# Paper Title: Programming Languages: Principles and Design

# Unit Code: 6G6Z1110

# Name: Aubrey Harley Monk

# ID: 18026172

## Answers:

## QUESTION 1

1a)

* Higher-Order Functions – A Higher-Order function accepts functions as parameters or can return other functions.
* Pure Functions – Pure Functions have no side-effects meaning they will always return the same value given the same input values.
* Recursion – Recursion is where a function calls itself, it is typically used for iteration as data and variables are immutable so there are no loops for iteration.

1b)

* Encapsulation – Encapsulation groups data and methods together inside a class so that they cannot be accessed directly.
* Inheritance – Inheritance allows classes too inherit properties and methods from parent classes creating a class hierarchy.
* Polymorphism – A polymorphic object is an object that has more than one type assigned to it.

1c)

In the Object-Oriented paradigm code is executed in a certain order however, in the functional paradigm code can be executed in any order. In the functional paradigm functions are considered pure and have no side-effects as they cannot modify variables, they will always return the same value given the same input, whereas in the object-oriented paradigm a function could modify a variable in another part of the code, this would be considered a side-effect. Both paradigms allow higher order functions, a higher order functions is a function which can return other functions or accepts other functions as parameters for example you could call a print function giving another function as its parameter, this would output the return value of that function to the console. In the functional paradigm computation is done without modifying state meaning that variables in this paradigm are immutable and cannot be changed once created, for example if you defined a variable like ‘let a = 1’ then tried to change the value like ‘a = 3’ this would not be possible in the functional paradigm. However, in the Object-oriented paradigm state can be changed, meaning variables are mutable and can be modified after they are created. Iteration in the functional paradigm is not possible using loops like it is in the object-oriented paradigm as a loop must modify variables to meet a certain condition for example a counter could be incremented during each iteration of the loop until it reaches a certain value. However, in the functional paradigm recursion is used for iteration. Recursion is where a function can call itself typically until it reaches a base case. If you wanted to iterate over a list in the object-oriented paradigm you could use a for loop and in the functional paradigm you could use recursion where the function call itself each time using the tail of the list as a parameter. In the object-oriented paradigm data is abstracted into and object containing properties and methods, encapsulation allows for these properties and methods to be grouped together inside a class preventing them from being directly accessed. Also, there is Inheritance in the object-oriented paradigm meaning that classes can inherit properties and methods from parent classes for example a Car class could inherit properties and methods from a vehicle class. Objects can be polymorphic this means that they can have more than one type assigned to them. Within the functional paradigm data cannot be abstracted into objects, instead functions and variables are used.

## 1d)

One feature of the language is variable typing meaning variables are declared as a specific type before they are used, here is an example in the code: “int total = 0”, you can see that total is defined as an Integer.

Iteration in the form of a ‘while’ loop is also a feature in this language: ‘while (true){…}’ loops until a value is returned from the ‘isHappy()’ method and ‘while (n > 0){…}’ loops while n is greater than 0.

Another feature is mutable variables, as you can see here in the code: “total += (n % 10) ^^ 2;” that the variable ‘total’ is modified within the while loop meaning that it is indeed mutable.

The ‘Array’ data structure is a feature of this language, the var ‘past’ is an example of this: “int[int] past” which in this case is an associative array where the data and the index is of type integer.

Another feature of this language is the ability to declare/create functions, in this case there is an ‘isHappy()’ function declared as so: ‘bool isHappy(int n) pure nothrow{…}’ Boolean is the return type of this function and it has an integer as the only parameter.

If statements are also a feature of this language, here are some examples:

“if (total == 1)

return true;”

and

“if (total in past)

Return false;”

In this case the function ‘isHappy()’ returns true if total is equal to 1 and returns false if total is in past.

This language seems to also have the ability to handle exceptions due to the use of the ‘nothrow’ keyword in the function declaration.

This language also has a built-in print function: ‘writeln’.

The last feature I am going too discuss is the ability to import and use standard library functions, the import is seen here within the main method: ‘import std.stdio, ……’ and are used in: “int.max.filter!isHappy.take(8).writeln;”.

## 1e)

For this task I would choose the ‘Object-Oriented Paradigm’ (OOP) using the ‘Python’ language. The main reason for choosing OOP is that it allows data abstraction which will be very important for this role as you would be able to abstract data from large data sets into the form of objects with properties and methods, a good benefit of this is that the properties and methods would be encapsulated within the class so they cannot be changed in other parts of the code except from calling class methods, this avoids certain data from being interfered with by other parts of the program reducing side-effects. Another benefit is re-usable code, classes could be re-used in any of the “short applications” that are being written. For data modelling you would be able to create models using objects that represent certain things or data. Inheritance is another big benefit of using OOP for example if you are doing data processing you could crate generic data classes and then create subclasses for specific types of data that could inherit the properties and methods from the generic data class. For data visualization you can create methods in the data classes that return trees or graphs, external library functions could even be used to aid with this. Polymorphism is also another key feature as the classes written for multiple types of data, this means that you could re-use your code for different data sets with different datatypes. Some downfalls of OOP are that it is not the most efficient paradigm for analyzing data as it is high-level and could be slow with really large data sets, also since variables are mutable there could be lots of side-effects caused by functions in the code that could mess with the data in other parts of the code. I chose Python as the language as it has the largest choice of data-science library’s in comparison to other languages and it also has many useful inbuilt data visualization functions.

## QUESTION 2

## 2a)

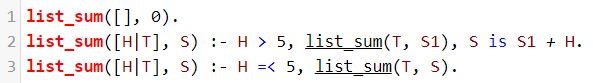
Raw Code:

list\_sum([], 0).

list\_sum([H|T], S) :- H > 5, list\_sum(T, S1), S is S1 + H.

list\_sum([H|T], S) :- H =< 5, list\_sum(T, S).

Screenshots:





## 2b)

Procedural solution in C:

Raw code:

#include <stdio.h>

#include <string.h>

int main(void) {

char input[] = "";

int letters = 0;

int numbers = 0;

printf("Please enter a string of letters or numbers: \n");

scanf("%s", input);

int i;

for (i = 0; i < strlen(input); i++){

if (input[i] >= 'a' && input[i] <= 'z') {

letters++;

}else if (input[i] >= '0' && input[i] <= '9') {

numbers++;

}

}

if(letters > numbers){

printf("letters");

}else if (numbers > letters){

printf("numbers");

}else if (letters == numbers){

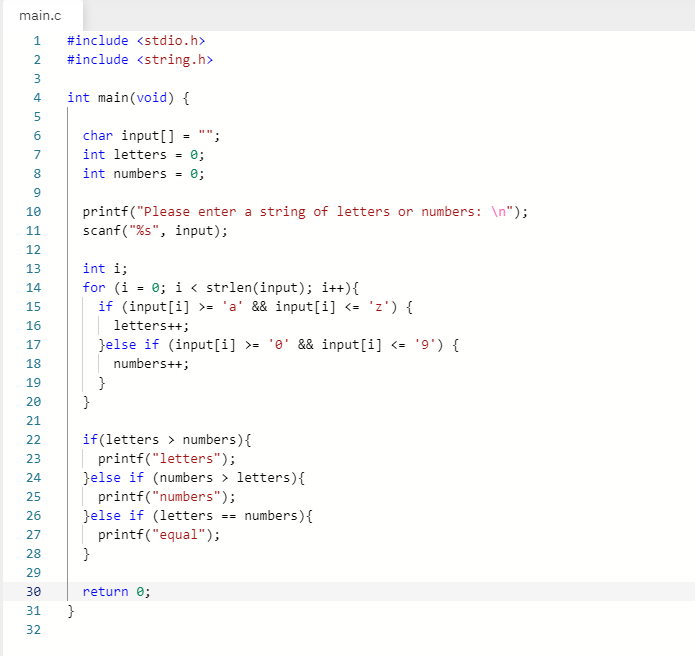
printf("equal");

