PPL – Assignment1 – Part1

By Yair Derry & Gal Noy

Question 1:

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.

1.2. 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”.

1.3. 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).

1.4. Dsa

1.5.

- 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.

1.6. a lexical address as a tuple: [var : depth pos] where: var is the name of the variable, depth is the number of contours that are crossed to reach the variable declaration and pos is the offset of the variable within the declaration.

For example, let's look at this code:

```
(lambda (x y)
  ((lambda (x) (+ x y))
    (+ x x)) 1)
```

Adding lexical addresses will look like this:

```
(lambda (x y)
  ((lambda (x) (+ [x : 0 0] [y : 1 1]))
    (+ [x : 0 0] [x : 0 0])) 1)
```

1.7.

```
<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: CondClause
[], elseClause: ElseClause)
| ( quote <sexp> ) / LitExp(val:SExp)
| ( <cexp> <cexp>* ) / AppExp(operator:CExp, operands:CExp[]))
<binding> ::= ( <var> <cexp> ) / Binding(var:VarDecl,
val:Cexp)
<cond-clause> ::= ( <cexp> <cexp>+ ) / CondClause (test: CExp, body: CExp[])
<else-clause> ::= ( <cexp>+ ) / ElseClause(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

```
; Signature: take(lst pos)
; Type: [List(T) * Number -> List(T)]
; Purpose: Return a new list with the first 'pos' elements of the input list 'lst'
; Pre-conditions: 'lst' is a list and 'pos' is a non-negative integer
;                 'pos' is less than or equal to the length of 'lst'
; Tests: (take '(1 2 3 4 5) 3) => '(1 2 3)
;        (take '() 3) => '()
;        (take '(1 2 3) 0) => '()

; Signature: take-map(lst func pos)
; Type: [List(T) * (T -> T) * Number -> List(T)]
; Purpose: Return a new list obtained by applying 'func' to the first 'pos' elements of 'lst'
; Pre-conditions: 'lst' is a list, 'func' is a function that takes one argument and returns one value,
;                 and 'pos' is a non-negative integer
;                 'pos' is less than or equal to the length of 'lst'
```

```
; Tests: (take-map '(1 2 3 4 5) (lambda (x) (x+1)) 3) => '(2 3 4)
;        (take-map '() (lambda (x) (x+1)) 3) => '()
;        (take-map '(1 2 3) (lambda (x) (x+1)) 0) => '()


; Signature: take-filter(lst pred pos)
; Type: [List(T) * (T -> Boolean) * Number -> List(T)]
; Purpose: Return a new list obtained by filtering the first 'pos' elements of 'lst' using 'pred'
; Pre-conditions: 'lst' is a list, 'pred' is a function that takes one argument and returns a Boolean value,
;                 and 'pos' is a non-negative integer
;                 'pos' is less than or equal to the length of 'lst'
; Tests: (take-filter '(1 2 3 4 5) (lambda (x) (> x 1)) 3) => '(2 3 4)
;        (take-filter '() (lambda (x) (> x 1)) 3) => '()
;        (take-filter '(1 2 3) (lambda (x) (> x 1)) 0) => '()


; Signature: sub-size(lst size)
; Type: [List(T) * Number -> List(List(T))]
; Purpose: Return a list of sub-lists of length 'size' obtained by taking consecutive elements from 'lst'
; Pre-conditions: 'lst' is a list and 'size' is a positive integer
;                 'size' is less than or equal to the length of 'lst'
; Tests: (sub-size '(1 2 3 4 5) 3) => '((1 2 3) (2 3 4) (3 4 5))
;        (sub-size '() 3) => '()

; Signature: sub-size-map(lst, func, size)
; Type: [List(T) * (T -> T) * Number -> List(List(T))]
; Purpose: applies the function 'func' to each sub-list of 'lst' of size 'size'
;          and returns a list of the results.
; Pre-conditions: lst is a list, func is a function that takes a single argument,
;                 and size is a non-negative integer.
; Tests: (sub-size-map '(1 2 3) (lambda (x) (* x x)) 2) => '((1 4) (4 9))

