Assignment 2 PPL

Q1.1

It is not required, it can appear, but is not required. It is useful in all kinds of languages because it improves maintainability, it breaks down complex missions to smaller ones. It is also useful in L3 because of the same reasons.

Q1.2

a.

Because they can be so versatile, they can form a lot of different structures that can accomplish tasks that regular primitives cannot. Special forms can control what happens in them and what they do with their components(arguments), they can also determine scopes. For example, the ‘let’ expression creates a new scope, bind references names to their values and evaluate a value according to what we write inside of it’s body, inside of its scope. Another example is of course the ‘if’ expression that can actually change the flow of the program according to the conditional that is first evaluated inside of the condition parameter, something that some primitive expression can’t do.

b.

It must be defined as a special form, because it has its shortcut semantics functionality that needs to happen, which means that if it gets to a part in the whole condition expression that is true, then it stops and doesn’t make unnecessary evaluation of the rest of the condition expression because if it stumbles upon a true part in the whole condition expression, then the whole expression is gonna evaluate to true(in the case of the ‘or’ operator shortcut semantics). This functionality, like we wrote in q1.2.a, can be done only with a special form, that can actually control the flow of the program and can do what it is programmed to do with its arguments. Primitives are evaluated no matter what, which is not what we want here.

Q1.3

a.

The value of the program is 3, because in a regular ‘let’ expression, the binded pairs don’t ‘know’ each other because the initial values are computed before any of the variables become bound. So it returns ‘y’ which is (* x 3) and because the binded pairs inside the ‘let’ don’t ‘know’ each other the x is the x defined as 1 outside before the ‘let’ expression which means that ‘y’ is (* 1 3) = 3.

b.

The value of the given program is 15. The reason for that is because in the ‘let*’ expression the bindings are being done sequentially so the later bindings ‘know’ the bindings that came before them, because the ones that came before them are already visible to the later binded ones and are in their environment. So in this case, y is binded to (* x 3), and in this case, in (* x 3), x is the x binded to 5 in the binding that came before that, so (* x 3) here is (* 5 3) which means that y is 15. ‘y’ is returned so 15 is returned.

c.

The equivalent is as follows:

```
(let ((x 5))
      (let ((y (* x 3)))
y))
```

d.

The equivalent is as follows:

```
((lambda (x)
         ((lambda (y)
              y)
        (* x 3)))
5)
```

Q1.4

a.

The role of the function valueToLitExp is to convert values into literal expressions. This is important so that the interpreter can understand them(because he can understand them as literal expressions) and execute the evaluation.

b.

Because in the normal evaluation, the evaluation of the expressions is only done when needed, only when we need the value of the expression, we evaluate it. So the unevaluated expressions are just passed around according to their positions and we don't need to evaluate them that specific time so no need to turn them into literal expressions for evaluation that time.

c.

It is not needed in the environment model interpreter because here, in each scope, in each environment, we bind value names and their values in that specific environment. For that reason we just put the value there as it is, after it has been computed we just put the value there, the interpreter doesn't need to convert the value into a literal expression, it just uses the value as is.

Q1.5

a.

The reasoning is because normal order doesn't evaluate expressions at first sight, only when it reaches them. So there can be a lot of cases that it is more efficient because, this way it saves us computation time if we didn't really need to evaluate some expressions in order to get to the right result. For example: if we look at: (if true 1 ((1535884*531831)/5)), in applicative order we would evaluate everything, every expression in there so we would even need to evaluate what's inside the 'else' clause, which is complicated, even though it is not reachable and not relevant to us to get the right result(which is 1). In normal order we won't evaluate expressions until we really need them so we would have just gotten inside the 'if', see that the condition is true so we would just return 1, we just evaluate what we come across, which saves us a lot of computation time.

b.

The reasoning is because applicative order is already computing expression's values, which can save us a lot of time in some cases. For example, if there is an

expression that has multiple appearances inside another expression like, for example:
(lambda x ((x*x)*x*4))(sqrt(20000))
in normal order the sqrt(20000) would have been put in every appearance of the ‘x’ argument in the lambda expression and would have been computed again multiple times. On the other hand, if we use applicative order we could save a lot of computation time because in this order we already compute the value of sqrt(20000) only once, and put that value wherever the argument ‘x’ appears in the lambda expression.

Q1.6

a.

Because, in the environment model, when there are environments, a given reference name is known in that environment as the only reference name of that name, not looking outside of this environment, so even if there’s a reference name of the same name in another place, because it’s in another environment, it is not ‘known’ in the current environment, so renaming it for distinction between those references named the same is not needed because the other reference name is outside the current environment and is not visible in this environment. Only the one in the current environment is visible and ‘known’ by other in its environment.

b.

The Answer is, no, it’s not required. The reason for that is that the fact that it’s closed means that there are no free variables, so all the variables are bound in the current term/in the term they are in. So if they are bound in the current term and there are no free variables, it means that all the variables in the term are bound to the current term so there won’t be a problem with variables binding outside/around the term.

Q2.d

equivalent ProcExp:

```
(define pi 3.14)
(define square (lambda (x) * x x))
(define circle
        (lambda (x y radius)
                (lambda (msg)
                        (if (eq? msg 'area)
                                ((lambda () (* (square radius) pi)))
                                (if (eq? msg 'perimeter)
                                        ((lambda () (* 2 pi radius)))
                                        #f)))))
(define c (circle 0 0 3))
(c 'area)
```

Expressions passed as operands to the L3applicativeEval function:

1. 3.14
2. lambda (x) (* x x)
3. lambda (x y radius)
   ```
                   (lambda (msg)
                           (if (eq? msg 'area)
                          ((lambda () (* (square radius) pi)))
                           (if (eq? msg 'perimeter)
                                        ((lambda () (* 2 pi radius)))
                                        #f)))
   ```
4. (cricle 0 0 3)
5. circle
6. 0
7. 0
8. 3
9. lambda (msg_1) (if (eq? msg_1 ‘area) ((lambda () (* (square 3) pi))) (if (eq? msg_1 ‘perimeter) ((lambda () (* 2 pi 3)) #f)))
10. (c ‘area)
11. c
12. ‘area
13. if (eq? ‘area ‘area) ((lambda () (* (square 3) pi))) (if (eq? ‘area ‘perimeter) ((lambda () (* 2 pi 3)) #f)))
14. (eq? ‘area ‘area)
15. eq?
16. ‘area

17. ‘area
18. (* (square 3) pi)
19. lambda (* (square 3) pi)
20. (* (square 3) pi)
21. *
22. (square 3)
23. square
24. 3
25. *
26. 3
27. 3
28. pi

environment diagram shown in the next page.