question 1.1:

If we forbid the appearance of the defined variable in its defined values, any var reference in L1 can be replaced by its defined value expression.

question 1.2:

Yes. Since L2 supports user procedure which may contain recursion calls, a variable which is defined by lambda may appear in the body of the lambda as a recursion call. So the var references cannot be replaced by their defined value, unless we apply the “y-combinator” pattern.

question 1.3:

No , we can simulate any function with more than one parameter in L2 through currying or nesting

example:

```
(lambda (x y) (+ x y)) ;; L2
;; L22 equivalent using currying:
(lambda (x) (lambda (y) (+ x y)))
```

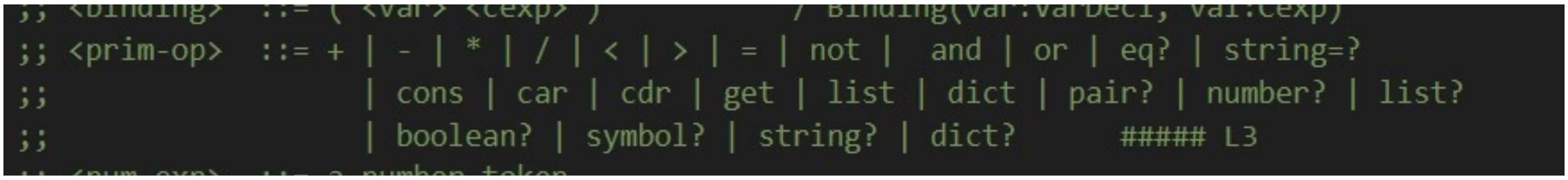
question 1.4:

Yes, there are programs in L2 which cannot be transformed into equivalent programs in L23, for example:

```
(lambda (f x) (f x)) ;
```

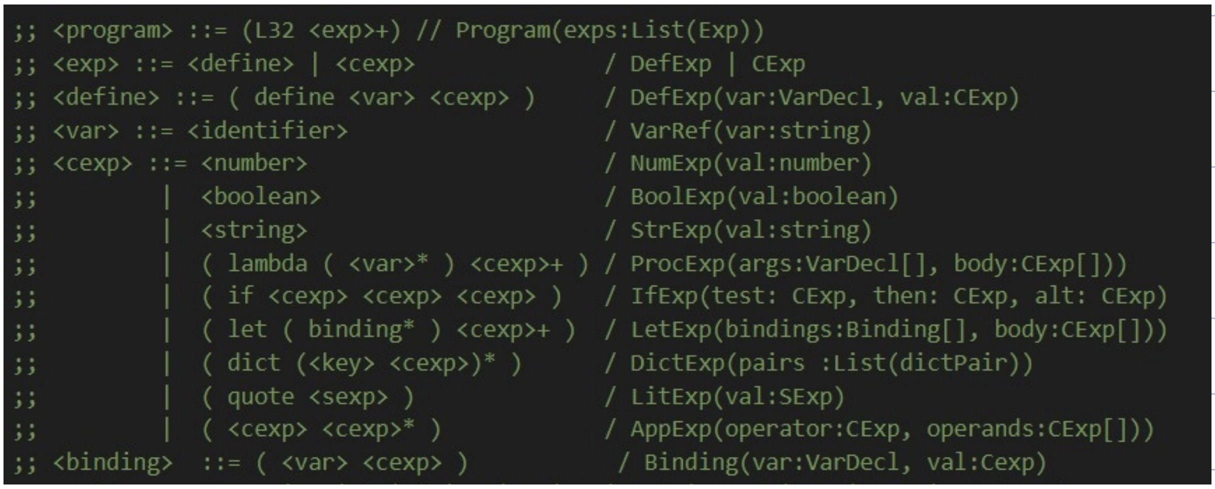
this structure is not allowed in L23 because a function cannot get another function

Question 2.1.a:



```
;; <binding>  ::= ( <var> <cexp> )           / Binding(var:VarDecl, val:Cexp)
;; <prim-op>  ::= + | - | * | / | < | > | = | not |  and | or | eq? | string=?
;;                  | cons | car | cdr | get | list | dict | pair? | number? | list?
;;                  | boolean? | symbol? | string? | dict?      ##### L3
;; <num-exp>  ::= a number token
```