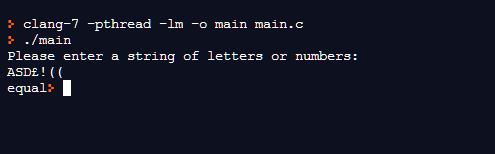
}

return 0;

}

Commented code screenshot:





Functional solution in Haskell:

Raw code:

import Data.Char

lettersOrNumbers :: [Char] -> [Char]

lettersOrNumbers input

| countLetters input > countNumbers input = "letters"

| countNumbers input > countLetters input = "numbers"

| otherwise = "equal"

countLetters :: [Char] -> Int

countLetters s = length (filter (isLower) s)

countNumbers :: [Char] -> Int

countNumbers s = length (filter (isNumber) s)

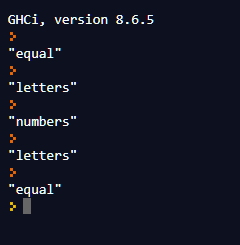
main :: IO ()

main = do

print(lettersOrNumbers "sdf12345")

Commented code screenshot:



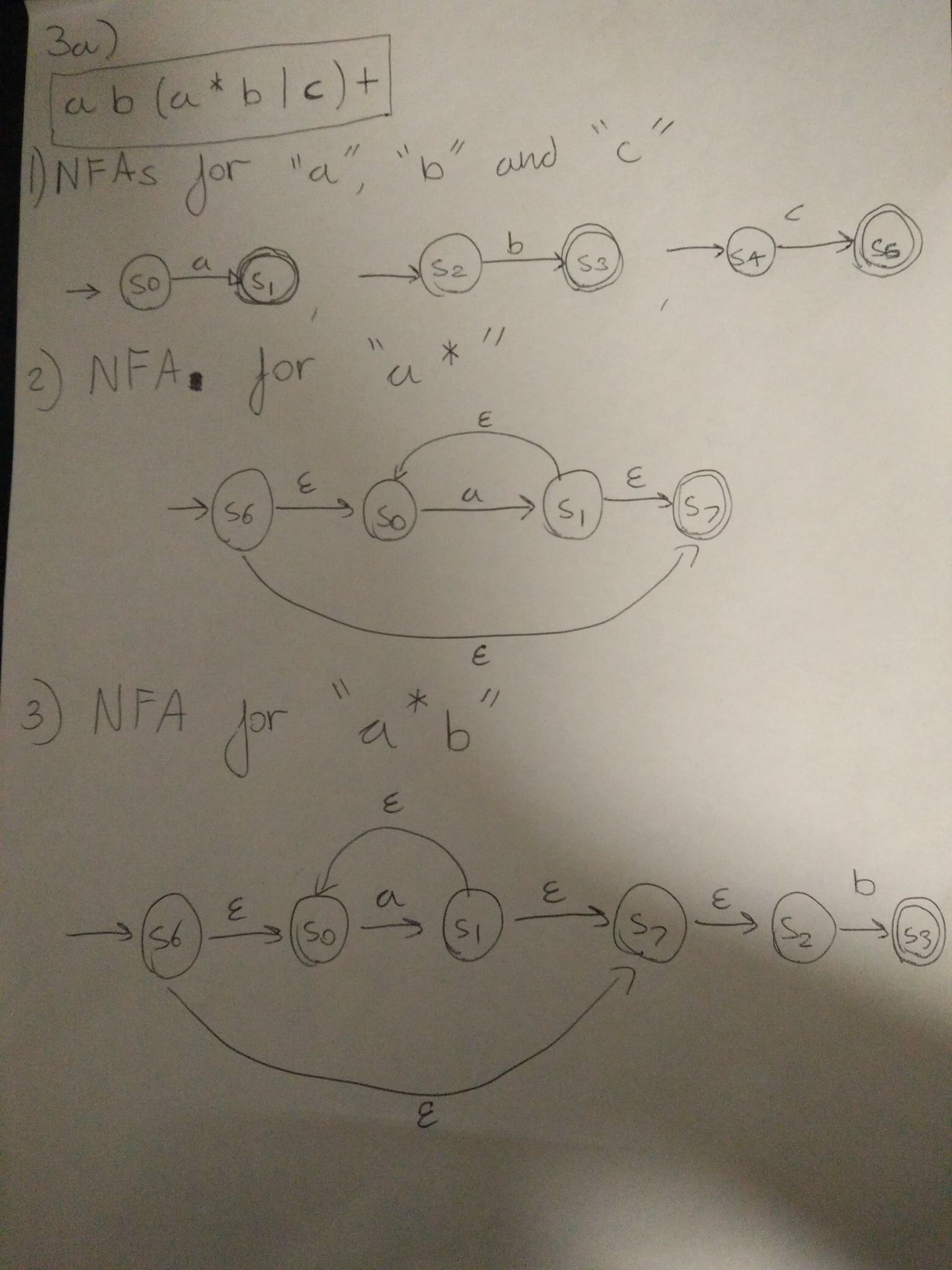


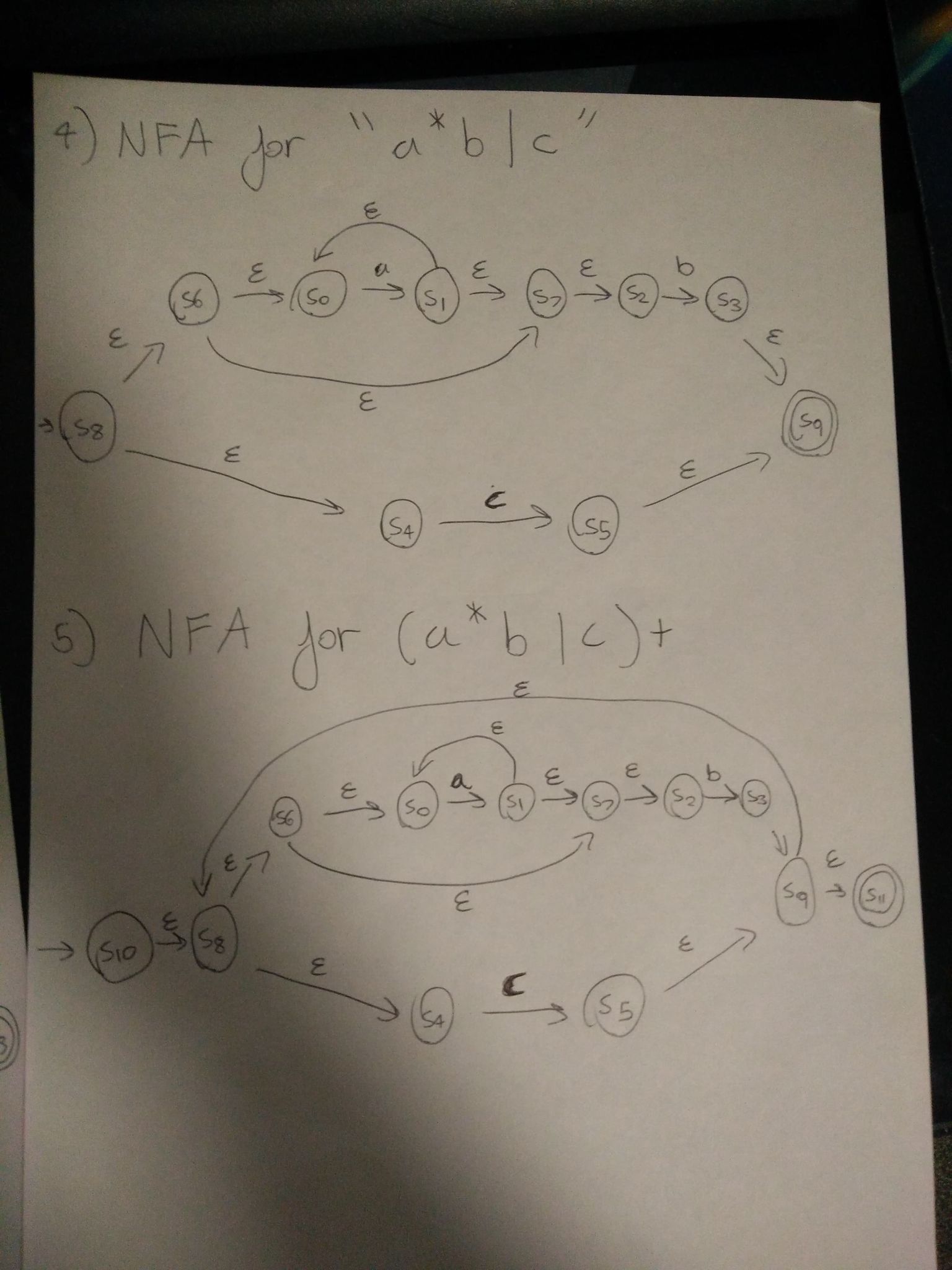
## 2c)

