1. a) /(?=.\*[A-Z].\*[A-Z])(?=.\*[0-9].\*).{3,}$/

This will ensure there are at least two upper case letters, one digit and the minimum length of the string is three characters.

b) /([0-9]\*)\.([0-9][0-9]\*)E([0-9][0-9]\*)/

There should be zero or more digits before the dot. One or more digits after dot and before E. There should be one or more digits after E.

c) /(^[A-Za-z])([^\@\#\%\^\&\\*\(\)\$]){0,9}$/

The ^ indicates the first character which should only be character. The next condition states that there should not be any special characters. Total length of string should not exceed 10.

1. CFG for given program:

*completeprogram -> program identifier declrlist begin stmtlist end identifier;*

*declrlist -> declr declrlist | Є*

*declr -> funclist | proclist | variablist | Є*

*funclist -> func funclist | Є*

*proclist -> proc proclist | Є*

*variablist -> variab variablist | Є*

*stmtlist -> stmt stmtlist | Є*

*func -> function identifier (argumlist) variablist begin stmtlist end identifier;*

*argumlist -> var identifier argumlisttail | Є*

*argumlisttail -> , var identifier argumlisttail*

*proc -> procedure identifier() variablist begin stmtlist end identifier;*

*variab -> var identifier;*

*stmt -> identifier stmttail; | return expression; | funccall;*

*stmttail -> := expression*

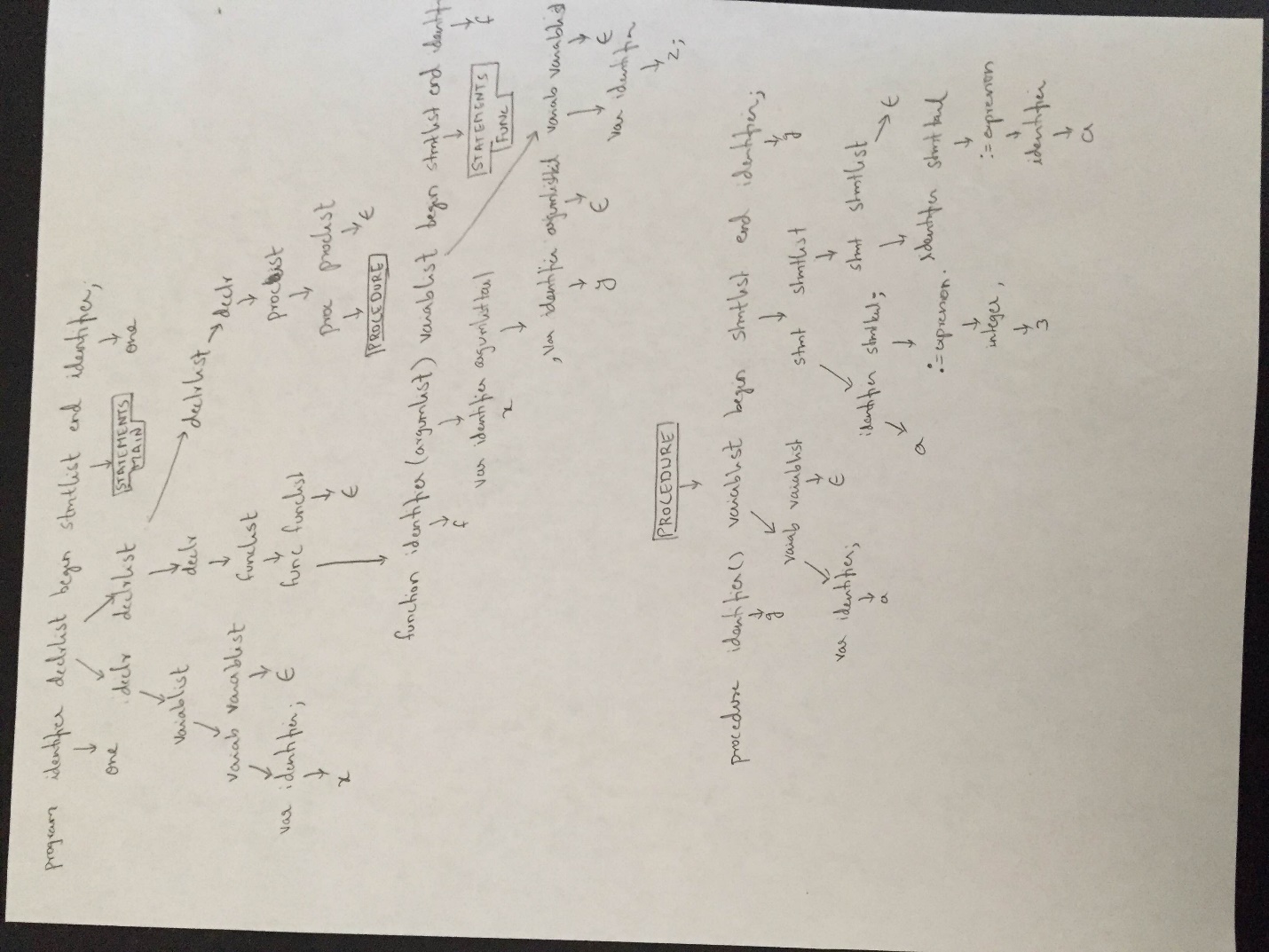
*expression -> (expression) | expression operation expression | integer | identifier*

