Answers to the theoretical part:

Q1.1: Special forms are required in programming languages because they have unique behavior that cannot be replicated by simple primitive operators or functions.

For example: "if" is a special form because its behavior cannot be replicated by a function or operator, because in if we can make decisions based on a Boolean condition and in operators and functions we calculate everything in advanced before using the operator, func…

Q1.2: Every prog in L1 can be done in L0 by simply replacing the variable that we used define on with its expression that was given to the define with him.

Q1.3: Every prog in L2 can be done in P20 by simply writing the whole lambda every time we want to use it instead of calling it by its name we defined in define.

Q1.4: Advantages of PrimOp:

1. Efficiency: PrimOp functions are implemented as built-in operations of the interpreter or compiler, which means that they are executed more efficiently than user-defined functions.

2.Simplicity: PrimOps are easy to define and implement as they are simple functions with fixed inputs and outputs.

Advantages of Closure:

1.Flexibility: Closure allows the programmer to define and manipulate functions as values, which provides more flexibility and expressiveness in the code.

2.Encapsulation: Closures can encapsulate data and functions together, which can make it easier to manage and maintain code by grouping related functionality.

Q1.5:

map- can be done parallel because there is no dependency of some elements on other elements

reduce- should be sequential because the order of the elements matter, for example: reduce((a: number, b: number) => a/b,1, [1,2]) would have 2 different results if we would run it in parallel on the elements of the lst.

filter- can be done parallel because there is no dependency of some elements on other elements

all- can be done parallel because there is no dependency of some elements on other elements

compose- should be sequential because the order of the elements matter, for example:

f = (a: number) => a-a

g = (a: number) => a/a

compose(f,g) would always return 0 while compose(g,f) would always cause run-time error because of div by 0.

Q1.6: Lexical address refers to the way in which the memory location of a variable or function is determined based on its position within the source code.

Example:

```
(lambda (a b)
        (Lambda (x)
                (+ b x)
        )
)
```

[a: 1 0] [b: 1 1] [x: 0 0]

Q1.7:

```
>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)
>condClause> ::= (<cexp> <cexp>+)
                |(else <cexp>+)
>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 <condClause>+) / CondExp(condClauses: condClauses[])
         |( quote <sexp> ) / LitExp(val:SExp)
         |( <cexp> <cexp>* ) / AppExp(operator:CExp,
         |operands:CExp(([]
>binding> ::= ( <var> <cexp> ) / Binding(var:VarDecl,
                                          val: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:

; Signature: (lambda (lst pos) ...)

; Type: Procedure

; Purpose: Takes a list and a positive integer and returns a new list containing the first pos elements of the input list.

; Pre-conditions: lst is a list and pos is a positive integer.

; Tests:

(check-equal? (take '(1 2 3 4 5) 0) '())

(check-equal? (take '(1 2 3 4 5) 3) '(1 2 3))

(check-equal? (take '(1 2 3) 4) '(1 2 3))

; Signature: (lambda (lst func pos) ...)

; Type: Procedure

; Purpose: Takes a list, a function, and a positive integer and returns a new list of the first pos elements of applying the function to each element of the input list.

; Pre-conditions: lst is a list, func is a function, and pos is a positive integer.

; Tests:

(check-equal? (take-map '(1 2 3 4 5) (lambda (x) (* x x)) 3) '(1 4 9))

(check-equal? (take-map '(apple banana orange) (lambda (x) (string->symbol x)) 2) '(apple banana))

(check-equal? (take-map '(1 2 3) (lambda (x) (+ x 1)) 4) '(2 3 4))

; Signature: (lambda (lst pred pos) ...)

; Type: Procedure

; Purpose: Takes a list, a predicate function, and a positive integer and returns a new list containing the first pos elements of the input list for which the predicate function returns true.

; Pre-conditions: lst is a list, pred is a function that takes one argument and returns a boolean, and pos is a positive integer.

; Tests:

(check-equal? (take-filter '(1 2 3 4 5) odd? 3) '(1 3 5))

```
(check-equal? (take-filter '(apple banana orange) (lambda (x) (string=? x "banana")) 2) '(banana))

(check-equal? (take-filter '(1 2 3) (lambda (x) (> x 1)) 4) '(2 3))


; Signature: (lambda (lst size) ...)