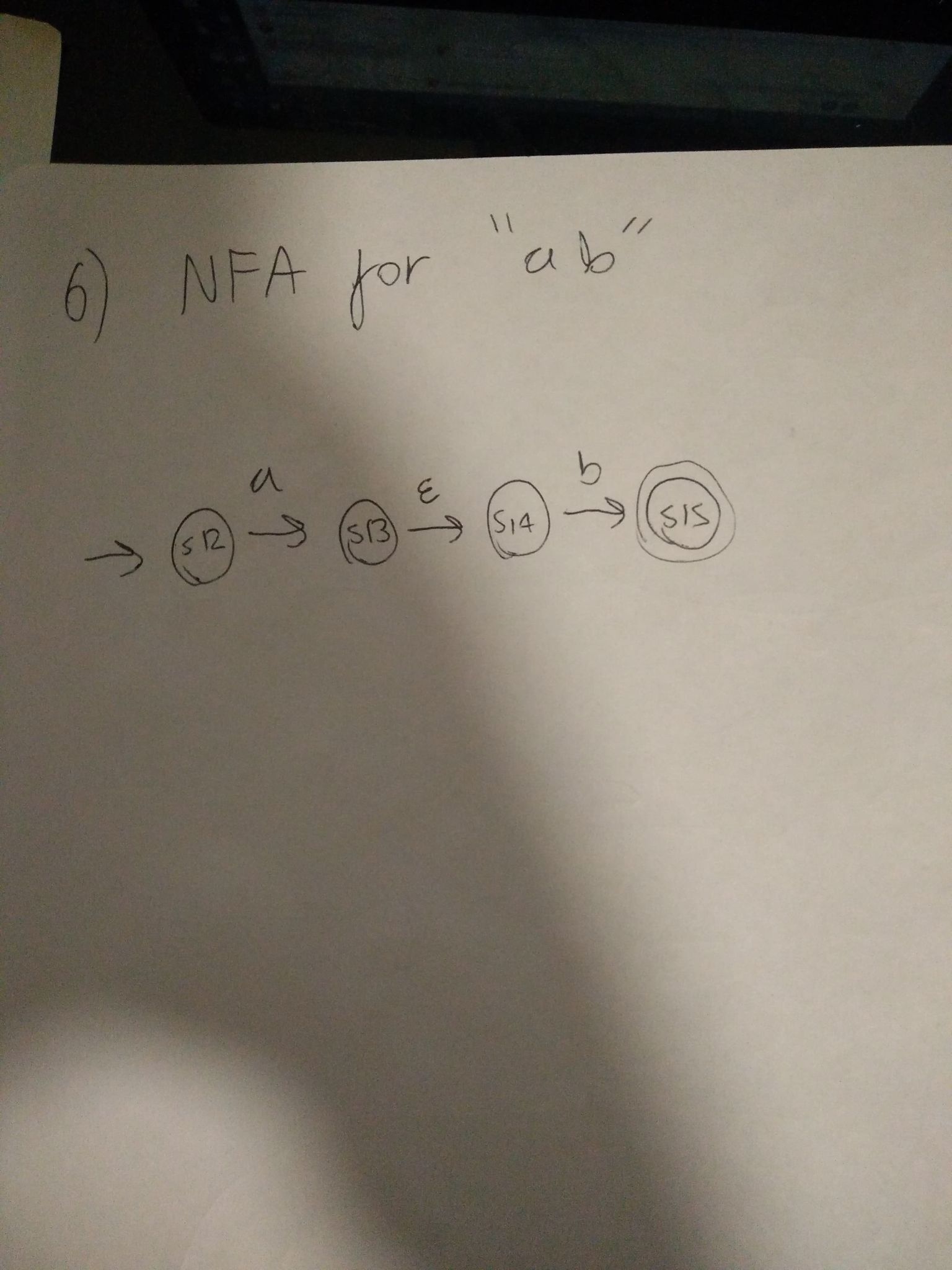
In C to store the counters I used variables of type int (one for letters and one for numbers) and I used a variable of type char[] (char array) to store the input string. However, in Haskell I did not use variables to store anything I just used functions with different return types. To count the amount of letters and numbers in C I used an If/Else If statement within a For Loop, the for Loop iterates over each char in the input string, it does this by iterating from 0 to the length of the input string using ‘i’ as a counter and incrementing this each time until ‘i’ is equal to the length of the input, to get the specific char I just indexed the input within the loop: ‘input[i]’. To get the length of the input I used the ‘strlen’ function. The If/Else If statement that is used inside the loop, increments the letters counter if the char is within a-z or it increments the numbers counter if the char is within 0-9. In Haskell to count the amount of letters or numbers I could’ve chosen to use an iterative approach using recursion but instead I chose to use the ‘filter’ and ‘length’ functions as it resulted in cleaner and more readable code, I created 2 functions ‘countLetters’ and ‘countNumbers’, one for counting letters and one for counting numbers in the input string, for each, the input string is given as the parameter, the filter function then returns a list of either all the lowercase letters in the input string or it returns a list of all the numbers in the input string, the length function is then used to return the length of this list, this length is then returned by the count functions. In C for comparing the number/letter count variables I used another If/Else If statement this works by using the ‘>’, ‘<’ and ‘==’ operators to compare the count variables, depending on the comparison the program will output either: “letters”, ”numbers” or “equal”. However, In Haskell I created a separate function ‘letterOrNumbers’ to do this, this function uses a guard instead of an if statement, the guard is used instead again to produce more readable and cleaner code, although they work in a similar way, again using the ‘>’, ‘<’ and ‘==’ operators. The function takes the input string as the parameter and returns the output string (type char[]), when comparing the count it calls either the ‘countLetters’ and compares it with the ‘countNumbers’ function and vice versa, then depending on the result it either returns “letters”, ”numbers” or “equal”. To output the result the C program uses the printf function within the count comparison with the appropriate hard-coded string as a parameter, however the Haskell program uses the print function with the ‘letterOrNumbers’ function as its parameter (making it a Higher-Order Function) and the input string as the parameter for the ‘letterOrNumbers’ function. Another difference in C is that I used the ‘scanf’ function to get the user input for the input string whereas in Haskell the input string is hard-coded in. In both solutions characters that are not a number or letter are ignored as they are no included in the comparisons so the counters are not increased.