Question 2.2.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[]))
;;         |  ( if <cexp> <cexp> <cexp> )   / IfExp(test: CExp, then: CExp, alt: CExp)
;;         |  ( let ( binding* ) <cexp>+ )  / LetExp(bindings:Binding[], body:CExp[]))
;;         |  ( dict (<key> <cexp>)* )      / DictExp(pairs :List(dictPair))
;;         |  ( quote <sexp> )              / LitExp(val:SExp)
;;         |  ( <cexp> <cexp>* )            / AppExp(operator:CExp, operands:CExp[]))
;; <binding>  ::= ( <var> <cexp> )           / Binding(var:VarDecl, val:Cexp)
```

(2.4)

a) In 2.1 no need to change because we added primitive operations only. in 2.2, 2.3 we need to modify because we must force evaluate expressions for all fields.

b) No the environment model changes nothing that has to do with the dictionaries in all implementations

c) In 2.2, we define dict as a special form, meaning the parser or evaluator handles it before normal evaluation, But this is not possible in standard parsing of primitive operators or user procedures, because: Primitive ops and user procedures evaluate arguments first, so (a 1) is not data — it's an application, 2.2 can control the evaluation and treat such syntax as data, enabling (a 1) to be interpreted as a (key . value) pair.

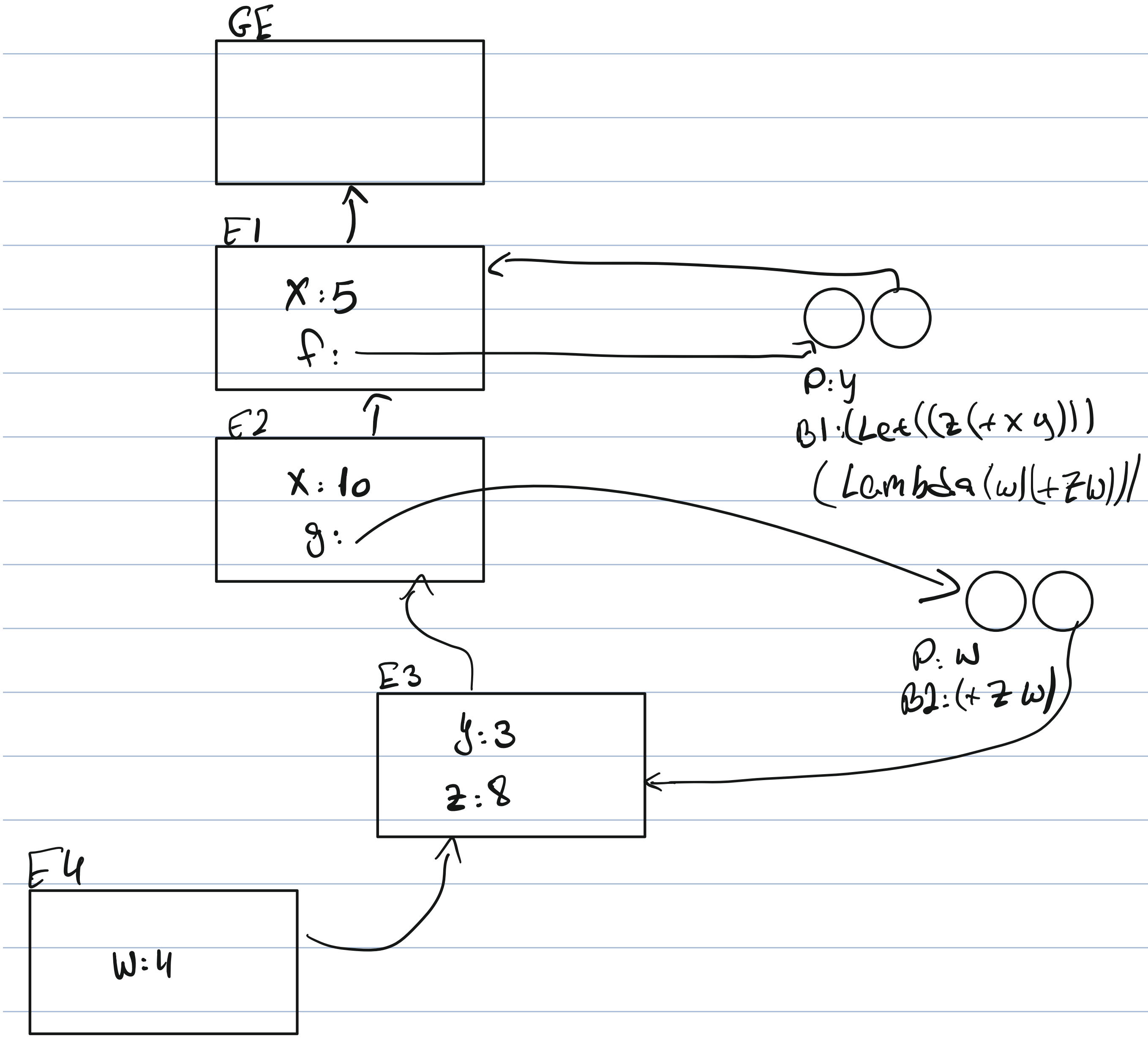
d) Yes, for example: (dict (a (+ 1 2)) (b (lambda (x) (* x x)))) and storing the number 3 and a procedure, but 2.1/2.3 only accept a quoted list of literal pairs and never evaluate inside that literal.

No, there are no L32 expressions that cannot be transformed to L3 using method 2.5 .

e) For us, 2.2 is recommended because it is more natural and readable than 2.1 and 2.3, it is also has clearer syntax for users and allows precise parsing control.

For example : in 2.1 and 2.3, the syntax requires quoting a list of pairs which is not clear enough.

4.a:



Output: 12

4.b:

(Let* (x 2) (h lambda (x) (lambda (y) (* x y))))

(Let (x 3))

(Let (f (h x)))

(F 2)