; Type: Procedure

; Purpose: Takes a list and a positive integer and returns a list of sublists of the input list where each sublist has size elements.

; Pre-conditions: lst is a list and size is a positive integer.

; Tests:

(check-equal? (sub-size '(1 2 3 4 5) 2) '((1 2) (3 4)))

(check-equal? (sub-size '(a b c d e f) 3) '((a b c) (d e f)))

(check-equal? (sub-size '(1 2 3 4 5) 4) '((1 2 3 4) (2 3 4 5)))


; Signature: (lambda (lst func size) ...)

; Type: Procedure

; Purpose: Applies a function to each element of a list, and then returns a list of sublists where each sublist has size elements.

; Pre-conditions: lst is a list, func is a function, and size is a positive integer.

; Tests:

(check-equal? (sub-size-map '(1 2 3 4 5) (lambda (x) (* x x)) 2) '((1 ...)))


; Signature: (define count-node (tree val))

; Type: Procedure

; Purpose: Count the number of nodes in a binary tree that have a specific value.

; Pre-conditions: 'tree' is a binary tree, 'val' is any Scheme value.

; Tests:

  (assert (= (count-node '() 'a) 0))
```

```
  (assert (= (count-node '(a) 'a) 1))
  (assert (= (count-node '(a b) 'a) 1))
  (assert (= (count-node '(a (b) c) 'b) 1))
  (assert (= (count-node '(a (b) (a)) 'a) 2))


; Signature: (define mirror-tree (tree))
; Type: Procedure
; Purpose: Return the mirror image of a binary tree.
; Pre-conditions: 'tree' is a binary tree.
; Tests:
  (assert (equal? (mirror-tree '()) '()))
  (assert (equal? (mirror-tree '(a)) '(a)))
  (assert (equal? (mirror-tree '(a b)) '(a (b) ())))
  (assert (equal? (mirror-tree '(a (b) (c d))) '(a ((d) c) ((b)))))


; make-ok : any -> (cons "ok" any)
; Returns a cons cell with the string "ok" as its car and the given value as its cdr.
; None.
; (make-ok 10) => '("ok" . 10)

; make-error : string -> (cons "error" string)
; Returns a cons cell with the string "error" as its car and the given message as its
cdr.
; None.
; (make-error "invalid input") => '("error" . "invalid input")
```

```
; ok? : any -> boolean
; Returns true if the given value is a cons cell with the string "ok" as its car.
; None.
; (ok? '("ok" . 10)) => #t
; (ok? '("error" . "invalid input")) => #f

; error? : any -> boolean
; Returns true if the given value is a cons cell with the string "error" as its car.
; None.
; (error? '("ok" . 10)) => #f
; (error? '("error" . "invalid input")) => #t

; result? : any -> boolean
; Returns true if the given value is either an "ok" or "error" cons cell.
; None.
; (result? '("ok" . 10)) => #t
; (result? '("error" . "invalid input")) => #t
; (result? '("other" . 5)) => #f

; result->val : any -> any
; Returns the cdr of the given "ok" or "error" cons cell.
; Requires the input to be an "ok" or "error" cons cell.
; (result->val '("ok" . 10)) => 10
; (result->val '("error" . "invalid input")) => "invalid input"

; bind : (any -> any) -> (cons "ok" any) or (cons "error" any) -> (cons "ok" any) or
(cons "error" any)
; Takes a function f and a result res. If res is an "error" cons cell, bind returns res
unchanged.
```

; Otherwise, bind applies f to the cdr of res and returns the result as an "ok" cons cell.

; None.

; (bind (lambda (x) (make-ok (+ x 1))) (make-ok 10)) => '("ok" . 11)

; (bind (lambda (x) (make-ok (+ x 1))) (make-error "invalid input")) => '("error" . "invalid input")