Assignment 2

Question 1: Language variations:

1. No. Every L1 program using define can be transformed into an equivalent L11 program without it.
   The define form is only used for global variable declarations, which can be simulated using a let expression to bind values locally.
2. Yes. In L2, define can be used to create **recursive functions**, which cannot be directly expressed using let.
   To enable recursion, a function must be able to reference itself by name — which let does not allow.
   Example:

```
(define fact
  (lambda (n)
    (if (= n 0) 1 (* n (fact (- n 1))))))
```

   This recursive function cannot be rewritten in pure L21 without tricks like fixed-point combinators (e.g., Y-combinator), which are outside standard L2 semantics.

3. No. Any L2 program can be transformed into an equivalent L22 program using **currying** and **nesting**.
   Multiple parameters can be rewritten as nested single-parameter lambdas, and multiple expressions in the body can be wrapped in sequencing constructs like begin.
4. Yes. L2 allows higher-order functions, but L23 does not.
   Any program that **passes a function as an argument** cannot be rewritten in L23.
   Example:

```
(define apply-twice
  (lambda (f x) (f (f x))))
```

   This function receives another function f as an argument and applies it twice. Such behavior is not possible in L23 due to the restriction on passing functions.

Question 2.1:

```
<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)
            | (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 | eq? | string=? |
              cons | car | cdr | list | pair? | list? | dict |dict? | get
              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>*)
```

Question 2.2:

<cexp> :: = …

| (dict (<binding>+))	/ DictExp(entries: Binding[])
| (get <cexp> <cexp>)	/ GetExp(dict: CExp, key: CExp)
| (dict? <cexp>)	/ IsDictExp(val: CExp)

Question 2.4:

a. Yes, some implementations should be modified for normal order, especially the user-level version (Q2.1): unevaluated arguments are passed to get, which may expect a fully evaluated list.

b. 2.1 – no change. It does its work inside the primitive code.
2.2 – no change. It only builds a constant list.
2.3 – might need care if the closure captures variables that can later change. If I keep everything immutable, it's fine.

c. With a quote: the inner list is **data**, so every version can read it.
Without the quote: each (a 1) is parsed as a function call.
Versions 2.1 and 2.3 evaluate their arguments first, try to call a, and crash.
Version 2.2 has its own parser rule, so it grabs the fields **before** evaluation and therefore accepts the nicer syntax.

d. Values allowed only in the special form: any inline expression like (lambda (x) (+ x 1)), (random 10), etc.
In 2.1/2.3 you would need an extra quote, which either evaluates the expression too early or requires awkward quoting. After applying the transformation of 2.5 every L32 expression can be converted to L3; without it, any program containing DictExp cannot be written directly in L3.

e. Version 2.1 – primitive operator:
Advantages:

- Easy to add to evaluator only.
- Good performanceme.

Disadvantages:

- User must write (dict '((a . 1) …)) (quoting hassle).
- Breaks only if normal-order thunk not handled, but syntax is still clumsy.

Version 2.2 – special form:
Advantages:

- Clean syntax: (dict (a 1) (b 2)).
- Works in both applicative and normal order.
- Supports arbitrary expressions as values.

Disadvantages:

- Requires adding a new parse rule and an eval clause.

Version 2.3 – user-level closure:
Advantages:

- No interpreter change; just a macro / compiler pass.
- Works after desugaring to L3.