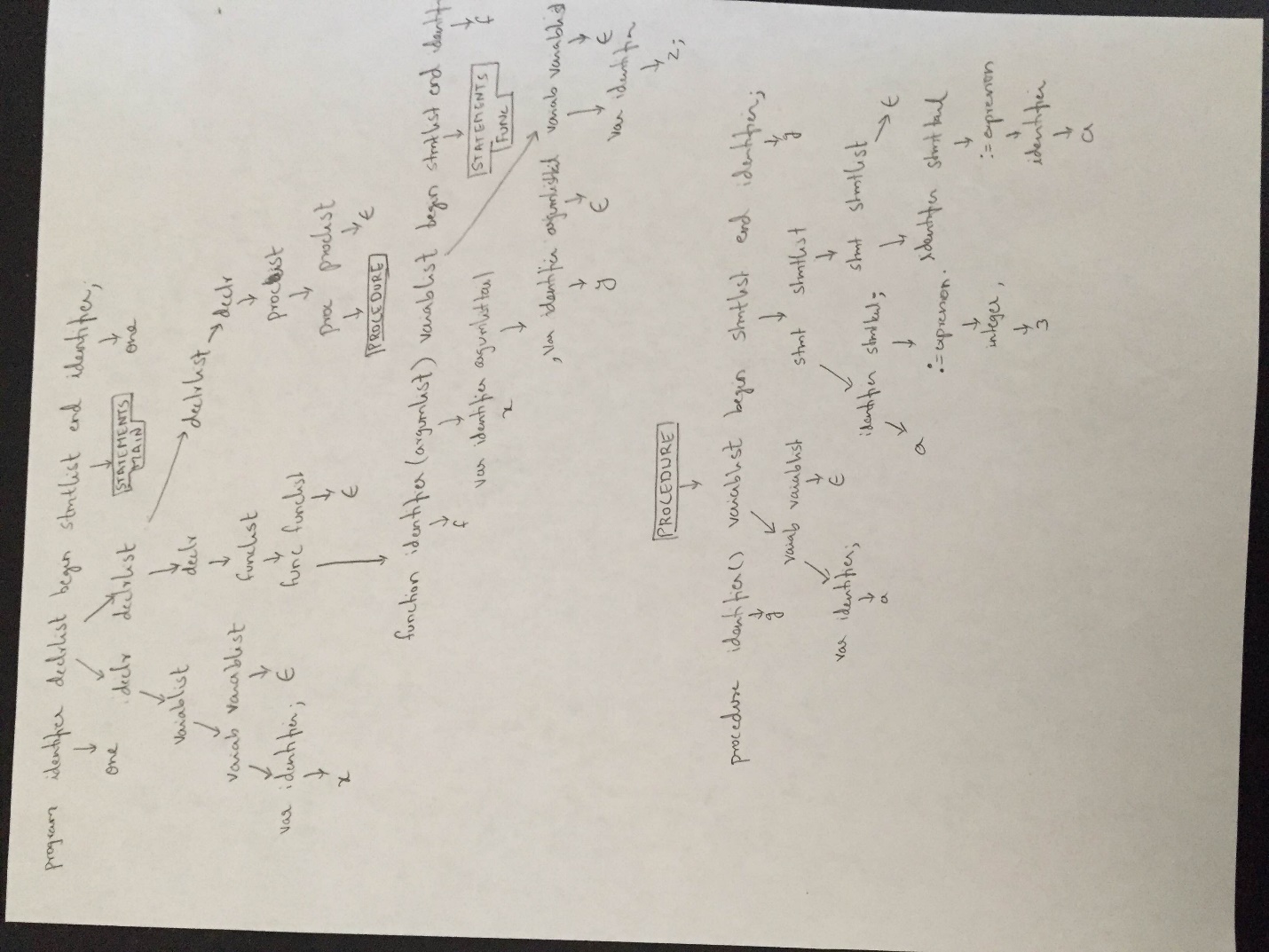
*funccall -> identifier (funccall|identifier| Є)*

