Answer 1

(a) P -> [a-zA-Z0-9]* (([A-Z][0-9]) | ([0-9][A-Z])) [a-zA-Z0-9]*

(b) F -> (+ | -)? [0-9]* (.[0-9])? [0-9]* ([E | e] [+|-]? $[0-9]^{+}$)?

(c) P -> [a-zA-z] [A-Za-z0-9_]{0,19}

———————————————————————————————————————

Answer 2

(a) PROG —> program NAME ; DECL FUNC PROC begin STMT end NAME ;
(b) DECL —> var N; DECL | ε
(c) N —> (NAME N) | (,NAME N) | ε
(d) FUNC —> function NAME (ARGS) DECL begin STMT end NAME;
(e) ARGS —> var NAME ARGS | (, var NAME ARGS) | ε
(f) STMT —> (E ; STMT) | (R ; STMT) | (C ; STMT) | (P ; STMT) | ε
(g) E —> (NAME := EXPR)
(h) EXPR —> NAME OP EXPR | NAME EXPR | NUMBER EXPR | ε
(i) R —> return NAME;
(j) PROC —> procedure NAME() DECL begin STMT end NAME;
(k) C —> NAME();
(l) P —> print(NAME(NAME));
(m) OP —> + | - | * | /

PROG → program NAME ; DECL FUNC PROC begin STMT end NAME ;

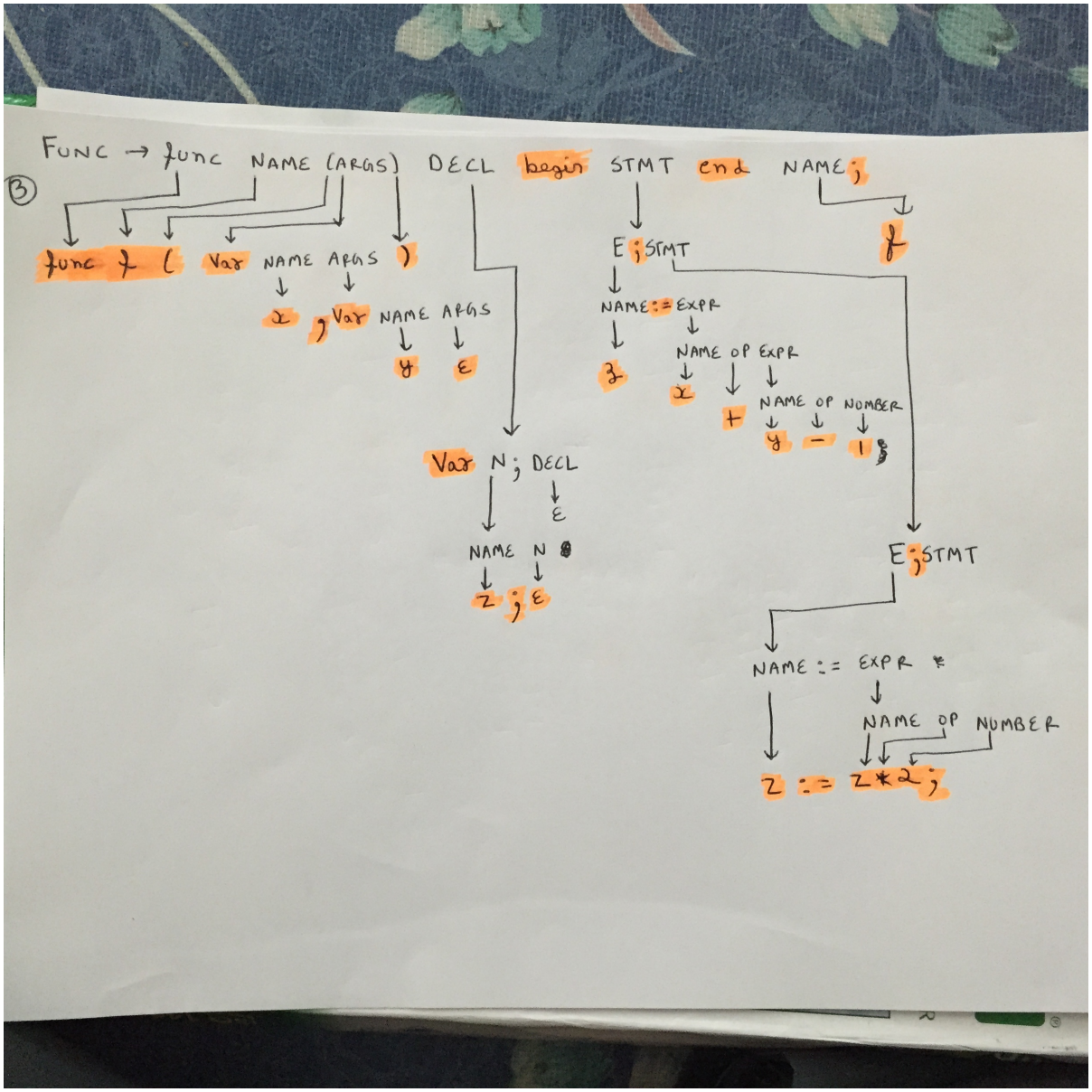
DRAWING parse Trees for Non Terminals are by one
~~NON~~ TERMINALS HIGHLIGHTED