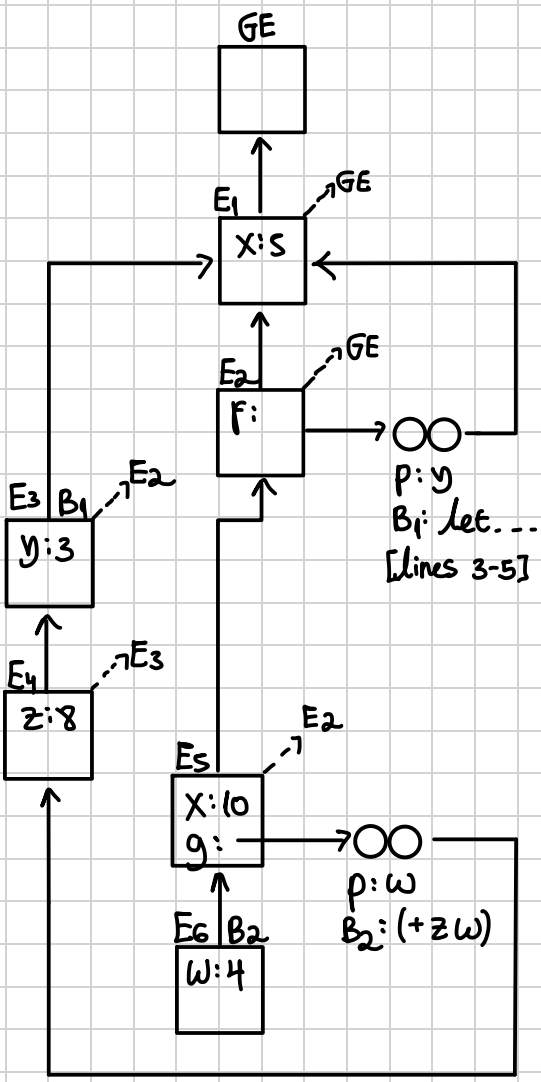
Disadvantages:

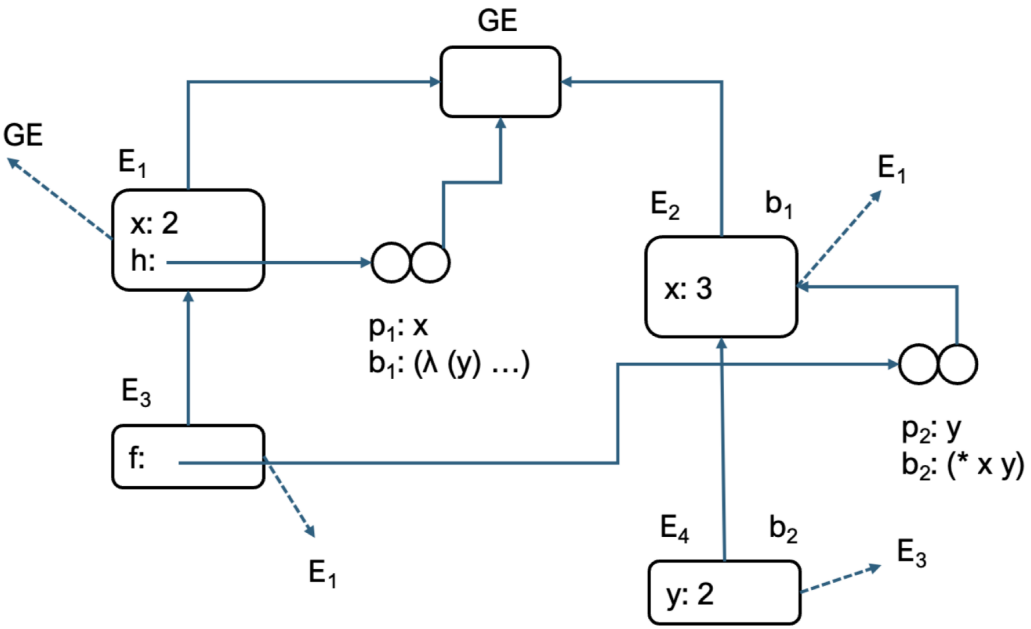
- Harder for users to write by hand.
- Error messages point to generated code.
- Needs explicit forcing in normal order.

Preferred: version 2.2 because it gives the nicest user syntax, avoids quoting pitfalls, and remains stable across evaluation strategies while requiring only moderate implementation effort.

```
1  (let* ((x 5)
2         (f (lambda (y)
3              [(let ((z (+ x y)))
4                 (lambda (w)
5                   (+ z w))))])B2
6    (let ((x 10)
7          (g (f 3)))
8      (g 4)))
9
```

B1






```
(let*  ((x 2)
        (h (lambda (x)
             (lambda (y)
               (* x y)))))

(let ((f (h 3)))
     (f 2)))
```