==> <Closure (a) (sum-of-squares (+ a 1) (* a 2))>
    ==>
    normal-eval[ (sum-of-squares (+ 5 1) (* 5 2)) ] ==>
        normal-eval[ sum-of-squares ] ==> <Closure (x y) (+ (square x) (square y))>
    ==>
    normal-eval[ (+ (square (+ 5 1)) (square (* 5 2))) ] ==>
        normal-eval[ + ] ==> <prim-op +>
        normal-eval[ (square (+ 5 1)) ] ==>
            normal-eval[ square ] ==> <Closure (x) (* x x)>
            ==>
            normal-eval[ (* (+ 5 1) (+ 5 1)) ] ==>
            normal-eval[ * ] ==> <prim-op *>
            normal-eval[ (+ 5 1) ] ==>
                normal-eval[ + ] ==> <prim-op +>
                normal-eval[ 5 ] ==> 5
                normal-eval[ 1 ] ==> 1
            ==> 6
            normal-eval[ (+ 5 1) ] ==>
                normal-eval[ + ] ==> <prim-op +>
                normal-eval[ 5 ] ==> 5
                normal-eval[ 1 ] ==> 1
            ==> 6
        ==> 36
        normal-eval[ (square (* 5 2)) ] ==>
            normal-eval[ square ] ==> <Closure (x) (* x x)>
            ==>
            normal-eval[ (* (* 5 2) (* 5 2)) ] ==>
            normal-eval[ * ] ==> <prim-op *>
            normal-eval[ (* 5 2) ] ==>
                normal-eval[ * ] ==> <prim-op *>
                normal-eval[ 5 ] ==> 5
                normal-eval[ 2 ] ==> 2
            ==> 10
            normal-eval[ (* 5 2) ] ==>
                normal-eval[ * ] ==> <prim-op *>
                normal-eval[ 5 ] ==> 5
                normal-eval[ 2 ] ==> 2
            ==> 10
        ==> 100
    ==> 136
```

6. the reason for switching from the substitution model to the environment model is optimization: Instead of eagerly substituting variables by their values when we apply a closure, we leave the body of the closure untouched, and maintain an environment data structure on the side.

7.

Equivalent

```
0
```

normal order `(0)` => `0`

applicative order `(0)` => `0`

Non Equivalent

```
(define test
  (lambda (x y)
    (if (= )
        0
        y)))

(define zero-div
  (lambda (n)
    (/ n 0))) ; division by zero!

(test 0 (zero-div 5))
```

normal order `(test 0 (zero-div 5))` => `0`

applicative order `(test 0 (zero-div 5))` => `Error!`

8. The valueToLitExp procedure is needed only in the applicative order Interpreter because expressions are evaluated as soon as possible, and the ir value is needed to be added to the AST, but in form of an expression and not a value/ failing to do so would result in a parse tree syntax error. In the n ormal interpreter due to the lazy approach arguments are not evaluated unti l it must, hence no need to create an AST for them.

9. There is no need in valueToLitExp in the environment model. we are keeping a data structure (the environment) which keeps the value of the variables, and that's why when we are mapping a closure we don't need the AST for the evaluation process.

10. Changing a let to a closure depends on the model we are using. In the old models that we used (old languages) we were changing a let expression to a closure like in the function rewriteLet:

```
const rewriteLet = (e: LetExp) : AppExp => {
    const vars = map((x) => x.var, e.biniding);
    const vals = map((x) => x.val, e.binding);
    return makeAppExp (makeProcExp ( vars, e.body ), vals);
}
```

In the other hand, in the L4 model, we used the evalLetL4 which didn’t create a closure and saves the relevant data in the environment and delayed the building of the closure.