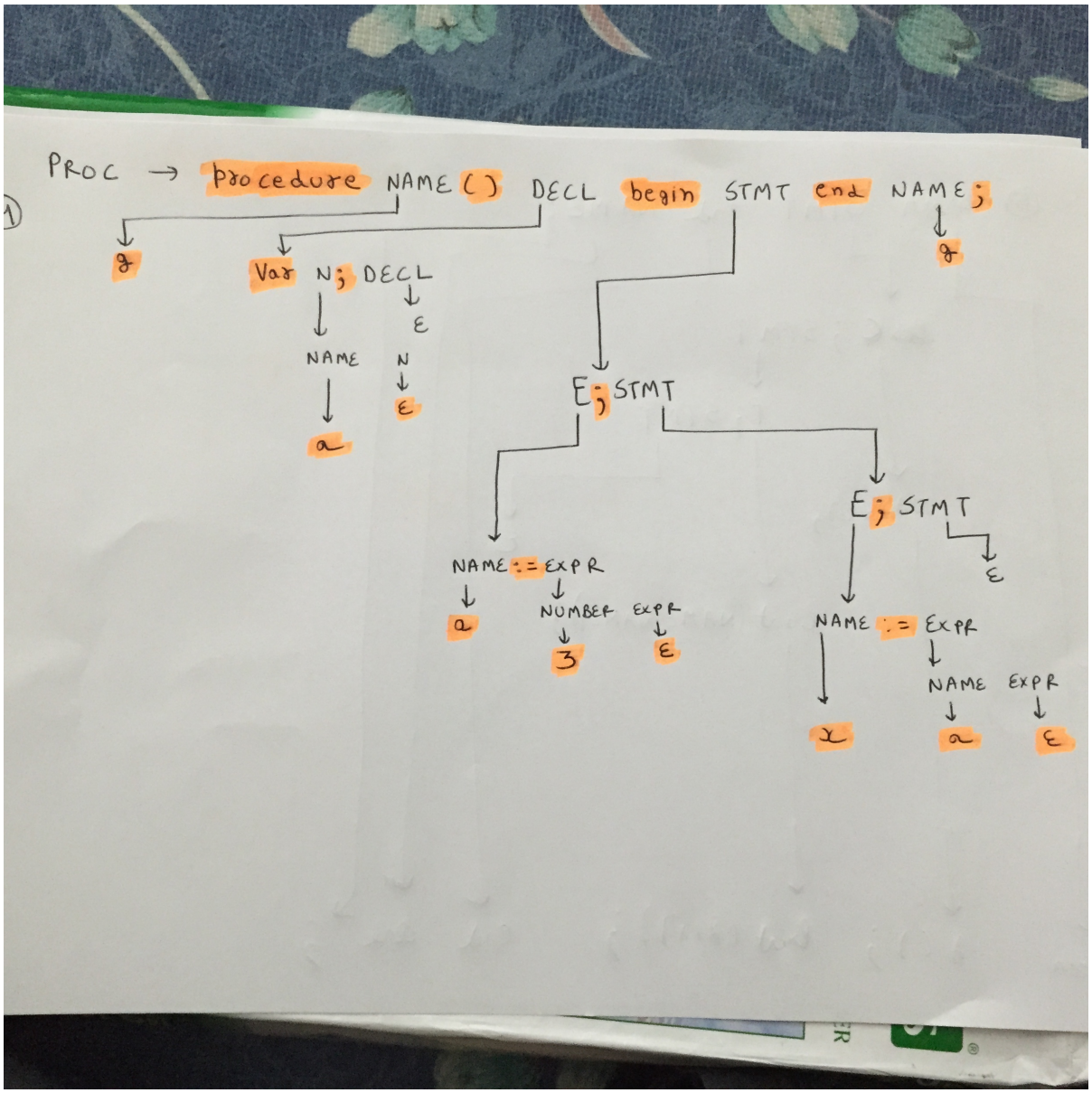
① NAME → z

② DECL → Var N ; DECL

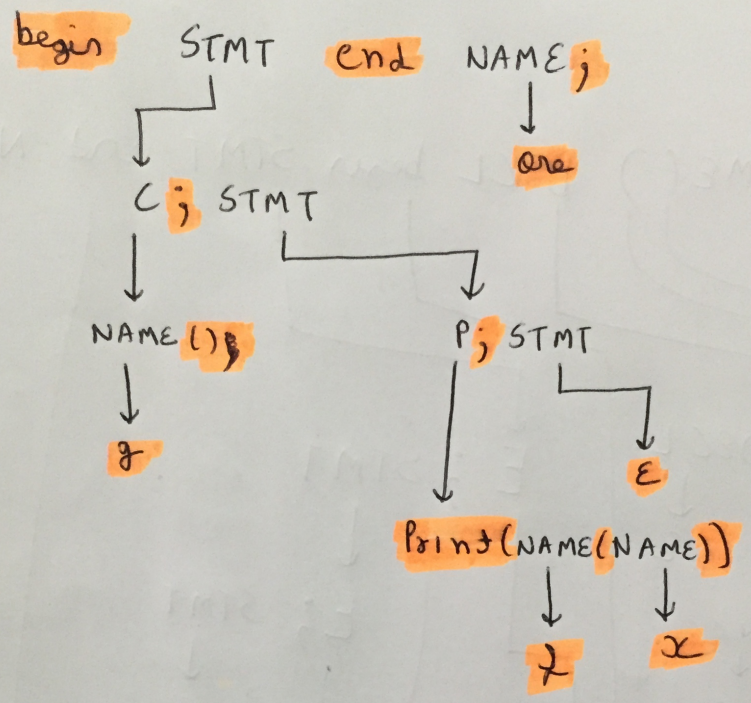
Var | NAME → z | N → ε | ; | DECL → ε

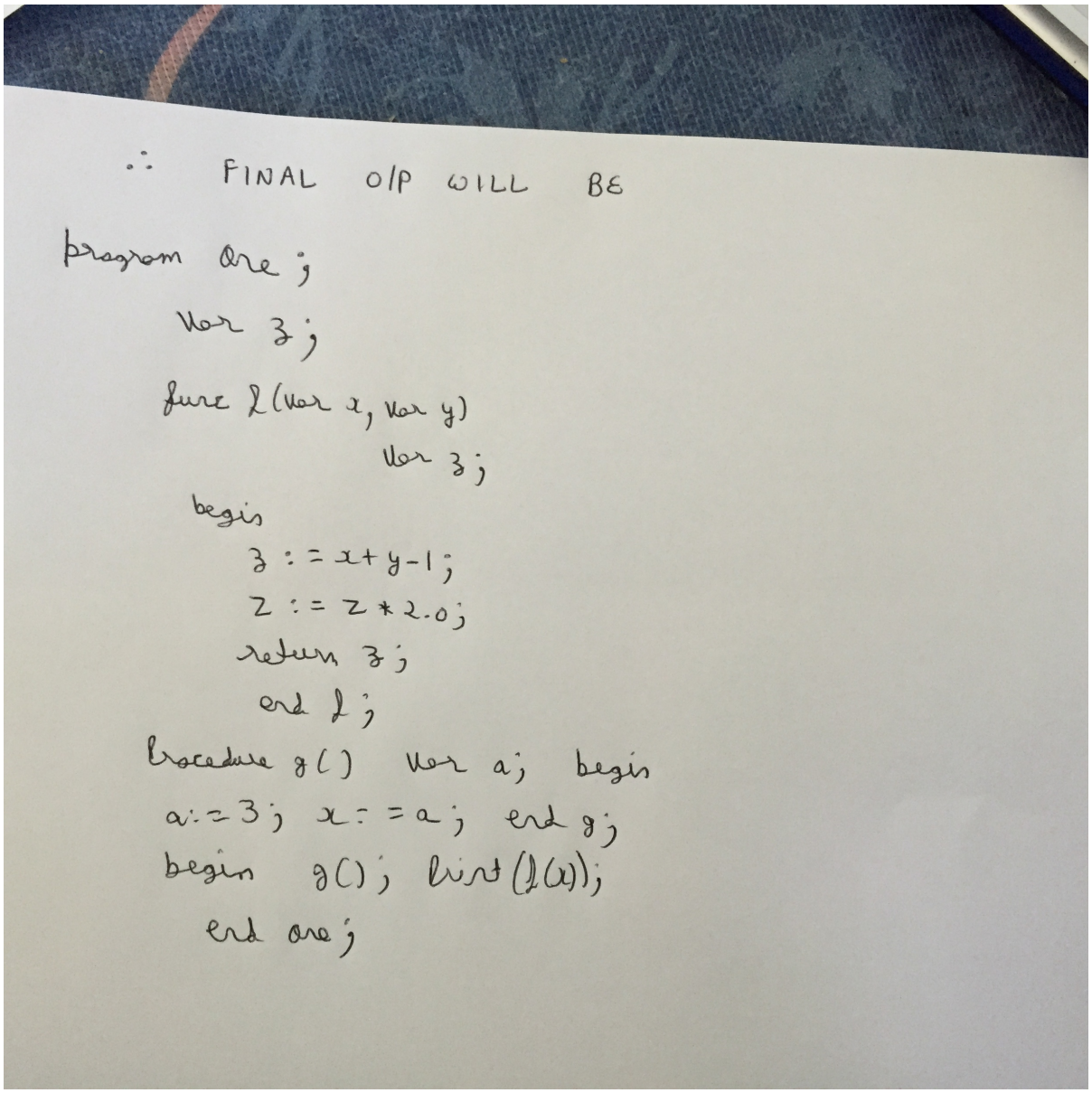
Var z ;


③
FUNC → func NAME (ARGS) DECL begin STMT end NAME ;
func f (Var NAME ARGS)
x , Var NAME ARGS
y ε
Var N ; DECL
NAME N
z ; ε
E ; STMT
NAME := EXPR
z
NAME OP EXPR
x + NAME OP NUMBER
y − 1 ;
f
E ; STMT
NAME := EXPR ;
NAME OP NUMBER
z := z * 2 ;


PROC → procedure NAME ( ) DECL begin STMT end NAME ;
g
Var N ; DECL
ε
NAME
N
a
ε
E ; STMT
E ; STMT
ε
NAME := EXPR
NAME := EXPR
a
NUMBER
EXPR
3
ε
x
NAME
EXPR
a
ε

(5)


begin
STMT
end
NAME ε ;
are
C ; STMT
NAME ( ) ;
P ; STMT
g
ε
Print (NAME (NAME))
x
x



∴ FINAL O/P WILL BE

```
program one;
    var z;
  func f(var x, var y)
            var z;
   begin
      z := x+y-1;
      z := z*2.0;
     return z;
      end f;
procedure g() var a; begin
a:=3; x:=a; end g;
begin  g(); print(f(x));
  end one;
```

Answer 3

(a,b). In static scoping, if any function is called, the reference to a variable name is resolved by the static structure of the program i.e. where the called function resides.