*operation -> \* | / | + | -*

b)

Parse tree:





1. a) Static scoping: It is a convention where the scope(visibility) of a variable is such that it can be called or referred from within the block in which it is defined. This is determined when the code is compiled.

Dynamic scoping: It is convention where the scope(visibility) of a variable is such that it can be called or referred from within the block that is called directly or indirectly called by the block in which the variable is defined. When a symbol is defined, the compiler/interpreter will walk up the symbol-table stack to find the correct instance of the variable to use.

b) *int b = 10;*

*void func1()*

*{*

*int b = 5;*

*func2();*

*}*

*void func2()*

*{*

*print(b);*

*}*

*void main()*

*{*

*func1();*

*}*

Dynamic programming: Prints 5

Static programming : Prints 10

c) In statically scoped language, given the use of variable, we first look for definition of the variable in the same block, if we don’t find then we start looking in the next immediate outer block and keep doing this all the way out until we find the variable definition.

d) In dynamically scoped language, given the use of variable, we first look for definition of the variable in the same block, if we don’t find then we look in the function that called the current function and continue looking into all the callers one step at a time, going from the function in which it is used to all the way to the root.

This can also be put as we walk up the symbol tree to find the correct instance of variable.

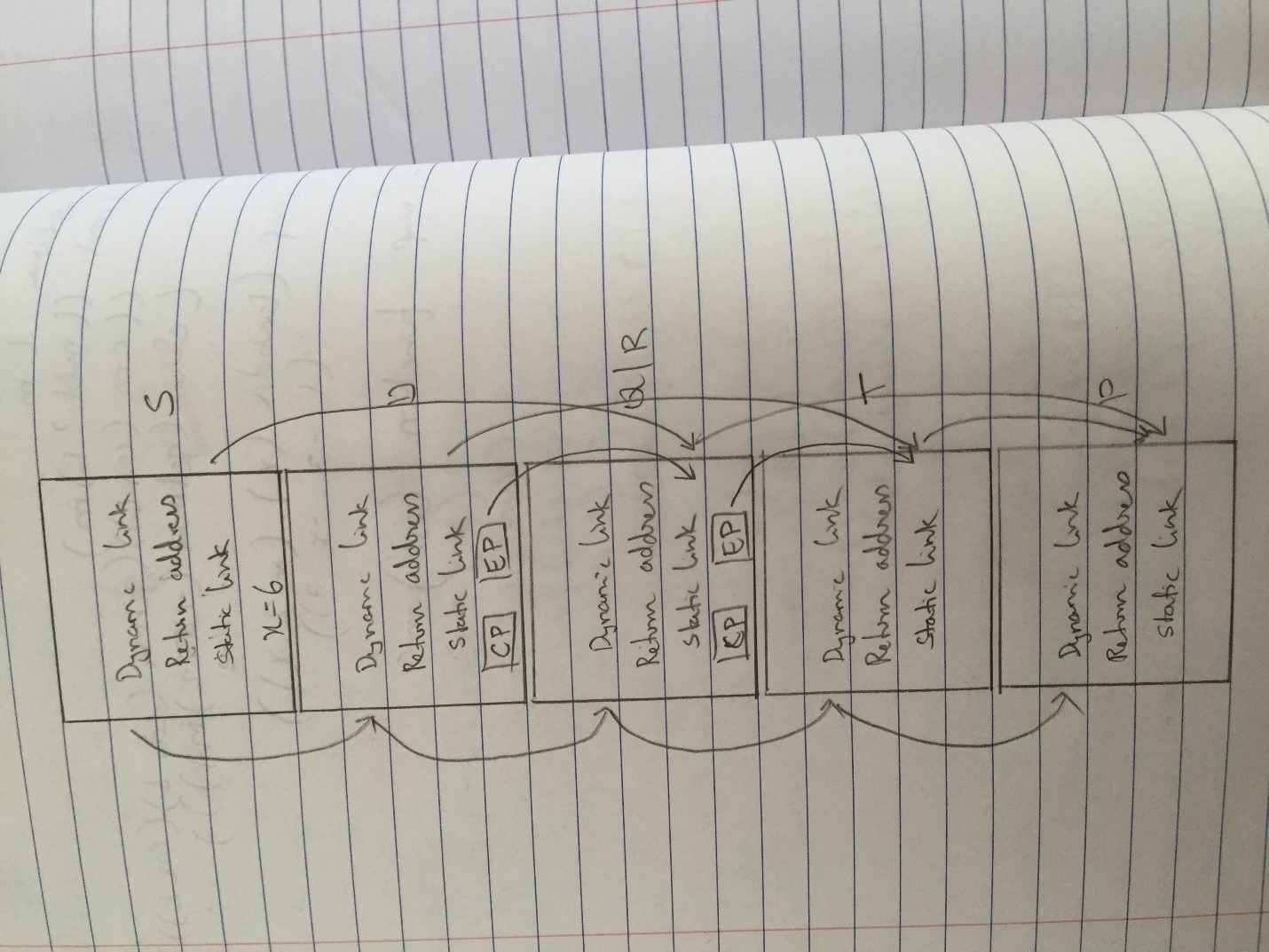
1. a) Call stack for the program.

