Q1.1

The programming languages require special forms because they provide a means of determining and controlling the flow of execution, defining new constructs that can only be used in specific cases where using ordinary functions or operators are not available. we can't simply define special forms as primitive operators because they provide different evaluation rules that are not possible to implement using primitive operators. for example, "define" is a special form that creates named variables and used in scheme to bind a name to a value of an expression. also it has a side effect of changing the environment in which it is evaluated. so it cant be defined as a primitive operator.

Q1.2

No, because L1 doesn't contain recursive forms, so every program in L1 can be converted into a program in L0 and it is possible to replace every variable reference with its corresponding defined value expression.

Q1.3

Yes, because in L2 it is possible to define user procedures that may contain recursive calls, so there are cases that variable reference can not be replaced with their defined value expression. For example, when a lambda function is defined and it is referred to itself by appearing within its own body.

Q1.4

<u>Advantages of PrimOp:</u>

PrimOp representation is efficient and simple, in addition it reflects the implementation language's primitive operations.

<u>Advantages of Closure:</u>

The use of Closure representation provides a higher flexibility and abstraction in programming. In addition, it enables treating procedures as values.

Q1.5

<u>map:</u>

Can be applied in parallel. Because in functional programming, functions do not have any side effects and their output only depends on their input, so the evaluation of a function applied to one input has no effect on the evaluation of the same function applied to another input.

<u>reduce:</u>

Can be applied in parallel just in case the reducer procedure is associative and commutative.

filter:

Can be applied in parallel. Because applying a boolean function or predicate to one input value based on if the input satisfies the condition of the function, so the evaluating process doesn't affect the evaluation of the same function on another input.

all:

Can be applied in parallel. Because applying a boolean function or predicate to one input value based on if the input satisfies the condition of the function, so the evaluating process doesn't affect the evaluation of the same function on another input.

compose:

Can be applied in parallel just in case the procedure is associative.

Q1.6

We define the lexical address of a bound variable reference in the form

[var : depth pos] in a way that unambiguously describes to which variable declaration it is bound. For example:

```
((lambda (x y)              ;1
  ((lambda (x)  (+ x y))    ;2
   (+ x x))  1)             ;3
```

The lexical addresses annotations for the example above is:

```
(lambda (x y)                              ;1
  ((lambda  (x)  (+ [x : 0 0] [y : 1 1] ))     ;2
   (+ [x : 0 0]  [x : 0 0] ))  1)              ;3
```

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)
<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>+ <ElseClause> /
CondExp(CondClauses:List(CondClauses))
| ( quote <sexp> ) / LitExp(val:SExp)
| ( <cexp> <cexp>* ) / AppExp(operator:CExp,
| operands:CExp[]))
<binding> ::= ( <var> <cexp> ) / Binding(var:VarDecl,
val:Cexp)
<CondClause> ::= (<cexp> <cexp>+)  / CondClause(test: CExp, then:
CExp[])
<ElseClause> ::= (<cexp>) / ElseClause(then: 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>* )
```

Q2:

2.1.a:

```
; Signature: take(lst,x)

; Type:[List*Number -> List]

; Purpose:returns a new list whose elements are the first x elements of
the lst

; Pre-conditions:x is a natural number,lst is a list

; Tests: (take(list123)2) => '(12)
```

```
; Signature: take-map(lst,f,x)

; Type:[List*function*Number -> List]

; Purpose: returns a new list whose elements are the first x elements of
the lst that satisfy f
```

```
; Pre-conditions: x is a natural number,lst is a list, f is function

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

```
; Signature:take-filter(lst,pred,x)
; Type: [List*function*Number -> List]

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

; Pre-conditions: x is a natural number , lst is a list, f is function
; Tests: (take-filter (list 1 2 3 4) (lambda (x) (> x 1)) 2) => '(2 3)
```