Example for part(b)

```
A
{
        x=10;
                B
                {
                print(x);
                }
}

C
{
        x=20;
        B();
}
```

When C calls B, the variable x will take value from A and the code will print 10.

In dynamic scoping, if any function is called, the reference to a variable name is resolved using the current state of program i.e. it will use the value of variable from that point from where the function is called.

Example for part(b)

```
A
{
        x=10;
        B
        {
                print(x);
        }
}

C
{
        x=20;
        B();
}
```

When C calls B, the variable x will take value from the point from where function is called i.e. C and hence the program will print 20.

(b).

```
A()
{
int x=10;
        B()
        {
        print(x);
        }
}

C()
{
        int x=20;
        B();
}
```

Here if static scoping is present result would be 10
if dynamic scoping present then the result will be 20

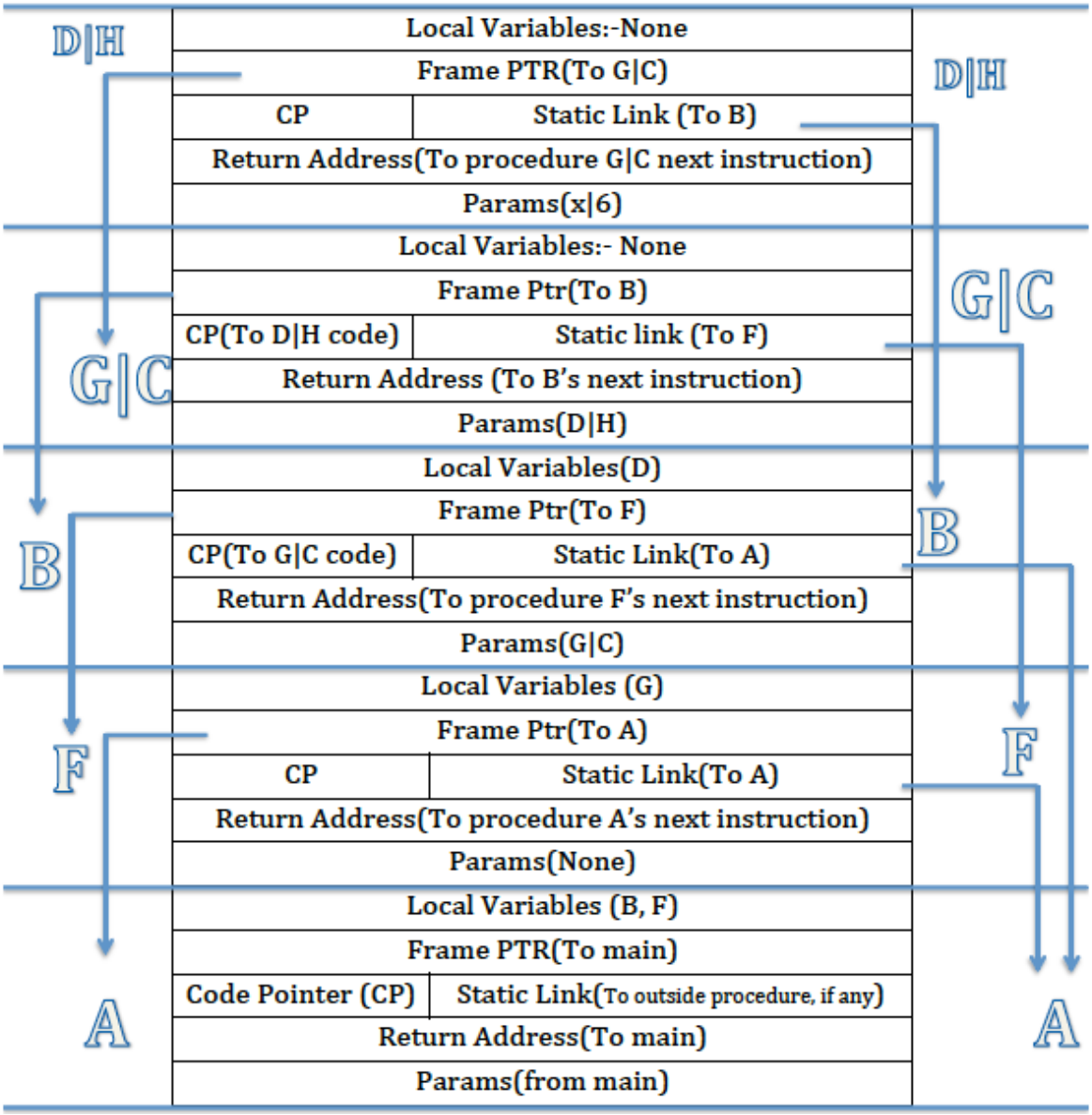
(c). In a block structured, statistically scoped language, the name of variable is resolved by using the static links of the block. Static links are developed during the compile time using the block structure of the program(environment of the block where it is defined). Number of hops across the static link needed are also calculated at compile time for each new reference variable.

