Question1 : Theoretical Question

1.1

Special forms are required in order to provide syntax and functionality that cannot be achieved by primitive operators alone. Special forms are typically used to define control flow constructs, such as conditions or loops.

For example, the special form of ‘if statement’:

(if <condition><then - a><else - b>) is evaluated as follows:

Let p=evaluate(<condition>)
If p is true - the value of (<then - a>) is evaluated

if p is false, the value of (<else - b>) is evaluated.

This type of structure cannot be achieved by using only primitive operators and gives us, as programmers, a new way to construct our program.

1.2

There is no program that can be executed in L1 that L0 can’t execute. This is due to the fact that L1 doesn't support recursion and either L0.

which means every expression that was bound with the special form “define” ,can be replaced with the original expression. for example. if in L1 we will write: define(a 2), and write the expression b = a+2. we can simply rewrite it in L0 as: b = 2+2.

1.3

Yes, there are some programs that can be written in L2 but cannot be converted to L20. in particular, programs that contain recursive functions(a variable that is defined as a lambda is passed as an argument, or appears in the body of the lambda). Therefore these references cannot be replaced by their original value.

We can see, for example, the program that was presented in the lecture to find the approximation to a given number’s root value. which the main function in that program used recursive calls.

1.4

Advantages of PrimOps:

- PrimOps are typically very efficient, as they are directly implemented by the interpreter or compiler.
- PrimOps can be used to define a small set of fundamental operations that are commonly used in programming. This can simplify the implementation of higher-level functions and data structures.

Advantages of Closures:

- Closures provide a powerful way to encapsulate data and behavior into reusable functions. This can make code more modular and easier to reason about.
- Closures can be used to define higher-order functions, which take other functions as arguments or return functions as values. This allows for more flexible and expressive programming.

1.5

- **map**: this function can be applied in parallel. The given function is applied to each element of the list in an independent way, with no side effects.

- **filter:** this function can be applied in parallel. The given function is applied to each element of the list in an independent way, with no side effects.

- **reduce**: the reduce function can be applied in a parallel manner only if the given reducer is a commutative / associative procedure.

- **all**: this function can be applied in parallel. The given function is applied to each element of the list in an independent way, with no side effects.

- **compose**: the compose function can be applied in a parallel manner only if the given reducer is a commutative / associative procedure.

1.6

Lexical address is a 'reference' to the location where the ‘VarRef’ is defined.

For example: let take Lexical address that:

- How many levels of definition must be traversed to reach its definition.
- Where it is located in the list of definitions at this level
- we will define primOp as free Lexical address

(lambda (x y)

((lambda (x) (+ **x y**)(+ **x z**))))

->

(lambda (x y)

((lambda (x)

(**[+ free] [x : 0 0] [y : 1 1]**))(**[+ free] [x : 0 0] [z :free]**))

1

)

)

Q1.7 Let us define L31 as the L3 language with the addition of 'cond' special form (as described in practical session 4)

Note: The cond expression **must include at least one cond-clause**, and **must include an else-clause** (in contrast to the decription in practical session 4)

Complete the concrete and abstract syntax of L31:

```
<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><elseclause>) cond-exp(cond-clauses:
                                                          list:cond-clause),
        |  ( quote <sexp> )                    / LitExp(val:SExp)     elseClause: ElseClause)
        |  ( <cexp> <cexp>* )                  / AppExp(operator:CExp,
        |                                                operands:CExp[]))
<binding>  ::= ( <var> <cexp> )                / Binding(var:VarDecl,
                                                         val:Cexp)
```

<ElseClause>::= (<cexp>+)   // elseclause(Val: cexp[])

