Q1.1

As discussed in lecture 5, there is no real need for binding (define) in programming languages, it only improves readability and convenience. It is recommended but not required and therefore, if L11 is defined as L1 excluding "define", every program that can be written in L1 can be written in L11. You can replace every "define" by copying and substituting the value manually wherever it is needed.

Q1.2

As discussed in lecture 5, there is no real need for binding (define) in programming languages, it only improves readability and convenience. It is recommended but not required and therefore, if L21 is defined as L2 excluding "define", every program that can be written in L2 can be written in L21. You can replace every "define" by copying and substituting the value manually wherever it is needed.

Recursions can work without 'define' by using a complex work around.

Q1.3

There are programs that can be written in L2 but not in L22 such as:

L2:

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

The value of the program is a closure with two arguments and cannot be created in any way in L22.

Q1.4

There are programs that can be written in L2 but not in L23 such as the well-known function "map":

```
(define map
        (lambda (f lst)
                (if (eq? lst '())
                        '()
                        (cons (f (car lst))  (map f (cdr lst)))
                )
        )
)
```

There is no single behavior for the function, it depends on the mapping function.
With no high order functions each function would have only one defined behavior.

Q2.1

a.

```
;; <program> ::= (L3 <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)
;;          | <prim-op>                        / PrimOp(op: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[]))
;;          | ( quote <sexp> )                 / LitExp(val:SExp)
;;          | ( <cexp> <cexp>* )               / AppExp(operator:CExp, operands:CExp[]))
;; <binding>  ::= ( <var> <cexp> )              / Binding(var:VarDecl, val:Cexp)
;; <prim-op>  ::= + | - | * | / | < | > | = | not |  and | or | eq? | string=?
;;                  | cons | car | cdr | dict | get | dict? | pair? | number? | list
;;                  | boolean? | symbol? | string?      ##### L3
;; <num-exp>  ::= a number token
;; <bool-exp> ::= #t | #f
;; <var-ref>  ::= an identifier token
;; <var-decl> ::= an identifier token
;; <sexp>     ::= symbol | number | bool | string |
;;                (<sexp>+ . <sexp>) | ( <sexp>* )       ##### L3
```

Q2.2

a.

```
;; <program> ::= (L32 <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[]))
;;          | ( dict <binding>*)               / DictExp(bindings:Bindings[])
;;          | ( if <cexp> <cexp> <cexp> )      / IfExp(test: CExp, then: CExp, alt: CExp)
;;          | ( let ( binding* ) <cexp>+ )     / LetExp(bindings:Binding[], body:CExp[]))
;;          | ( quote <sexp> )                 / LitExp(val:SExp)
;;          | ( <cexp> <cexp>* )               / AppExp(operator:CExp, operands:CExp[]))
;; <binding>  ::= ( <var> <cexp> )              / Binding(var:VarDecl, val:Cexp)
;; <prim-op>  ::= + | - | * | / | < | > | = | not |  and | or | eq? | string=?
;;                  | cons | car | cdr | pair? | number? | list
;;                  | boolean? | symbol? | string?      ##### L3
;; <num-exp>  ::= a number token
;; <bool-exp> ::= #t | #f
;; <var-ref>  ::= an identifier token
;; <var-decl> ::= an identifier token
;; <sexp>     ::= symbol | number | bool | string |
;;                (<sexp>+ . <sexp>) | ( <sexp>* )       ##### L3
```

Q2.4

a. 2.1
No change needed. These use primitives or quoted literals, so evaluation happens before application.
2.2
Needs adjustments. dict is a special form — its operands aren't pre-evaluated, so the interpreter must handle delayed evaluation which requires modification.
2.3
In a similar way to implementation 2.1, **no change needed**. These use primitives or quoted literals, so evaluation happens before application.

b. 2.1
No change needed. Primitives don't depend on the environment.
2.2
No change needed, although ValueToLitExp becomes unnecessary — values are handled directly via environments.
2.3
Needs 'letrec' for recursion as studied in class.