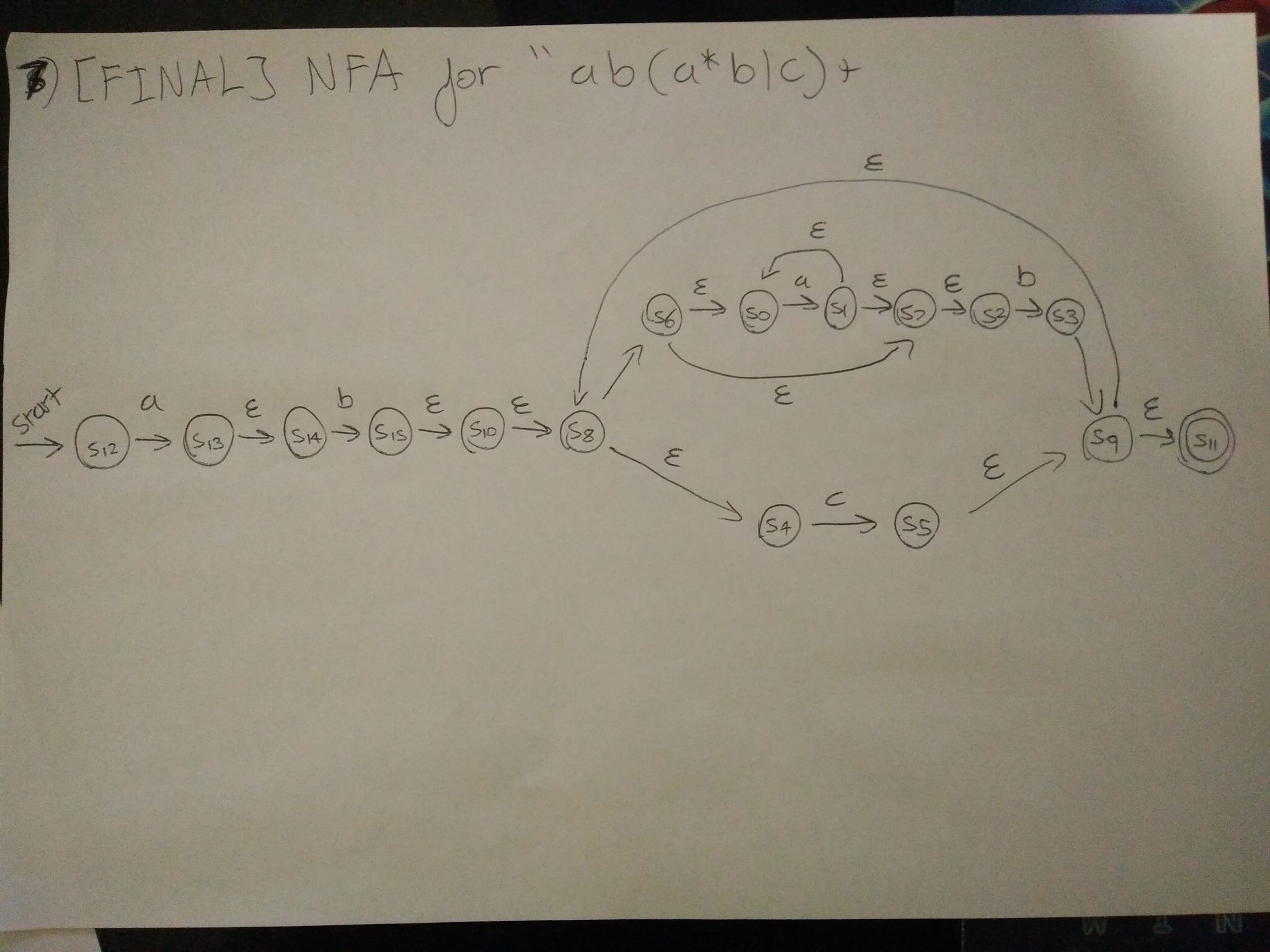
## QUESTION 3

## 3a)

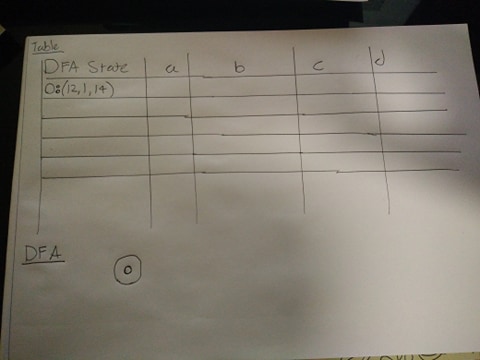


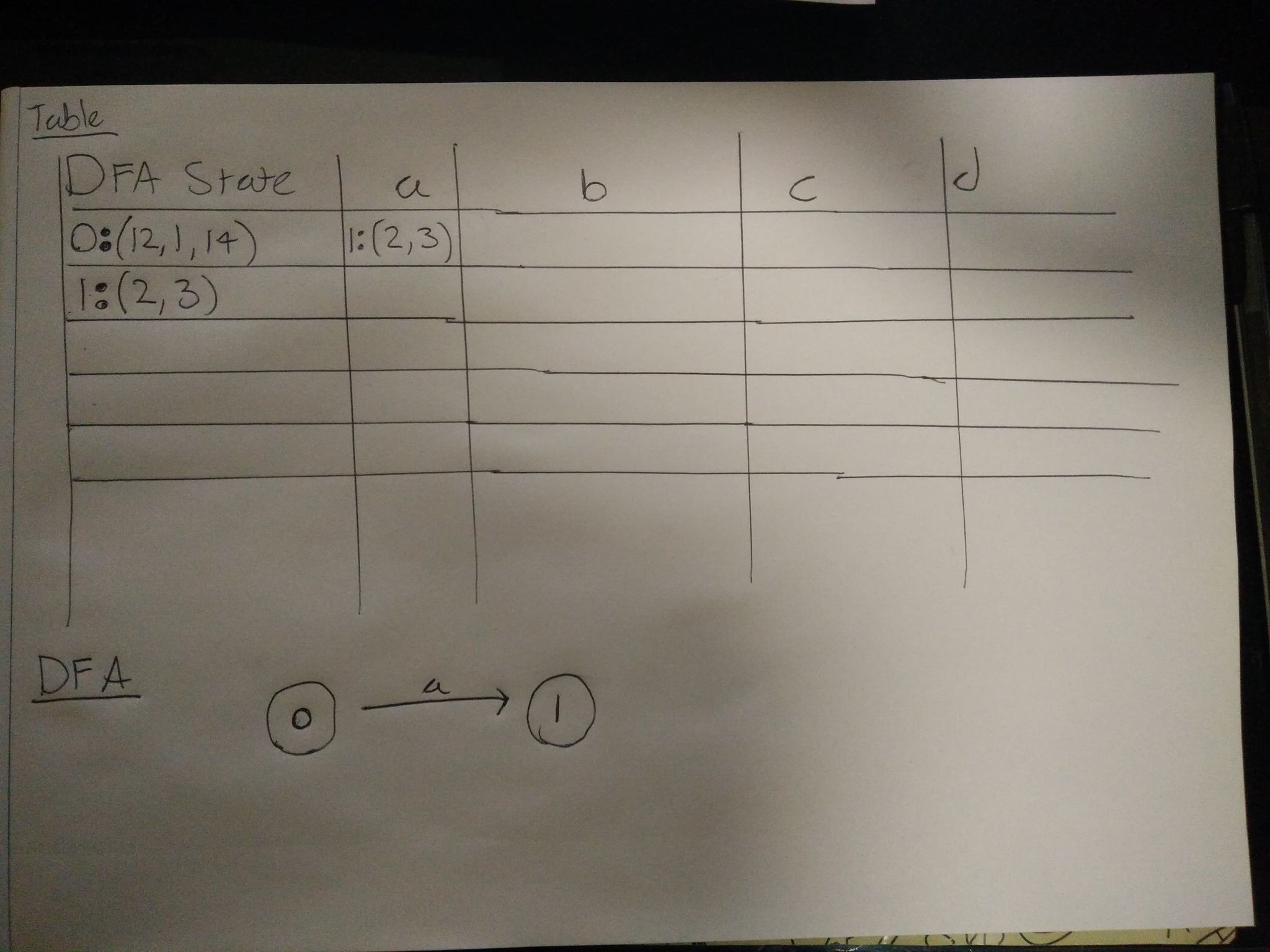