2.1.b:

```
; Signature:sub-size(lst,size)
; Type:[List(T)*Number -> List(List(T))]
; Purpose: returns a new list of all the sublists of lst of length size
; Pre-conditions: size is a natural number , lst is a list
; Tests: (sub-size (list 1 2 3) 1) => '((1) (2) (3))
```

```
; Signature:sub-size-map(lst,f,size)
; Type:[List(T)*function*Number -> List(List(T))]
; Purpose: returns a new list of all the sublists of lst of length size that all their elements
are mapped by f
; Pre-conditions: size is a natural number,lst is a list, f is function
; Tests: (sub-size-map (list 1 2 3) (lambda (x) (+ x 1)) 3) => '((2 3 4))
```

2.2.a:

```
; Signature:root(tree)
; Type:[List(T) -> T ]
; Purpose: returns the root value of the given tree
; Pre-conditions: Tree is a list
; Tests: (root ‘(0 1 2)) => 0
```

```
; Signature:right(tree)
; Type: [List(T) -> List(T)]
; Purpose: returns the right sub-tree of the given tree
; Pre-conditions: Tree is a list
; Tests: (right ‘(0 1 2)) => ‘(2)
```

```
; Signature:left(tree)
; Type: [List(T) -> List(T)]
; Purpose: returns the left sub-tree of the given tree
; Pre-conditions: Tree is a list
; Tests: (left ‘(0 1 2)) => ‘(1)
```

2.2.b:

```
; Signature: count-node(tree,val)
; Type:[List(T)*T - > Number]
; Purpose:counting how much the val repeated in the tree
; Pre-conditions:Tree is a list
; Tests: (count-node '(1 (#t 3 #t) 2) #t) => 2
```

2.2.c:

```
; Signature:mirror-tree(Tree)
; Type: [List(T) -> List(T)]
; Purpose: mirror the given tree and return it
; Pre-conditions: Tree is a list
; Tests: (mirror-tree ‘(1 (#t 3 4) 2) ) => '(1 2 (#t 4 3))
```

2.3.a:

```
; Signature:make-ok(val)
; Type:[T-list(T)]
; Purpose: returns an ok structure for the val of type result
; Pre-conditions:nothing
; Tests: (define ok (make-ok 1))
```

```
; Signature:make-error(message)
; Type: [T-list(T)]
; Purpose: returns an error structure for the message of type result
; Pre-conditions:nothing
; Tests: (define error (make-error “some error message”))
```

```
; Signature:ok?(x)
; Type: [List(T) -> bool]
; Purpose: checks if x is an ok structure
; Pre-conditions: x is a ok structure
; Tests: (define ok (make-ok 1))
 (ok? ok) → #t


; Signature:error?(x)
; Type: [List(T) -> bool]
; Purpose: checks if x is an error structure
; Pre-conditions: x is a error structure
; Tests: (define ok (make-ok 1))
(error? ok) → #f


; Signature:result?(x)
; Type: [List(T) -> bool]
; Purpose: checks if x is an error structure or an ok structure
; Pre-conditions: x is a ok structure or error structure
; Tests: (define ok (make-ok 1))
(result? ok) → #t


; Signature: result->val(x)
; Type: [List(T) -> T]
; Purpose:returns the result of the given structure
; Pre-conditions: x is a ok structure or error structure
; Tests: (define ok (make-ok 1))
(result->val ok) → 1
```

2.3.b:

; Signature:bind(f)
; Type:; Purpose: returns a new function which given a result, returns the activation of $func$ on its value or an error structure accordingly
; Pre-conditions:f is function
; Tests: (define inc-result (bind (lambda (x) (make-ok (+ x 1))))
(define ok (make-ok 1)) (result->val (inc-result ok)) → 2
Helper finctions:

```
; Signature: atom?(x)
; Type:[T -> bool]
; Purpose:checks if x is an pair or empty
; Pre-conditions: nothing
; Tests: (atom? 2) => #t
```