<cond-clause>::= (<cexp><cexp>+)) // cond-clause(test:cexp, then: list(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>* )
```

[4 points]

;---

;Signature   : take(lst,pos)

;Type   : [(int->Boolean)*list(T1)-> list(T1)]

;Purpose   : gets a list and a number pos and returns a new list whose elements are the first pos elements of the lists.

if the list is shoerter than the pos then return the list.

; Pre-condition : true

; Tests   : (take (list 1 2 3) 2) → '(1 2)

;---

;---

;Signature   : take-map(lst,func,pos)

;Type   : [(int->Boolean)*list(T1)-> list(T1)]

;Purpose   : returns a new list whose elements are the first pos elements mapped by func

; Pre-condition : true

; Tests   : (take-map (list 1 2 3) (lambda (x) (* x x)) 2) → '(1 4)

;---

;---

;Signature   : take-filter(lst pred pos)

;Type   : [(T1->Boolean)*List(T1)*(int->boolean)-> list(T1)]

;Purpose   : returns a new list whose elements are the first pos elements of the list that satisfy the pred.

; Pre-condition : true

; Tests   : (take-filter (list 1 2 3) (lambda (x) (> x 3)) 2) → '()

;---

;---

;Signature   : sub-size(lst size)

;Type   : [(int->Boolean)*list(T1)-> list(T1)]

;Purpose   : returns a new list of all the sublists of list of length size

; Pre-condition : true

; Tests   : (sub-size (list 1 2 3) 3) → '((1 2 3))

;---

;---

;Signature : sub-size-map(lst func size)

;Type : [(int->Boolean)*list(T1)-> list(T1)]

;Purpose : returns a new list of all the sublists of $list$ of length $size$ that all their elements are mapped by $func$.

; Pre-condition : true

; Tests : (sub-size-map '() 0 (lambda (x) (+ x 1))) → '(())

;---

;---

;Signature : root(tree)

;Type : [list(T) -> T]

;Purpose : gets a list representing a tree and returns the value of the root

; Pre-condition : true

; Tests : (root '(1 (#t 3 #t) 2) #t) → 1

;---

;---

;Signature : left(tree)

;Type : [list(T) -> list(T)]

;Purpose : gets a list representing a tree and returns the subtree of the left son, or an empty list if there is no left son.

; Pre-condition : true

; Tests : (left '(1 (#t 3 #t) 2) #t) → (#t 3 #t)

;---

;---

;Signature : right(tree)

;Type : [list(T) -> list(T)]

;Purpose : gets a list representing a tree and returns the subtree of the right son, or an empty list if there is no left son.

; Pre-condition : true

; Tests : (right '(1 (#t 3 #t) 2) #t) → 2

;---

;---

;Signature : count-nodes(tree val)

;Type : [list(T)*int -> int]

;Purpose : given a list representing a $tree$ and an atomic val, returns the number of nodes whose value is equal to val.

; Pre-condition : true

; Tests : (count-node '(1 (#t 3 #t) 2) #t) → 2

;---

;---

;Signature : mirror-tree(tree)

;Type : [list(T) -> list(T)]

;Purpose : given a list representing a $tree$, returns the mirrored tree.

; Pre-condition : true

; Tests : (mirror-tree '(1 (#t 3 4) 2)) -> '(1 2 (#t 4 3))

;---

;Signature : make-ok(val)

;Type : [val -> OK(val)]

;Purpose : gets a value and returns an ok structure for the $value$ of type result.

; Pre-condition : true

; Tests : (define ok (make-ok 1))

;---

;Signature : make-error(message)

;Type : [message -> EROR(message)]

;Purpose : gets an error $message$ and returns an error structure for the $message$ of type result.

; Pre-condition : true

; Tests : (define error (make-error “some error message”))

;---

;Signature : ok?(pred)

;Type : [pred -> Booolean]

;Purpose : type predicate for ok.

; Pre-condition : true

; Tests : (ok? ok) → #t

;--

;Signature : error?(pred)

;Type : [pred -> Booolean]

;Purpose : type predicate for error.

; Pre-condition : true

; Tests : (error? ok) → #f

;--

;Signature : result?(pred)

;Type : [pred -> Booolean]

;Purpose : type predicate for result.

; Pre-condition : true

; Tests : (result? ok) → #t

;--

;Signature : result(val)

;Type : [val -> int]

;Purpose : gets a $result$ structure and returns the value it represents, or the error message for error. If the given result is not a result, return an error structure with the message “Error: not a result”

; Pre-condition : true

; Tests : (result->val ok) → 1

;--

;Signature : bind(func)

;Type : [function -> function]

;Purpose : bind which given a function $func$ from a non-result to result, returns a new function which given a result, returns the activation of $func$ on its value or an error structure accordingly.

; Pre-condition : true

; Tests : (define inc-result (bind (lambda (x) (make-ok (+ x 1)))))

;--