; Signature: root(tree)
; Type: [List(T | List(T)) -> T | List(T)]
; Purpose: returns the value at the root of 'tree'
; Pre-conditions: tree is a list
; Tests: (root '(1 (2 #t 2) 2)) => 1

; Signature: left(tree)
; Type: [List(T | List(T)) -> T | List(T)]
; Purpose: returns the left subtree of 'tree'
; Pre-conditions: tree is a list
; Tests: (left '(1 (2 #t 2) 2)) => (2 #t 2)
```

```
; Signature: right(tree)
; Type: [List(T | List(T)) -> T | List(T)]
; Purpose: returns the right subtree of 'tree'
; Pre-conditions: tree is a list
; Tests: (right '(1 (2 #t 2) 2)) => 2

; Signature: count-node(tree, val)
; Type: [List(T | List(T)) * T -> Number]
; Purpose: counts the number of occurrences of 'val' in 'tree'
; Pre-conditions: tree is a binary tree (represented as nested lists),
;                 and val is a value that may be in the tree.
; Tests: (count-node '(1 (2 #t 2) 2) 2) => 3

; Signature: mirror-tree(tree)
; Type: [List(T | List(T)) -> List(T | List(T))]
; Purpose: returns a new binary tree (represented as nested lists) that is the
;          mirror image of 'tree'
; Pre-conditions: tree is a binary tree (represented as nested lists)
; Tests: (mirror-tree '(1 (2 #t 3) 2)) => '(1 2 (2 3 #t))

; Signature: make-ok(val)
; Type: [T -> '("ok" T)]
; Purpose: constructs a result with value 'val' and status 'ok'
; Pre-conditions: none
; Tests: (make-ok 42) => '("ok"  42)

; Signature: make-error(msg)
; Type: [String -> '("error" String)]
; Purpose: constructs a result with message 'msg' and status 'error'
; Pre-conditions: none
; Tests: (make-error "oops!") => '("error"  "oops!")

; Signature: ok?(res)
; Type: [T -> Boolean]
; Purpose: returns true if 'res' is a result with status 'ok', false otherwise
; Pre-conditions: none
; Tests: (ok? (make-ok 42)) => #t

; Signature: error?(res)
; Type: [T -> Boolean]
; Purpose: returns true if 'res' is a result with status 'error', false otherwise
; Pre-conditions: none
; Tests: (error? '("error" 42)) => #f
; Tests: (error? (make-error "42")) => #t
```

```
; Signature: result?(res)
; Type: [T -> Boolean]
; Purpose: returns true if the given value is a valid result, either ok or error, false otherwise
; Pre-conditions: none
; Tests:
;       (result? “ok”) => #f
;       (result? “error”) => #f
;       (result? 42) => #f
;       (result? '(ok 3)) => #t
;        (result? '(error "oops")) => #t

; Signature: result->val(res)
; Type: [T1 -> T2]
; Purpose: Extract the value from the given res. If res is not a valid result, return an error.
; Pre-conditions: none
; Tests:
;       (result->val '(ok 42)) => 42
;       (result->val '(error "oops")) => "oops"
;       (result->val “ok”) => ("error" "Error: not a result")

; Signature: bind(f)
; Type: [(T1 -> Result) -> (Result -> Result)]
; Purpose: Create a new function that takes a result as input, applies the given function to the value of the
;           result if it is an 'ok' result, and returns a new result. If the input result is an 'error' result, the
;           new function returns an 'error' result with the same message.
; Pre-conditions: f is a function that takes a value as input and returns a result.
; Tests:
        (define inc-result (bind (lambda (x) (make-ok (+ x 1)))))
        (define ok (make-ok 1))
        (result->val (inc-result ok)) → 2
        (define error (make-error “some error message”))
        (result->val (inc-result error)) → “some error message”
```