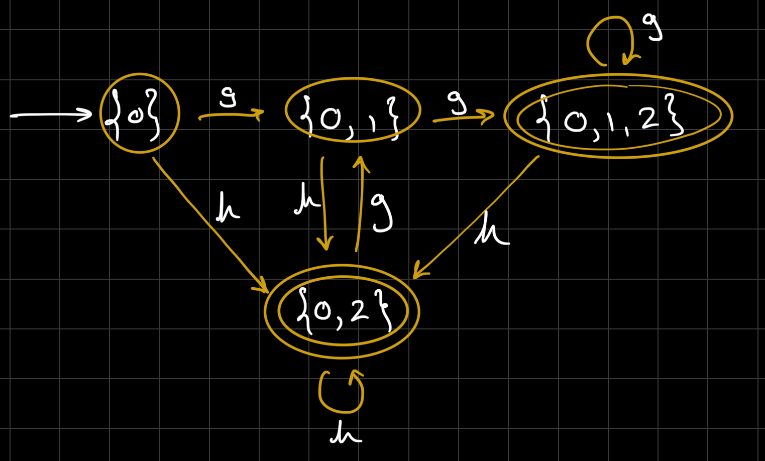
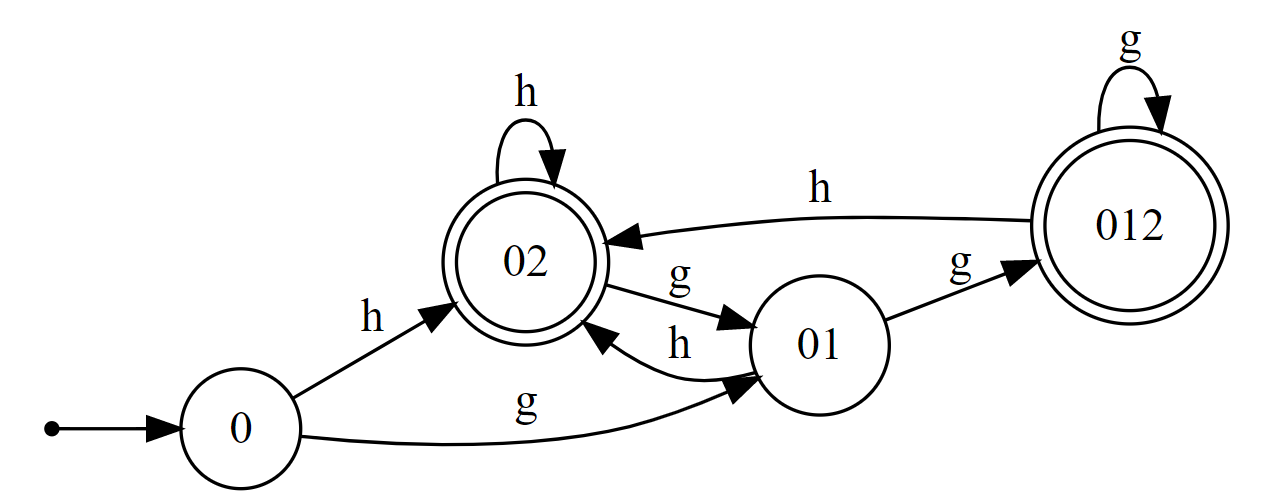
**Collaborative Solution 2022**

**Q1**

**a)**



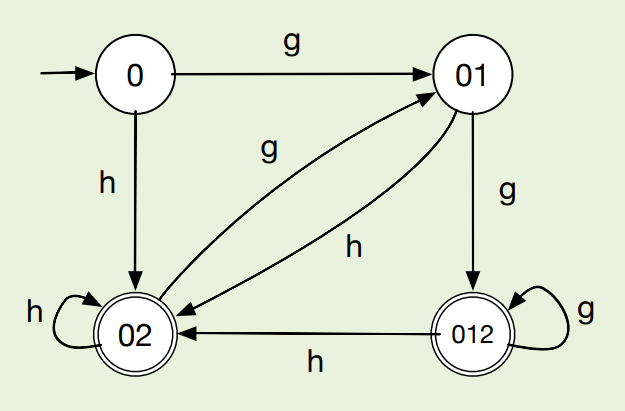


Courtesy of <https://hritikbhandari.me/NFA-to-DFA-Converter/>

^ DO NOT OPEN THE LINK! IT CONTAINS BROWSER HACK! (Unless you are a bold-ass motherfucker)

