Q1)

a) ([a-z])* [A-Z] ([a-z])* [0-9] ([a-z])* [A-Z] ([a-z])*
b) ([0-9])* [0-9] . [0-9] ([0-9])* E [0-9] ([0-9])*
   Here, '.' = decimal point
c) ([A-Za-z]) (E | [A-za-z0-9] | _) (E | [A-za-z0-9] | _) (E | [A-za-z0-9] | _) (E | [A-za-z0-9] | _) (E | [A-za-z0-9] | _) (E | [A-za-z0-9] | _)
   Here, 'E' = empty string

Q2) a)

PROG -> DECLS
DECLS -> DECL DECLS | DECL
DECL -> VARDECL ; | FUNDECL | PROCDECL

VARDECL -> VAR:TYPE

TYPE -> INT | FLOAT | DOUBLE | CHAR | STRING | CHAR | BOOL

--

FUNDECL -> FUN FNAME (FORMAL_PARA) VARDECL { STMT_LIST }

PROCDECL -> PROC PNAME() VARDECL { STMT_LIST }

FORMAL_PARA -> FP FORMAL_PARA | FP | E

FP -> VARDECL | VARDECL , | E

STMT_LIST -> STMT STMT_LIST | STMT E

STMT -> VAR=EXPR | RETURN EXP | VAR=FNAME(ACTUAL_PARA) |
FNAME(ACTUAL_PARA)| EXPR

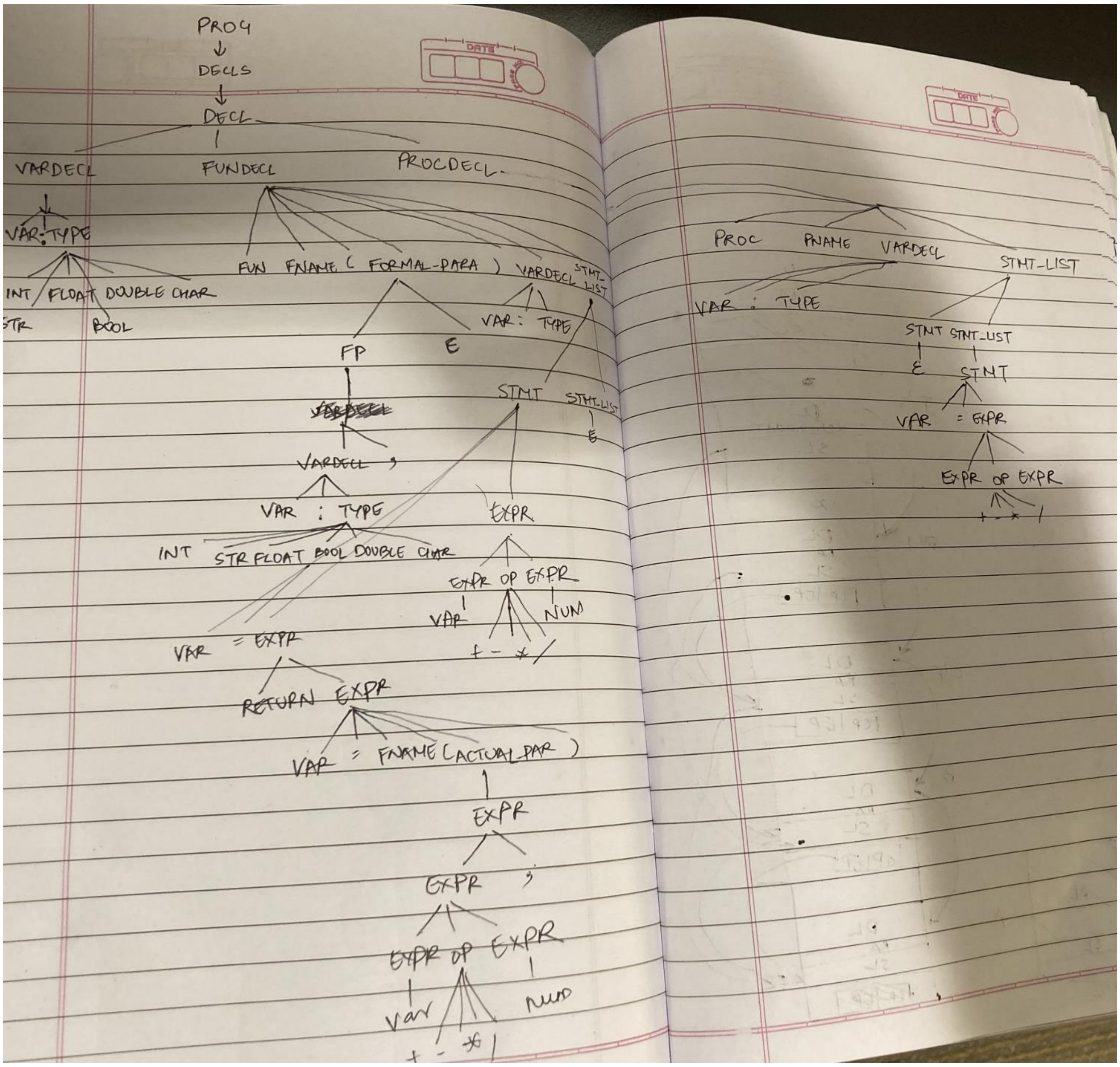
EXPR -> EXPR OP EXPR| -EXPR | (EXPR) | VAR | NUM