c. In 2.1 and 2.3, (dict (a 1) (b 2)) is invalid because the parser reads (a 1) as a function call, not a pair. To avoid this, we must quote the input: (dict '((a . 1) (b . 2))), so it's parsed as a LitExp. This form works the same in applicative and normal order since the values aren't evaluated.
In 2.2, dict is a special form with custom parsing. It treats (a 1) as a pair, not an application, and builds a DictExp node. This allows evaluation of the keys and values at runtime, supporting full expressions and different evaluation orders.

d. Yes in 2.2, values in the dict can be any valid L3 expression (like: ifs, lambdas and nested dict), because the parser creates AST nodes for them, and they are evaluated at runtime.
In 2.1 and 2.3, values are part of a quoted S-expression, so expressions like: (if #t 1 2) are treated as data, not code they're not parsed or evaluated as expressions.

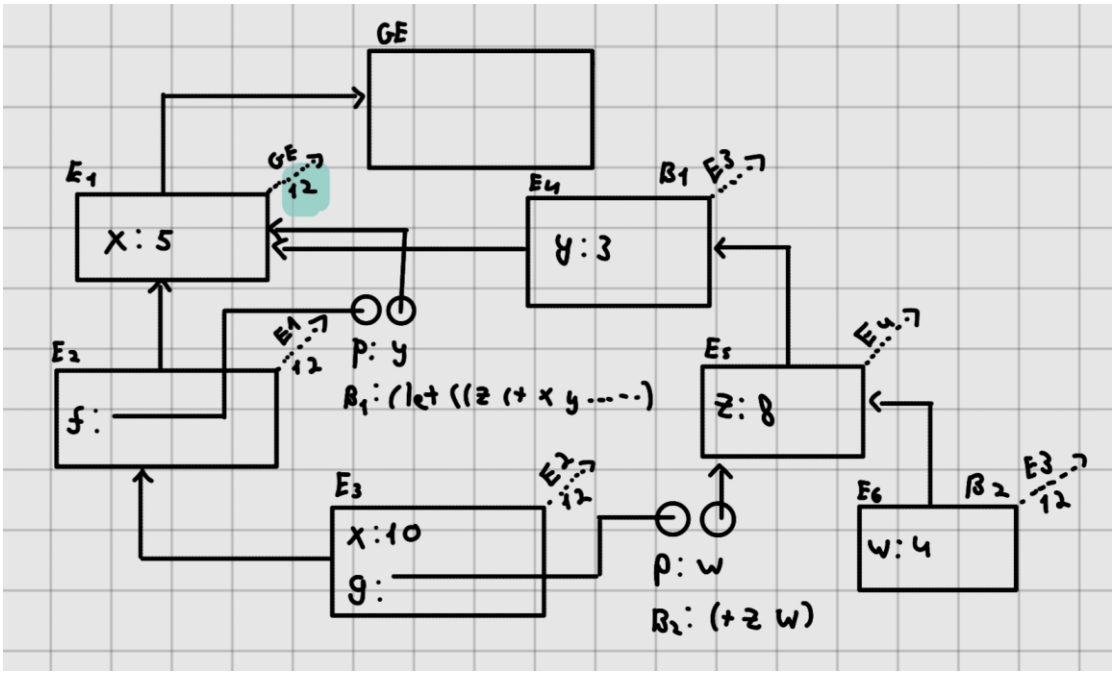
As a result, some L32 programs using expression values in dicts (like conditionals or lambdas) cannot be correctly transformed into L3 using the method in 2.5, because the transformed version wraps them in a quote, losing their semantics.

e. We would prefer the special form implementation(2.2). It has a few advantages:
- **supports full expressions** as values (like: if, lambda…)
- **Cleaner syntax** for there is no need for quotes.
- **Flexibility**. Can handle both applicative and normal order with proper evaluation logic.

On the other hand - Disadvantages:

- **Requires parser changes** - need to define and handle a new special form.
- **Sensitive to evaluation order** - implementation must account for delayed evaluation in normal order.

Q4.1



Q4.2

```
3  (let ((x 2)
4        (h (lambda (x)
5             (lambda (y)
6               (* x y)))))
7    (let ((f (h 3)))
8      (f 2)))
```