Why do so many of these have browser hacks

**Naranker’s solution:**



**b) i)**

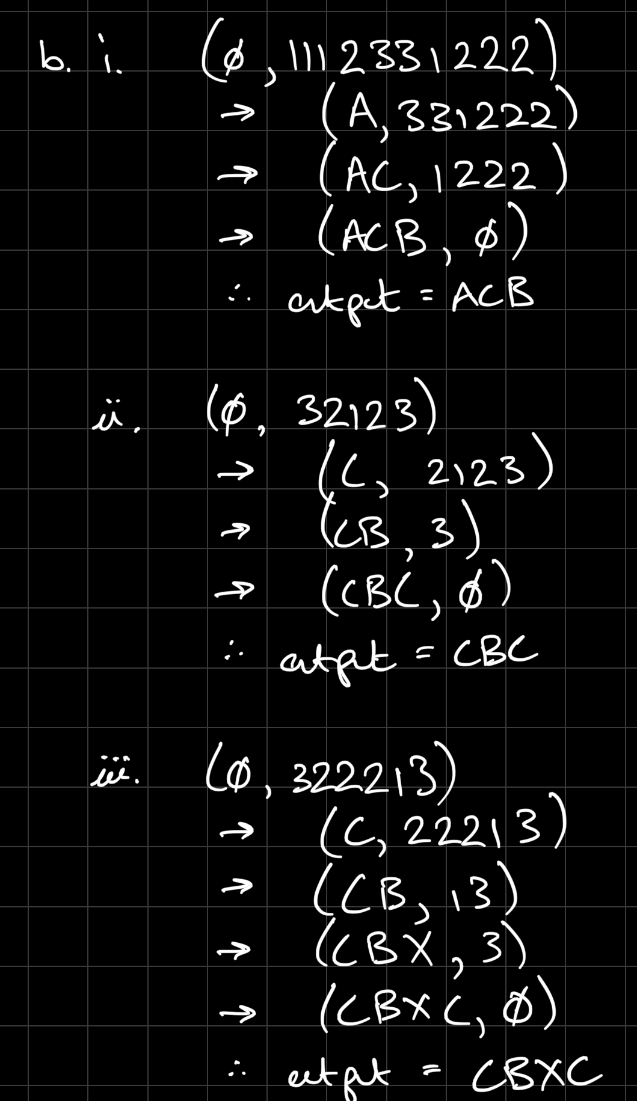
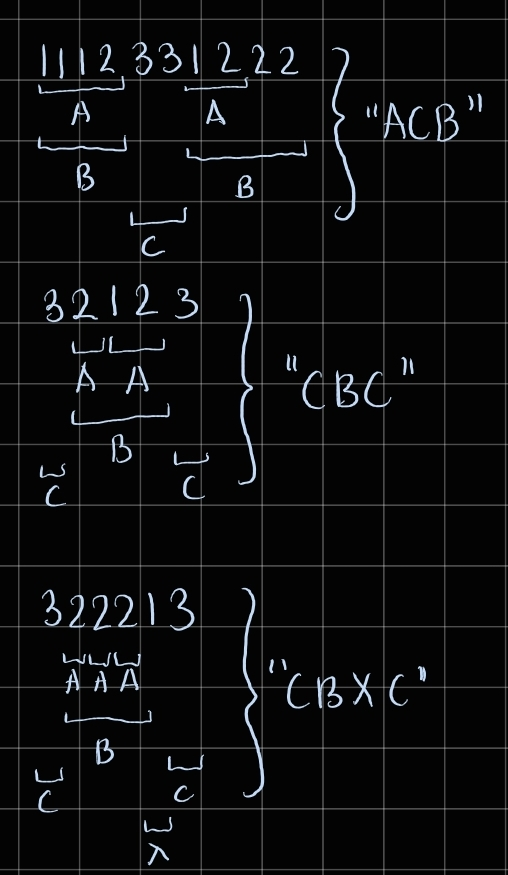
ACB

**b) ii)**

CBC

**b) iii)**

CBXC



**c) i)**

**Straight from F:**

word

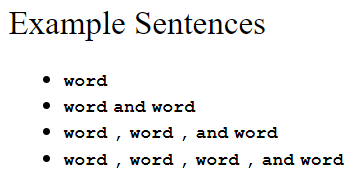
word , word

word and word

word , word , word

word , word , and word

**Using S as starting symbol:**



**c) ii)**

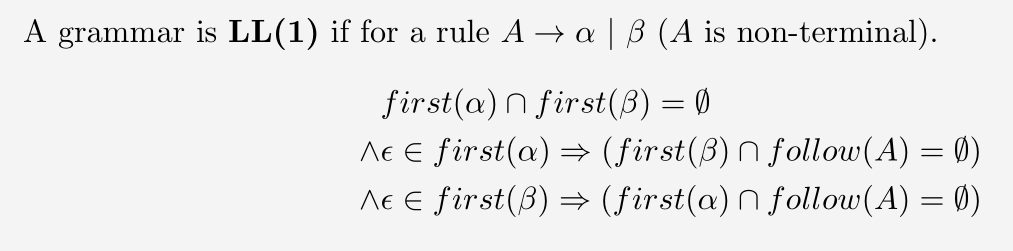
First(L) = { word }

First(F) = { word }

both first sets intersect so the grammar is therefore not LL(1).

