Run-Time Exercises

1

Problem 1: Call Graph and Call Tree

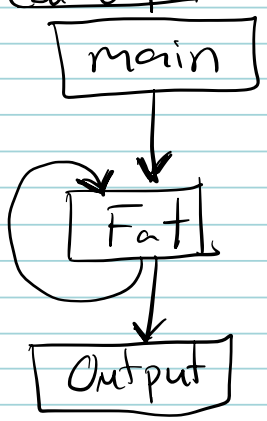
Consider the following C program:

```
void Output(int n, int x){
  printf("The value of %d! is %d.\n",n,x);
}

int Fat(int n){
  int x;
  if(n > 1)
    x = n * Fat(n-1);
  else
    x = 1;
  Output(n,x);
  return x;
}

void main(){
  Fat(4);
}
```

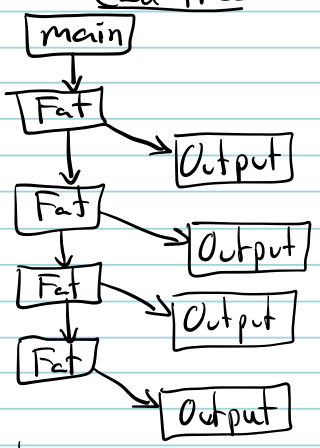
Questions:

(a) Show its call graph, *i.e.* caller-callee relationship for user defined procedures/functions.
(b) Show its call tree and its execution history, *i.e.*, the arguments' values and output produced.
(c) Discuss for this particular section of the code if the Activation Records (AR) can be allocated statically or not. Explain why or why not.

a) Call-Graph



b) Call Tree



Execution History

main calls, Fat(4)
Fat(4) calls Fat(3)
Fat(3) calls Fat(2)
Fat(2) calls Fat(1)
Fat(1) calls Output
Output returns to Fat(1)
Fat(1) returns to Fat(2)
Fat(2) calls Output
Output returns to Fat(2)
Fat(2) returns to Fat(3)
Fat(3) calls Output
Output returns to Fat(3)
Fat(3) returns to Fat(4)
Fat(4) calls Output
Output returns to Fat(4)
Fat(4) returns to main

c) In general the AR's cannot be allocated statically whenever they involve procedures that are either directly recursive as it the case of Fat or mutually recursive. In this case, there is a cycle in the call graph revealing that Fat is recursive. As a result, the AR cannot be allocated statically. However, the frames for Output can, as there is a single active instance of Output at a given point in time.

2

Problem 2: Call Graph and Call Tree

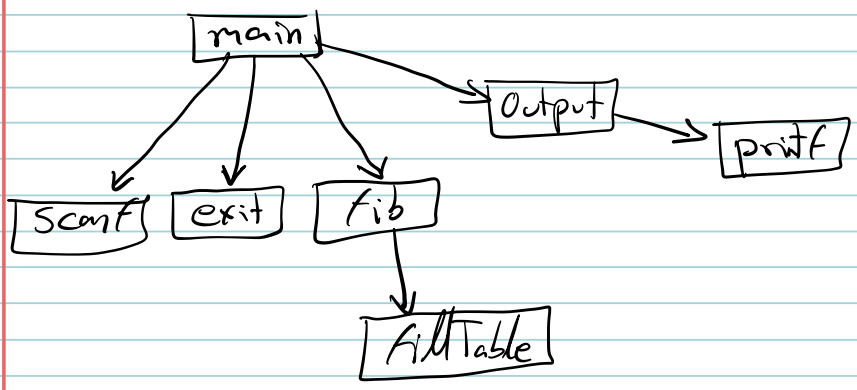
Consider the following C program:

```
#include <stdio.h>
#include <stdlib.h>

int table[1024];

void Output(int n, int x){
  printf(" Fib of %d is %d\n",n,x);
}

void fillTable(int idx){
  int i;

  for(i = 2; i <= idx; i++){
    table[i] = table[i-1] + table[i-2];
  }
}

int fib(int idx){
  if(table[idx] == 0){
    fillTable(idx);
  }
  return table[idx];
}

int main(int argc, char ** argv){
  int idx, n, k;

  for(k = 0; k < 1024; k++)
    table[k] = 0;
  table[0] = 1;
  table[1] = 1;

  while(1){
    scanf("%d",&n);
    if((n <= 0) || (n >= 1024))
      exit(0);
    k = fib(n);
    Output(n,k);
  }
}
```