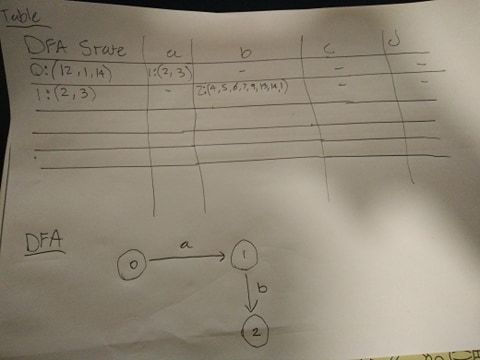


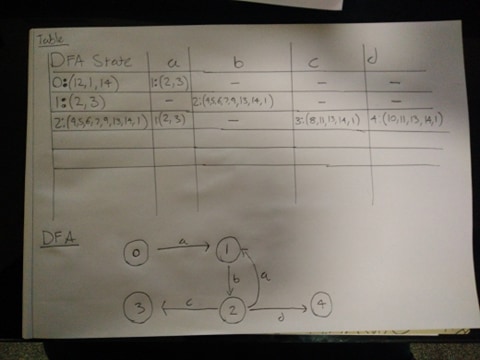


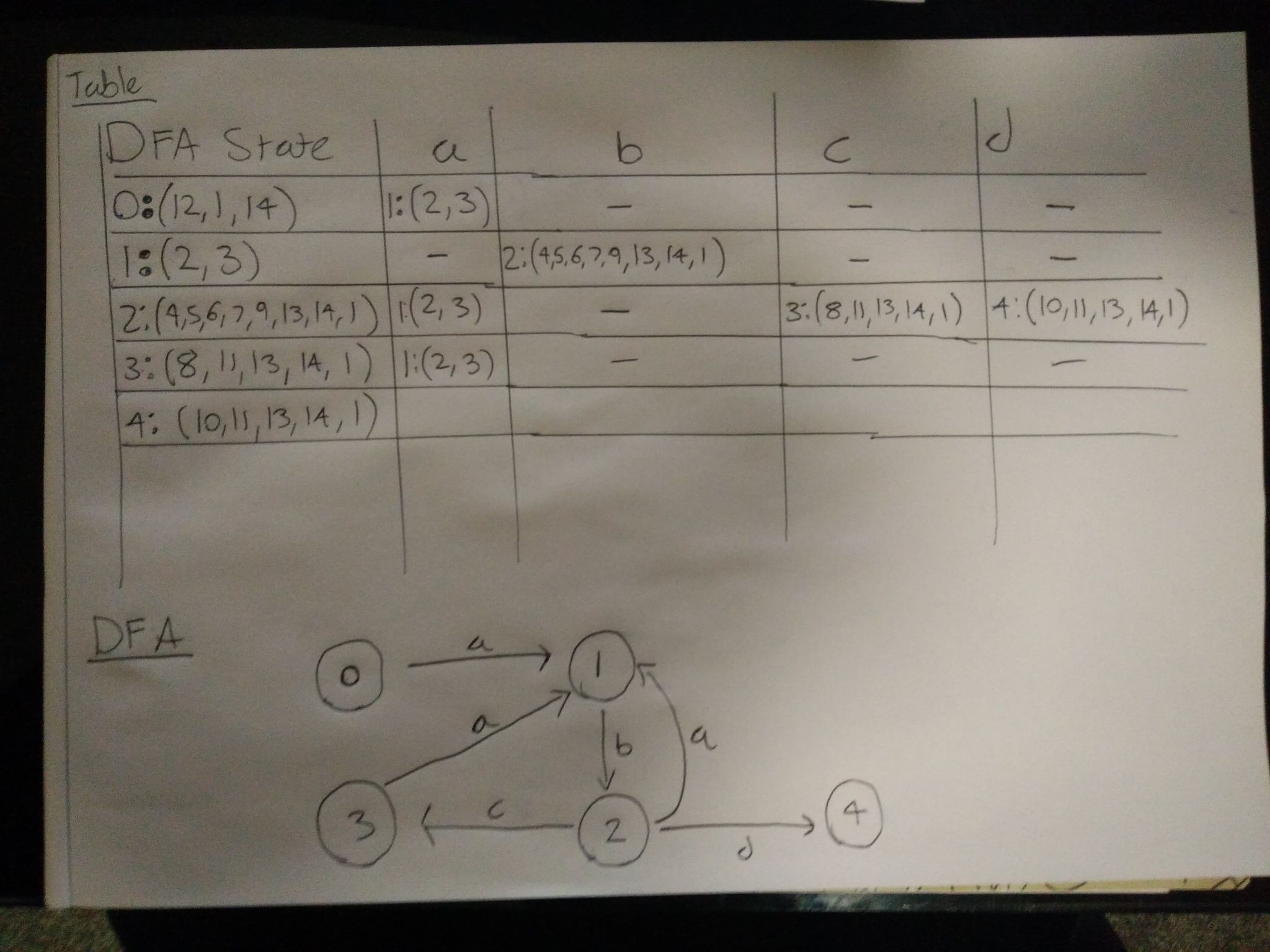
## 3b)