-------------------

By the definition of LL(1):

kl,

First(word), First(word and word), First(F), First(F , word , and word) all contain word and so there is a first set clash between any two pairs of alternatives. (Question asks for **all** cases)

Also the rule for F contains left recursion which is not allowed in an LL(1) grammar

**c) iii)**

L -> word [and word | , {word , } and word]

F -> word {, word}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

I managed to compress this down to : L -> word [{, word} and word]

(but there should be an extra ',' before and?)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

L -> word [[,{word,}] and word]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

L -> word LTail

LTail -> ε | and word | , word { , word } , and word

But I haven’t verified that this is definitely LL(1)

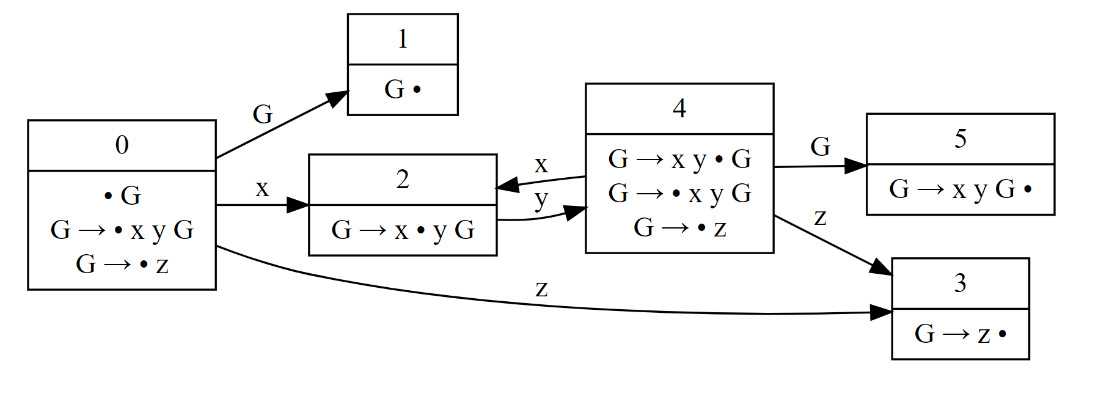
**d) i)**

G -> x

G -> y z G

(x y and z can be swapped around in any order)

**d) ii)**

2z

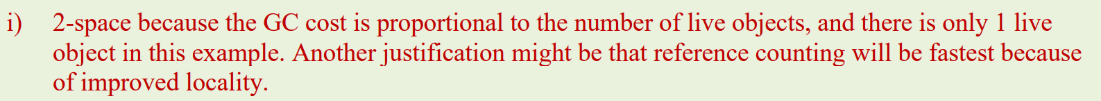
*[Graph uses G -> x y G and G -> z]*

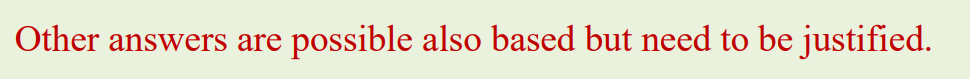
(x y and z can be swapped around consistently)

**e) i)**

A two-space garbage collector would give the fastest execution time. This is because both the reference counting garbage collector and the mark-sweep garbage collector would need to acknowledge each Num n object that gets created and free its memory whereas the two-space garbage collector wouldn't have any work to do apart from just swap the From and To space so that all the unreferenced Num n objects are in the garbage space.

Naranker’s solution:





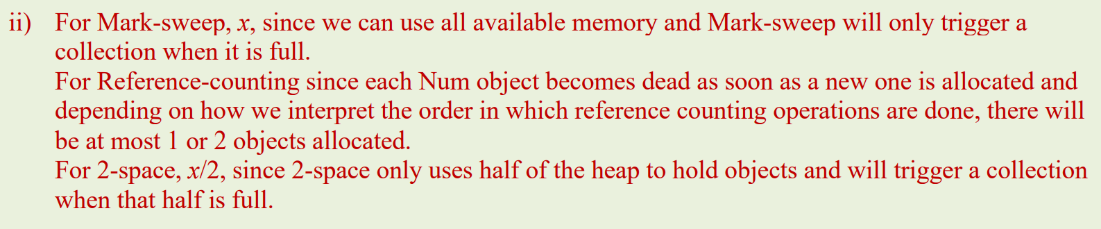
**e) ii)**

The two space garbage collector would allocate x / 2 Num objects as it would split the memory in 2 and allocate until the current half is full at which point it swaps the From and Two space.