CP indicates the code pointer and it points to the code location of the function that is passed.

In Q’s call stack, CP is pointing to code location of U.

In U’s call stack, CP is pointing to code location of S

Return address points to the address location where the code should continue execution from once the function exection is completed or function returned.



b) Closures need to be allocated on heap because it might be possible that we calls a function A that return a function (a closure) that is defined inside A and makes use of local variables of A. When the returned function is later called outside, there is an error because for the function to run it needs access to local variable of A which has been removed after A returned. To make this clear, consider the following example:

*function testFn()*

*{*

*var localVar = 10;*

*function innerFn(innerParam)*

*{*

*alert(innerParam + localVar);*

*}*

*return innerFn;*

*}*

*var someFn = testFn();*

*someFn(20);*

In this example, first we call *testFn* that return the function *innerFn* in *someFn.* However, with closures on stack, running *someFn* will throw an error because *someFn* has no idea where *localVar*  is because *testFn* has already returned and all its content(local variables including *localVar*) has been removed from stack.

1. Pass by value: 1 2 3 4 5 6

Pass by reference: 6 2 3 4 5 6

Pass by value result: 4 2 3 4 5 6

Pass by name: 1 2 3 6 5 6

1. *a)*

*with Text\_Io;*

*with Ada.Command\_Line;*

*use Text\_Io;*

*use Ada.Command\_Line;*

*procedure printsingleloop is*

*package Int\_Io is new Integer\_Io(Integer);*

*use Int\_Io;*

*task Print1 is*

*entry Start1;*

*end Print1;*

*task Print2 is*

*entry Start2;*

*end Print2;*

*task body Print1 is*

*i:Integer;*

*begin*

*i := 1;*

*while i <= 500*

*loop*

*if (i-1) mod 50 = 0 then*

*Print2.Start2;*

*accept Start1;*

*end if;*

*Put(i);*

*i:= i+1;*

*end loop;*

*Print2.Start2;*

*end Print1;*

*task body Print2 is*

*i:Integer;*

*begin*

*i :=501;*

*accept Start2;*

*while i <= 1000*

*loop*

*if (i-1) mod 50 = 0 then*

*Print1.Start1;*

*accept Start2;*

*end if;*

*Put(i);*

*i:= i+1;*

*end loop;*

*end Print2;*

*begin*

*NULL;*

*end printsingleloop;*

*b)* Concurrent programming is a type of programming wherein several streams of operations (similar to tasks in ada) are executing concurrently. However, each stream of operation executes as it would in sequential program except for the fact that the streams can communicate with each other.

In the above program, both the tasks are executing concurrently. They both start when the main program execution begins. Using entry statements tasks are communicating with one another. The first task executes and prints 1 to 50 numbers, tells second task to run and waits for signal from second. The second task then prints number from 501 to 550, signals first task to run(for which the first task was waiting until now) and waits for signal from first. This continuous with each task printing next 50 numbers alternately.

Effectively, it might look like the printing is not happening concurrently but programmatically the tasks are running concurrently. One task might not print values when other is, but it still exists. Hence, concurrent.