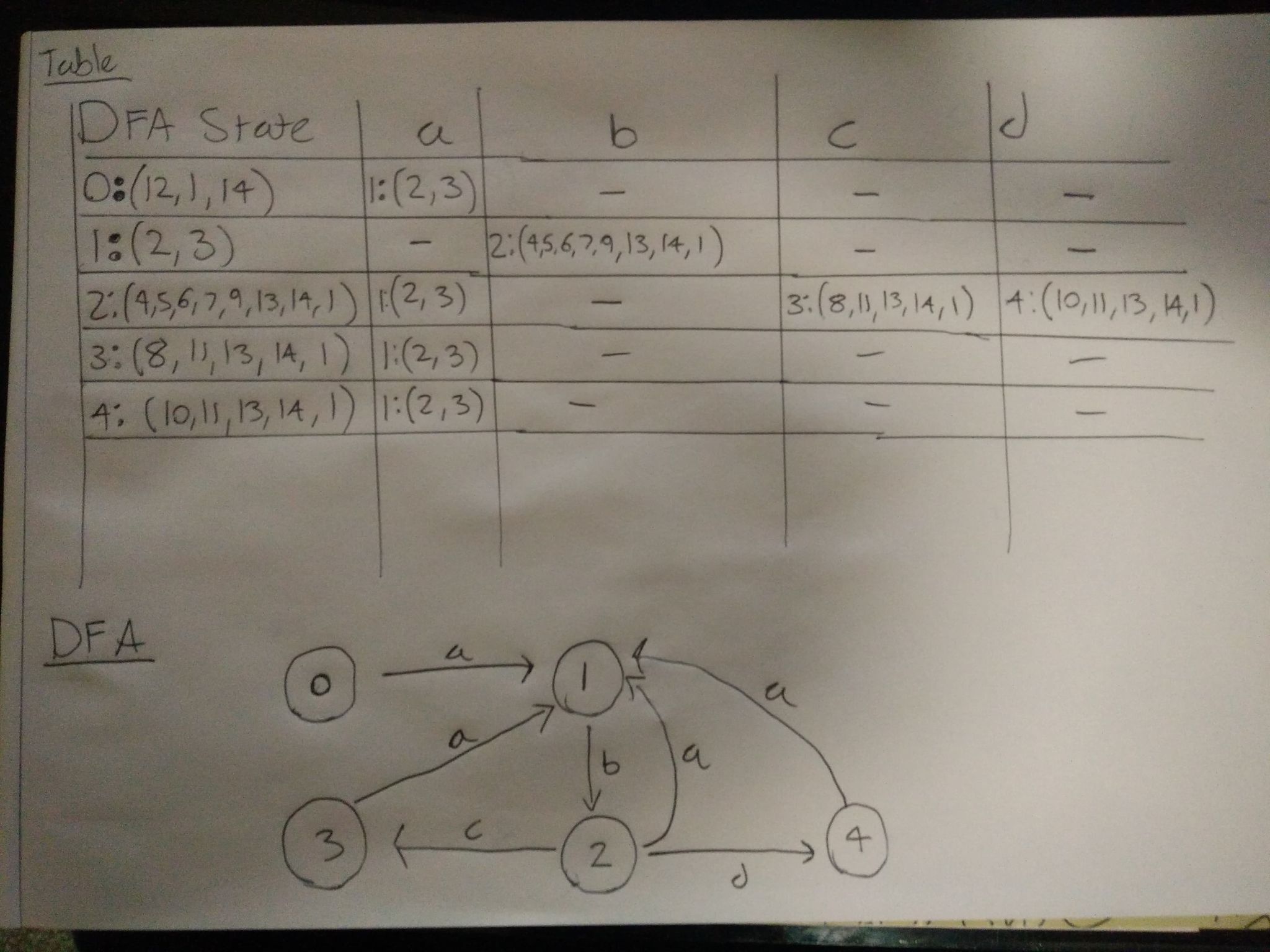


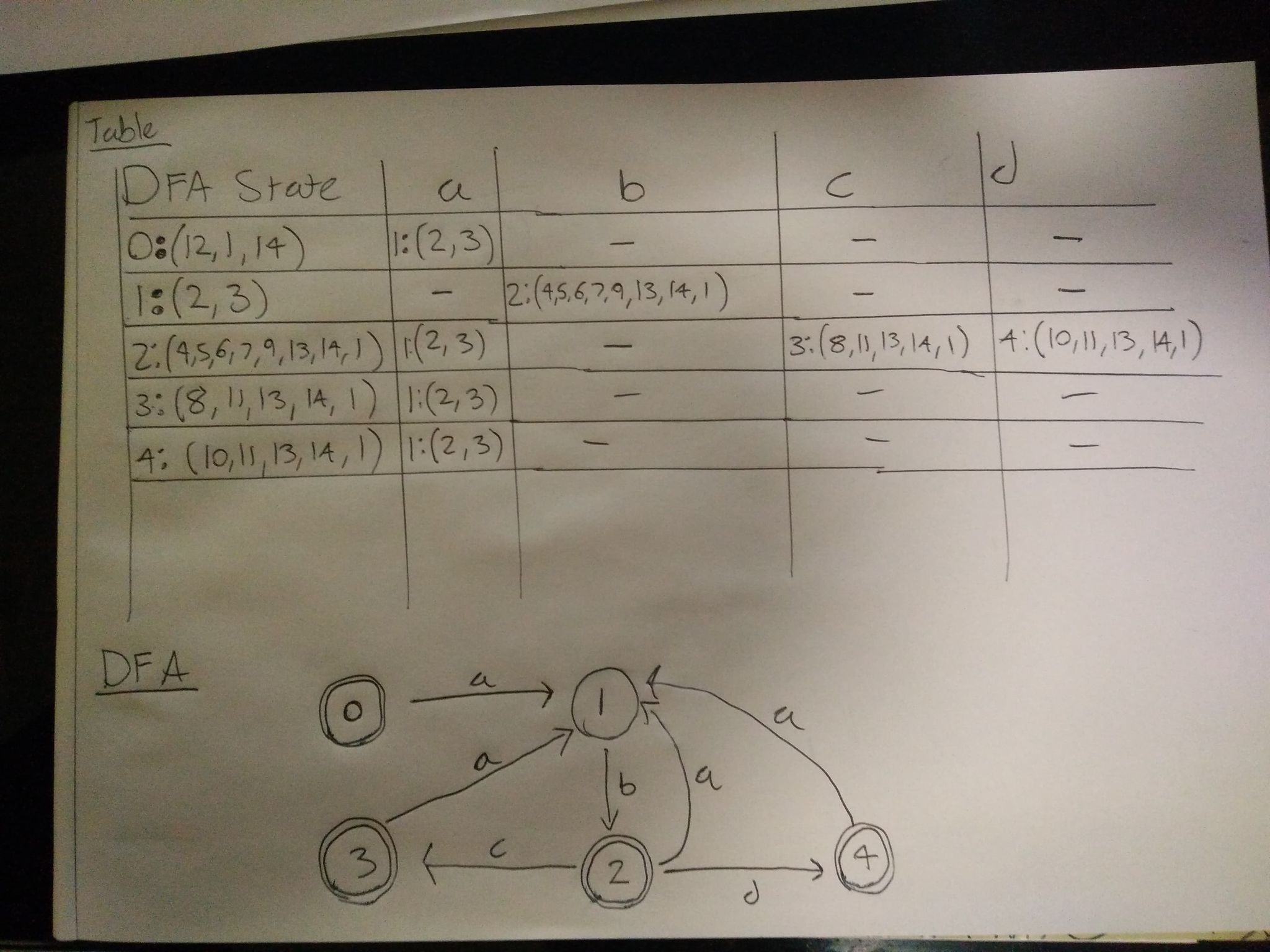


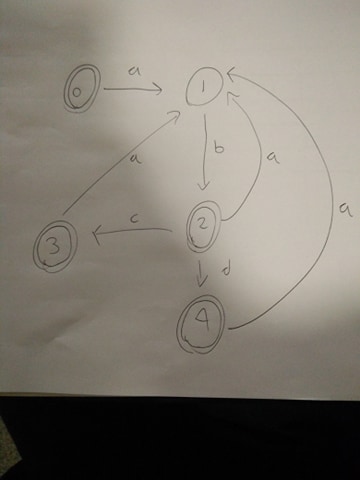






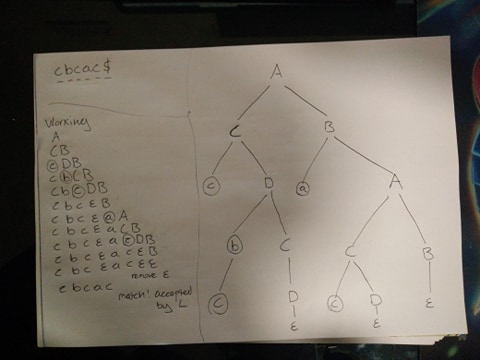




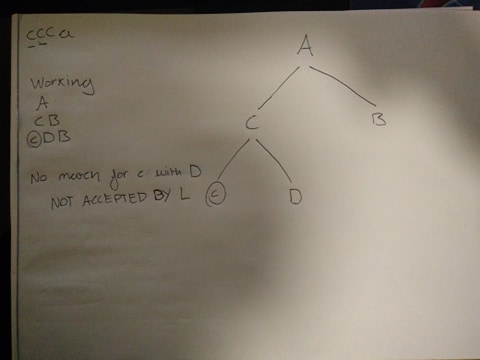


## 3c)

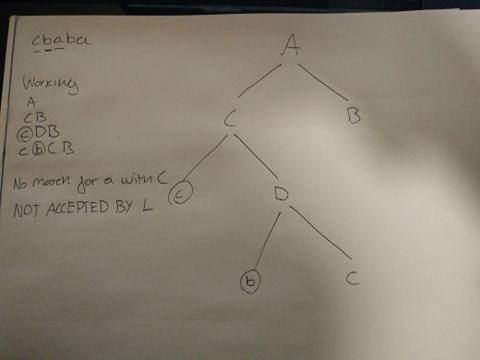
3ca)



3cb)



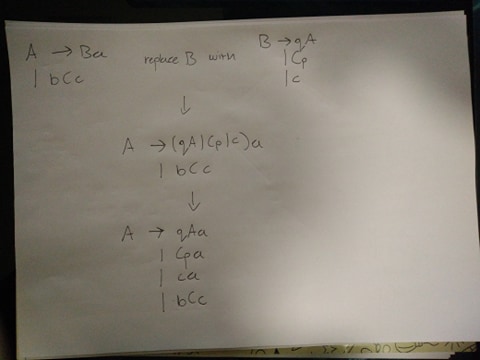
3cc)

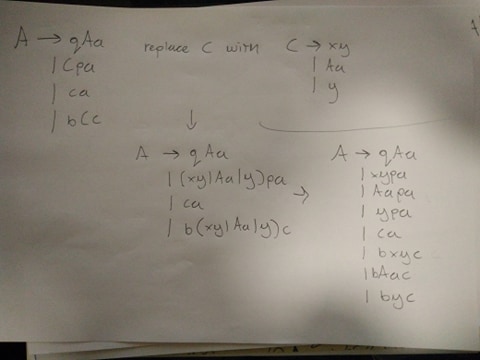


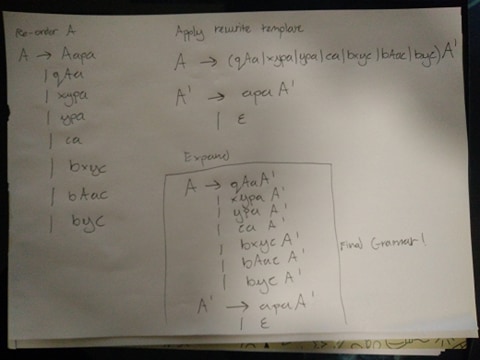
## 3d)

Problem: Indirect left-recursion, to solve this: rewrite grammar first, then apply re-write template.

Solution:







## 4a)

Raw code:

grammar question4;

/\*

LEXER RULES

\*/

/\* Define terminal symbols as tokens \*/

TRUE : 'true';

FALSE : 'false';

LCURLY : '{';

RCURLY : '}';

COLON : ':';

COMMA : ',';

DOUBLEQUOTE : '"';

fragment DIGIT : [0-9]; /\* fragment not counted as token, used to simplify grammar \*/