OP -> + | - | * | /

ACTUAL_PARA -> AP ACTUAL_PARA | AP | E

AP -> EXPR | EXPR, | E

b)



Q3)

a) Static scoping: the body of a function is evaluated in environment in which the function was defined, the binding is determined at compile team.

Dynamic scoping: the function body is evaluated in the environment of its call. The dynamic resolution can only be determined at run time (late binding)

b)

int x;

int main() {

x = 14;

```
f();
g();
}
void f() {
int x = 13;
h();
}

void g() {
int x = 12;
h();
}
void h() {
  printf("%d\n",x);
}
```

Value of x:

Static Scoping:

```
14
14
```

Dynamic Scoping:

```
13
12
```

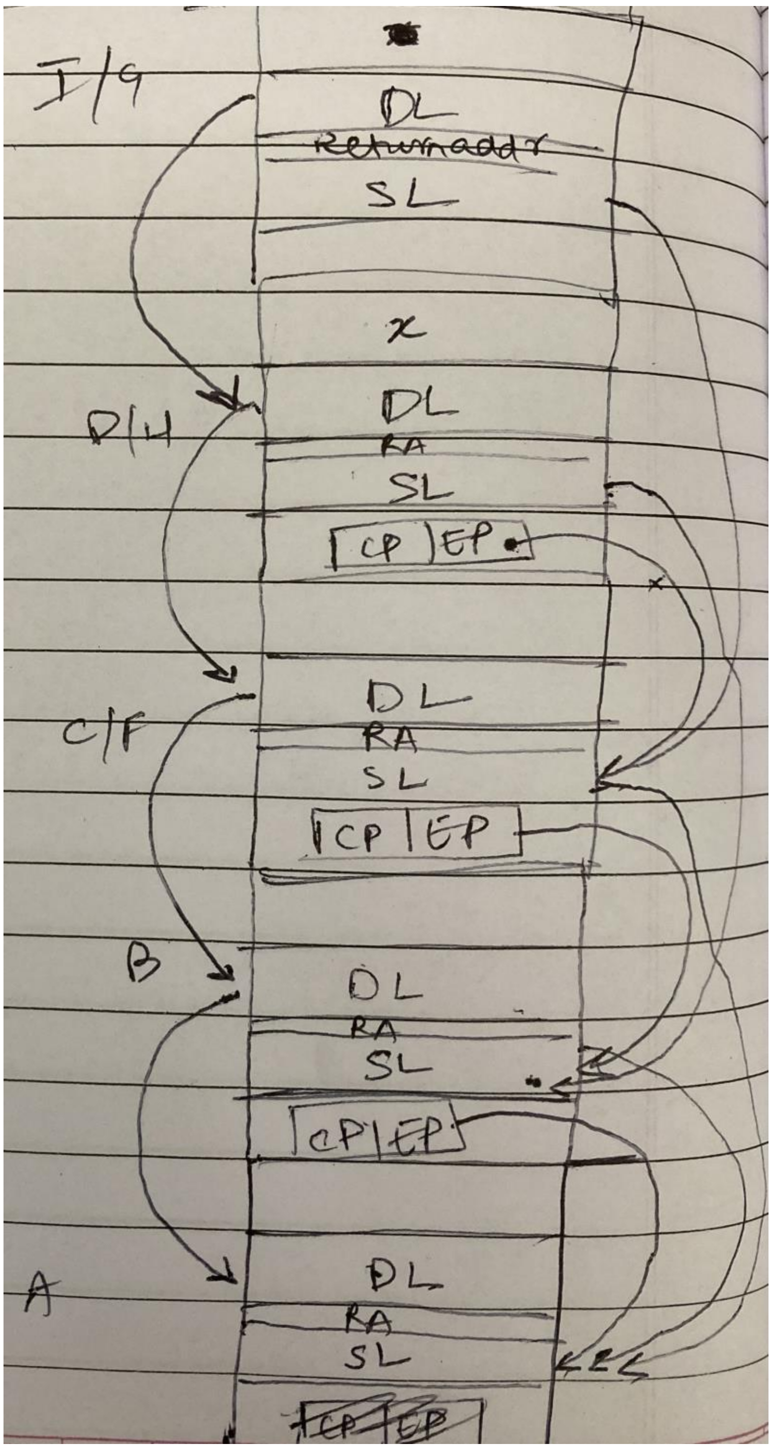
c) In a block structured, statically scoped language, what is the rule for resolving variable references (i.e. given the use of a variable, how does one find the declaration of that variable)?