The reference counter would allocate just 1 object at any one time as once the for loop goes to the next iteration, the reference count for n is decremented by 1 becoming 0 so the Num object that was created can then be freed.

The mark sweep garbage collector would be activated every once in a while, so would allow the allocation of a fraction of x objects before it is called at which point it frees these.

Naranker’s solution:



**Q2**

**a)**

t1 = d - 1

t2 = e + 1

t3 = t2 \* t1

t4 = b - c

t5 = t4 / t3

a = t5

**b) i)**

i is in liveOut(S9) because it is used again after S9 on line S8 without having been modified.

a and b are also in liveOut(S9) as they are used again after S9 on line S7 without having been modified.

The address of A is in liveOut(S9) as it is used on line S8.

?

**b) ii)**

reachIn(S7) = {S1, S2, S3, S4, S5, S7, S9z

Here some of the reaching definitions to S7 are of the same variable. F or example, S 1 and S9 both reach S7 yet they are both definitions for i. They are both able to reach S7 as S7 is within the loop and S1 "reaches" it when the loop is entered for the first time and then S9 "reaches" it in further iterations.

**b) iii)**

Only S1 and S2

S1 and S2 because they are the only reaching definitions for variables a and b.

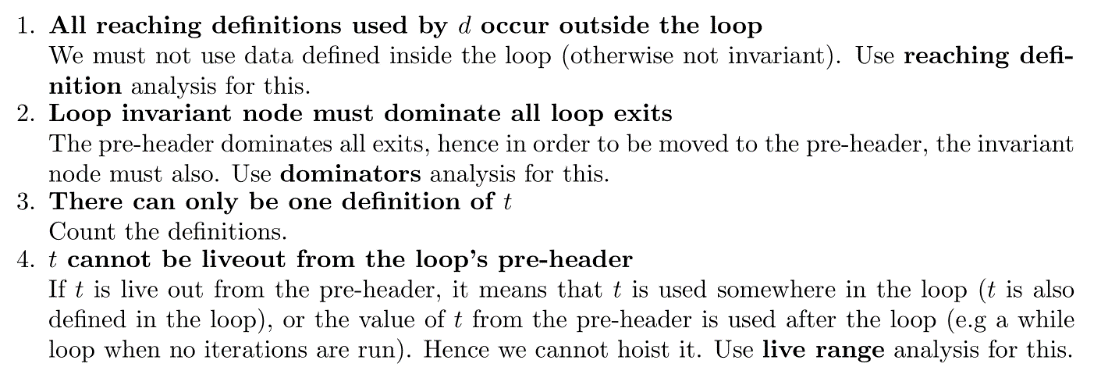
**b) iv)**

S7 is loop invariant because it depends on variables a and b and both a and b are unassigned within the loop.

**b) v)**

No it would not be valid because if the loop has 0 iterations, the function will return a/b when it should instead return 0.

More formally, r is in the liveout set of the pre-header of the loop, which breaks one of the 4 conditions required for loop invariant code motion. (See 4 below)

s

(From Oliver’s notes, copied from the slides)

**b) vi)**

S5 and S7

**b) vii)**

**Answer 1:**

renaming the variables holding r = 0 and r = a /b -> SSA ensures this. r1 = 0, r2 = a /b

now once we've generated the code we notice that the loop invariant node does not dinat om e all loop exits.

We therefore set t3 = phi(r1, r2).

and removing the phi operator into normal code we get:

r1 = 0

r2 = a /b

r3 = r1

while ( i < 10)

{

...

r3 = r2

...

}

return r3

**Answer 2 [Description of hoisting possibly not needed]:**

With SSA, we would have two versions of r defined, and r0 = 0 for S5 and an r2 = a/b for the loop redefinition of r in S7. We could then hoist the r2 out of the loop as it is invariant and replace all mentions of r2 in the loop with r1.

This r1 would be defined to be equal to phi(r0, r2) and would be equal to r0 if entering from the top of the program, and r2 if entering from the bottom of the loop. This single assignment of r1 would be the only definition that reaches S10. This allows for loop invariant code motion while retaining the semantics of the program.