fragment ALPHA : [a-z]; /\* fragment not counted as token, used to simplify grammar \*/

INTEGER : DIGIT DIGIT\*; /\* INTEGER defined as one or more DIGITS \*/

ALPHA\_STRING : DOUBLEQUOTE ALPHA\* DOUBLEQUOTE; /\* ALPHA\_STRING defined as zero or mor ALPHAs surrounded by quotes \*/

/\*

PARSER RULES

\*/

dict : LCURLY key COLON value (COMMA key COLON value)\* RCURLY; /\*DICT defined as one or more key:value pairs seperated by commas and all surrounded by curly braces \*/

key : ALPHA\_STRING;

value : literal;

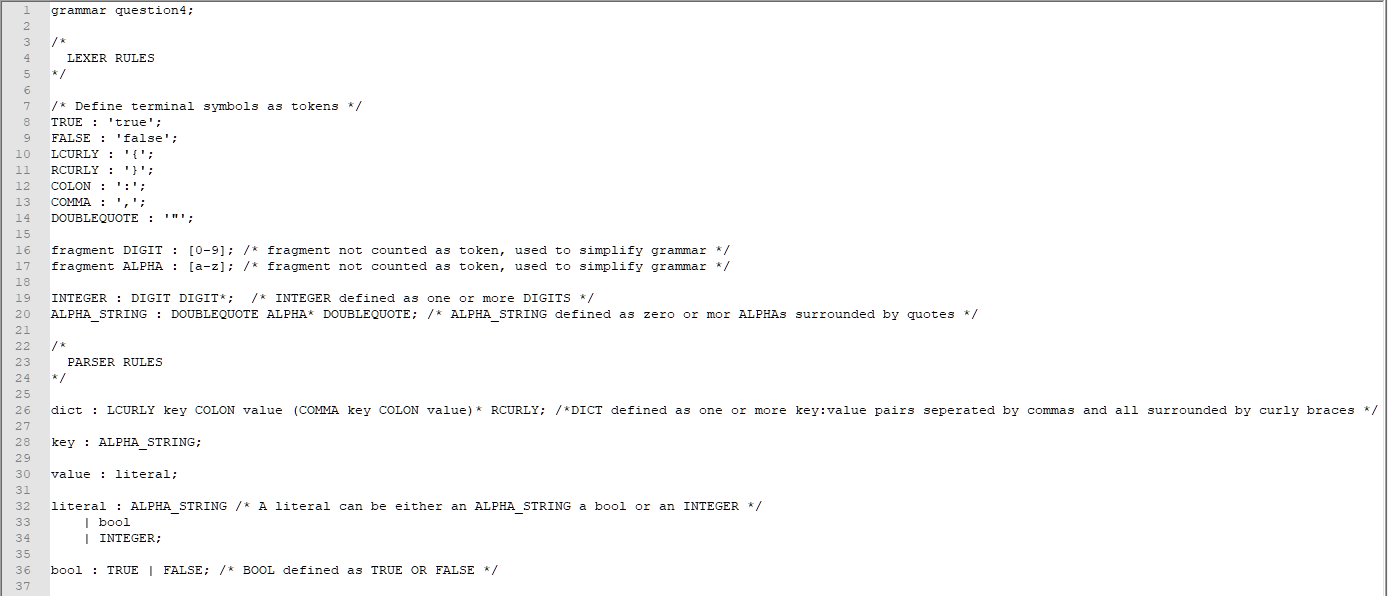
literal : ALPHA\_STRING /\* A literal can be either an ALPHA\_STRING a bool or an INTEGER \*/

| bool

| INTEGER;

bool : TRUE | FALSE; /\* BOOL defined as TRUE OR FALSE \*/

Screenshot of code:



## 4b)

For this solution I used the Decaf.g4 grammar and the Symbol Table provided from moodle.