- In statically scoped languages, variable references are resolved in the environment where they are defined. For ex.: in the above example variable x which is assigned a value in the main() function refers to the environment/closest visible binding for that name, which is the global definition for ‘x’.

(d) In a block structured but dynamically scoped language, what would the rule for resolving variable references be?

In dynamically scoped languages, to find the value of a name, you look at the closest binding on the call stack. The bindings between names and objects depend on the flow of control at run time, and in particular on the order in which subroutines are called, so they cannot in general be determined at compile time. The dynamic resolution can only be determined at run time (late binding)

Q4)



Q5)

(a) pass by value
 2 4 6 8 10
(b) pass by reference
 2 11 6 8 10
(c) pass by value-result
 2 4 6 7 10
(d) pass by name
 2 4 6 8 11

Q6)

```
with Text_Io;
use Text_Io;
Ada.Text_IO;
with Ada.Text_IO;

procedure HWMain is

    i1:Integer := 1;
    j1:Integer := 201;

    task Printer1 is
        entry startPrinter1;
    end Printer1;
    task Printer2 is
        entry startPrinter2;
    end Printer2;

    task body Printer1 is
    begin
        accept startPrinter1;
        count1:=0
        for i in i1..100 loop
            Int_Io.Put(i);
            i1 := i1 + 1
            count1 := count1 + 1
            if count1 := 10 then
                Printer2.startPrinter2;
        end loop;
    end Printer1;
    task body Printer2 is
    begin
        accept startPrinter2;
        count2:=0
        for j in j1..300 loop
            Int_Io.Put(j);
            j1 := j1 + 1;
            count2:=count2+1
            if count2 := 10 && j /= 300  then        --after last number of
2nd alternating sequence is printed, no need to make call to Task 1 again.
                Printer1.startPrinter1;
        end loop;
    end Printer2;

begin
    Printer1.startPrinter1;
end HWMain
```

b)

-- The printing of above sequence of numbers is not occurring concurrently, since, the task/thread body are executing in an alternating manner.

-- Two threads are concurrent if no assumption can be made about their relative order of operation between them. In this case, since the sequence generation is occurring in a predefined order, the 2 tasks cannot be said to be occurring concurrently.

Q7)

a)

```
  (define (fib n)
    (cond
      ((= n 0) 0)
      ((= n 1) 1)
      (else
        (+ (fib (- n 1))
           (fib (- n 2))))))
```

b)

```
(define (lfib x)
      (letrec
            ((fibl
                  (lambda (x a b)
                   (cond ((= x 0) a)
                  (else (fibl (- x 1) b (+ a b))))))
(fibl x 0 1)))
```