(d). In a block structures, dynamically scoped language, the name of variables can be resolved by using the dynamic links. Every time a new variable referenced can be found out by carrying across the dynamic links until the variable is finally found.

———————————————————————————————————————

Answer 4

(a).


D|H
Local Variables:-None
Frame PTR(To G|C)
CP
Static Link (To B)
Return Address(To procedure G|C next instruction)
Params(x|6)
D|H
G|C
Local Variables:- None
Frame Ptr(To B)
CP(To D|H code)
Static link (To F)
Return Address (To B's next instruction)
Params(D|H)
G|C
B
Local Variables(D)
Frame Ptr(To F)
CP(To G|C code)
Static Link(To A)
Return Address(To procedure F's next instruction)
Params(G|C)
B
F
Local Variables (G)
Frame Ptr(To A)
CP
Static Link(To A)
Return Address(To procedure A's next instruction)
Params(None)
F
A
Local Variables (B, F)
Frame PTR(To main)
Code Pointer (CP)
Static Link(To outside procedure, if any)
Return Address(To main)
Params(from main)
A

(b)

Suppose if there are two procedures, one is nested inside the other and the outer procedures returns the inner procedure as a first class object. In this case inner procedure will outlive the outer procedure and the value for it should have been popped till now from the stack. Therefore, we need to store the code and environment of inner procedure(which the outer procedure returned) somewhere other than stack. Heap is used to store them.

Example:

```
function A()
x: integer

        procedure B
        begin
        x:=x*2;
        end
begin
x:=3;
return B;
end

————————————-
procedure C
begin
D=A();
D();
end
```

In this case when function A is called first by procedure C, it returns Procedure B and stores it in D. Then it pops A from stack. Now procedure C again calls D which uses the value x but value of x has been popped from stack because A is popped from stack, therefore we need to store x somewhere other than procedure. Heap is the final solution to store closures.

Answer 5

(a)	**1**	**2**	**3**	**4**	**5**
(b)	**4**	**2**	**3**	**4**	**5**
(c)	**3**	**2**	**3**	**4**	**5**
(d)	**1**	**2**	**4**	**4**	**5**

Answer 6
(a)

```
with text_io;
with ada.integer_text_io;
use text_io;
use ada.integer_text_io;

procedure tsk is
task one is
        entry O;
end one;
task two is
        entry T;
end two;

task body one is
        begin
        accept O do
        new_line;
        put("ONE");
        two.T;
        end;
end one;

task body two is
        begin
        accept T do
        new_line;
        put("TWO");
        tsk;
        end;

end two;

begin
        one.O;
end;
```

(b). These 2 tasks are not occurring concurrently as one task is waiting until the other is finished. Concurrency is when two tasks executes parallaly without waiting for one other. Also, if the tasks are occurring concurrently, we can't predetermine the order of execution of those tasks. In this case, initially task one is called which prints one and then calls task 2 which in turn prints two and call task one (via procedure), hence these two tasks are occurring sequentially.