Raw code:

import antlr4 as ant

from DecafLexer import DecafLexer

from DecafParser import DecafParser

from DecafVisitor import DecafVisitor

from SymbolTable import HEAP, STACK, SymbolTable, VarSymbol, MethodSymbol

class DecafSemanticChecker(DecafVisitor):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.st = SymbolTable()

def visitProgram(self, ctx:DecafParser.ProgramContext):

self.st.enterScope()

self.visitChildren(ctx)

self.st.exitScope()

def visitMethod\_decl(self, ctx:DecafParser.Method\_declContext):

line\_num = ctx.start.line

method\_name = ctx.ID(0).getText()

return\_type = ctx.return\_type().getText()

method\_params = []

for i in range(len(ctx.data\_type())):

param\_type = ctx.data\_type(i).getText()

param\_name = ctx.ID(i+1).getText()

param\_symbol = VarSymbol(id=param\_name, type=param\_type, line=line\_num, size=8, mem=STACK)

method\_params.append(param\_symbol)

method\_symbol = self.st.probe(method\_name)

method\_symbol = MethodSymbol(id=method\_name,

type=return\_type,

line=line\_num,

params=method\_params)

self.st.addSymbol(method\_symbol)

self.visitChildren(ctx)

def visitMethod\_call(self, ctx:DecafParser.Method\_callContext):

line\_num = ctx.start.line

method\_name = ctx.method\_name().getText()

method\_symbol = self.st.probe(method\_name)

if method\_symbol != None:

if len(method\_symbol.params) != len(ctx.expr()):

print("Trying to call method on line " + str(line\_num) + " with wrong number of parameters. Method expects " + str(len(method\_symbol.params)) + " parameters")

else:

for i in range(len(ctx.expr())):

if method\_symbol.params[i].type == "int" and ctx.expr(i).literal().bool\_literal() != None:

print("Trying to call method on line " + str(line\_num) + " with incorrect parameter type. Method expects type int for parameter " + str(i))

elif method\_symbol.params[i].type == "boolean" and ctx.expr(i).literal().int\_literal() != None:

print("Trying to call method on line " + str(line\_num) + " with incorrect parameter type. Method expects type bool for parameter " + str(i))

self.visitChildren(ctx)

filein = open('test.dcf', 'r')

lexer = DecafLexer(ant.InputStream(filein.read()))

stream = ant.CommonTokenStream(lexer)

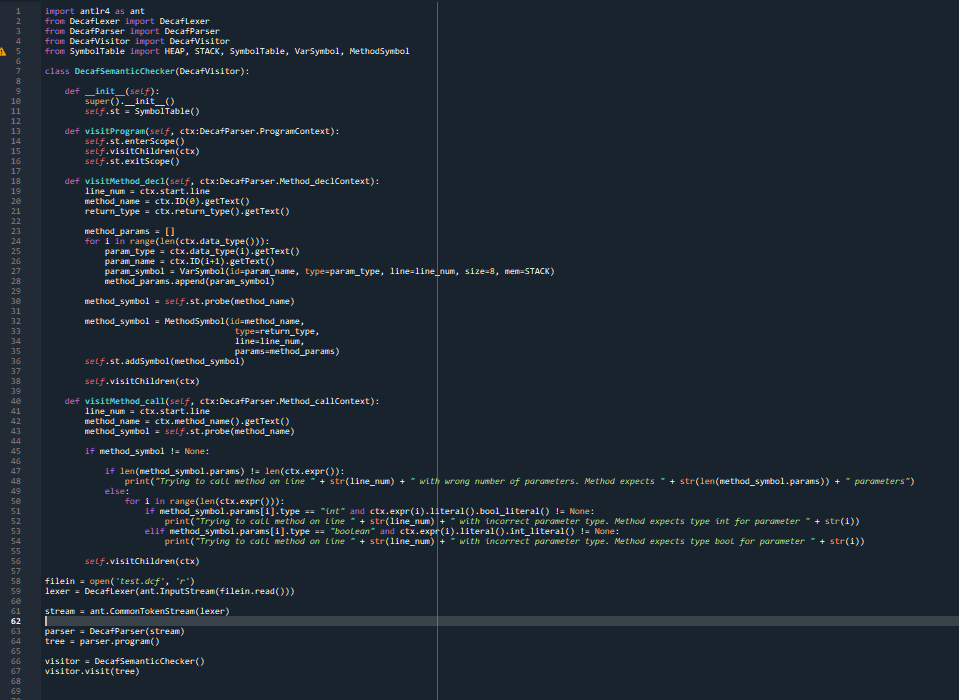
parser = DecafParser(stream)

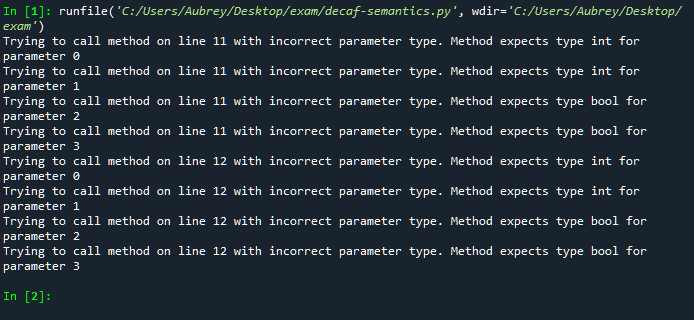
tree = parser.program()

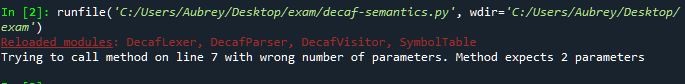
visitor = DecafSemanticChecker()

visitor.visit(tree)

Code screenshots:







## 4c)

The main inefficiencies I have found in this assembly code are on line 6-8, lines 9-11 and lines 13 – 15.

On lines 6-8 the value $1 is moved into the %rax register, then from the %rax register to %r11 register, then from the %r11 register into the memory, this means the program is taking extra steps to achieve the same result as just directly moving it from the %rax register into the memory. On lines 9-11 a similar thing is happening, the value $2 is moved into the %rax register, then from the %rax register to %r11 register, then from the %r11 register into the memory, this means the program is taking extra steps to achieve the same result as just directly moving $2 from the %rax register into the memory. You could even argue that the %rax register is not needed either in both these cases and you could just move the value straight into memory, however I believe assigning an integer is treated as an expression as the integer is being evaluated also the result of the expression is treated as a return value and the %rax is the function return value register, so moving it straight to memory may cause problems in larger/more complex programs.

On lines 13 to 15 the value in -8(%rsp) is being moved to the %rax register, then from the %rax register to the %r10 register, then from the %r10 register back into the memory (at -32%rsp) this is unnecessary as the value (which would be $1) stored at -8(%rsp) in memory so it is unnecessary to store it in memory again as it is wasting memory space and using extra CPU time.

For smaller programs like the one given these steps are unnecessary but however the compiler may be generating the assembly this way to avoid problems in bigger more complex programs. Why I believe the %r10 and %r11 registers are being used in this unnecessary way is if the values needed to be used in an operation following the assignment or needed to be used in other parts of the code it would be faster to access them from the CPU registers than to access them from memory, this is why the compiler is generating the code like this and storing the values in %r10, and %r11. However, they still need to be stored in memory as the %r10 and %r11 registers may be overwritten as there is only a limited amount of CPU registers available.