**PPL – Assignment1 – Part1**

**By Yair Derry & Gal Noy**

**Question 1:**

* 1. Special forms are syntax constructs in programming languages that cannot be expressed as regular functions or primitive operators. For example, "if" statements for conditions and "let" statements for variable bindings in a local scope. Special forms enable more complex syntax and semantics in programming languages.
  2. All programs in L1 can be transformed into equivalent programs in L0 because L0 has all the primitive operators of L1. The purpose of the "define" special form in L1 is to bind values of expressions to variables names. Therefore, excluding “define” from the language, doesn’t affect the functionality & logic of it, so any program in L1 can be expressed in L0 without the need for “define”.
  3. Yes. For example, this function in L2:

(define factorial

(lambda (n)

(if = 1 n)

1

(\* n (n factorial(- n 1)))))

To invoke this function, we need a way to refer to the function within itself, which cannot be done without some means of creating named definitions (the special form “define”, which is excluded in L20).

* 1. Dsa
* Map – can be applied to each element of the list in parallel. This is because the result of applying the function to each element depends only on the element itself, and not on any other elements in the list.
* Reduce – The order of the procedure application on the list items should be

sequential, that is because the result of applying the operator to a list of elements depends on the order in which they are combined with the accumulator.

* Filter – can be applied to each element of the list in parallel. This is because the predicate is applied separately to each element of the list, depends only on the element itself, and not on any other elements in the list.
* All – this boolean function is applied sequentially to each element of the list in the order they appear in it. The function returns #f as soon as it encounters an element for which the function returns #f, so applying the function in parallel could potentially produce incorrect results.
* Compose – The order of procedure application cannot be parallel because the result of applying each procedure depends on the result of applying the previous procedure. Therefore, applying the procedures in parallel could potentially produce incorrect results.
  1. Tbd

<program> ::= (L31 <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(args:VarDecl[],/ body:CExp[]))

| ( if <cexp> <cexp> <cexp> ) / IfExp(test: CExp,then: CExp,alt: CExp)

| ( let ( <binding>\* ) <cexp>+ ) / LetExp(bindings:Binding[], body:CExp[]))

**| ( cond ( <cond-clauses>+ <else-clause> ) / CondExp(condClauses: Cond-clauses[], elseClause: ElseClause)**

| ( quote <sexp> ) / LitExp(val:SExp)

| ( <cexp> <cexp>\* ) / AppExp(operator:CExp, operands:CExp[]))

<binding> ::= ( <var> <cexp> ) / Binding(var:VarDecl,

val:Cexp)

**<clause> ::= <cond-clause> | <else-clause> / CondClauseExp | ElseClauseExp**

**<cond-clause> ::= ( <cexp> <cexp>+ ) / CondClauseExp(test: CExp, body: CExp[])**

**<else-clause> ::= ( <cexp>+ ) / ElseClauseExp(body: CExp[])**

<prim-op> ::= + | - | \* | / | < | > | = | not | eq? | string=?

| cons | car | cdr | list | pair? | list? | number?

| boolean? | symbol? | string?

<num-exp> ::= a number token

<bool-exp> ::= #t | #f

<str-exp> ::= "tokens\*"

<var-ref> ::= an identifier token

<var-decl> ::= an identifier token

<sexp> ::= symbol | number | bool | string | ( <sexp>\* )

**Contracts for Question 2 procedures**