Questions:

(a) Show its call graph, *i.e.* caller-callee relationship for user defined procedures/functions.
(b) Show its call tree and its execution history, *i.e.,* the arguments' values and output produced when you input the value '4' and then you input the value '3' and lastly the value '0'.
(c) Discuss for this particular section of the code if the AR can be allocated statically or not. Explain why or why not.

a) Call Graph



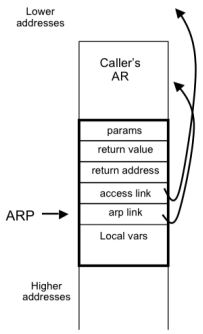
c) Pode ser alocado de forma estática, pois não existe nenhum ciclo no call-graph, revelando que fib não é recursive.
AR for fib can be allocated statically.

3

Problem 3: Activation Records

We used the abstraction of the activation record to save run-time information about where to find the non-local variables (via the access-link) and also the return addresses of procedures and functions.

```
01:  procedure main (){
02:     int a, b;
03:     procedure P(int p)
04:     begin (* P *)
05:        if (p > 0) then
06:           call P(p-1);
07:        else
08:           call Q(p);
09:     end (* P *)
10:     procedure Q(int q)
11:        int a,x;
12:        procedure R()
13:        begin (* R *)
14:            print(a,x);
15:        end (* R *)
16:     begin (* Q *)
17:        a = 1; x = 0;
18:        if (q == 1) return;
19:        call R();
20:     end (* Q *)
21:  begin (* main *)
22:    call P(1);
23:  end   (* main *)
```



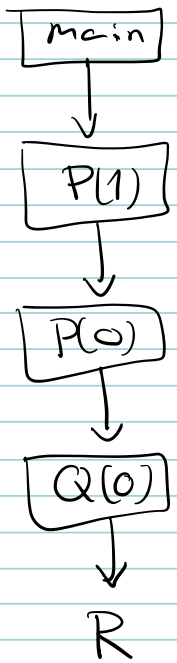
Questions:

(a) Draw the call tree for the code enclosed starting with the main procedure and ignoring library functions.

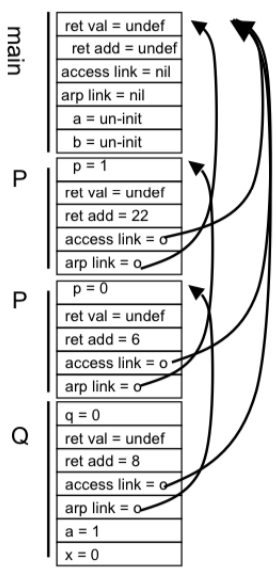
(b) Draw the configuration of the stack in terms of ARs indicating the values for the fields corresponding to parameters and local variables as well as the access link and ARP link (the stack pointers) when the execution reaches the statement on line 18. Use the organization for your activation record as shown above using the return address value as ***the same line*** as the call statement in the source code (Obviously after the call you do not execute the same call again, so the return is to the end of the same line in the source code). Justify the values for the access link field in each AR based on the lexical nesting of the procedures.

(c) Do you think using the display mechanism in this particular case would lead to faster access to non-local variables? Please explain.

a)



b)



c) Poderia ser útil e melhorar os tempos de acesso apenas para o processo R, porque iríamos precisar de 2 links para aceder às vars a e b. Mas como a função R não faz nenhum acesso a essas variáveis, o uso do método display não iria fazer nenhuma melhoria.

4

Problem 4. Run-Time Environments and Data Layout

Consider the following PASCAL source program shown below.

```
01: program main(input,  output);
02:  var track[0..3]: integer;
03:  var id: integer;
04:  function P1(a:integer):integer;
05:  var cnt: integer;
06:  begin (* P1 *)
07:    id := a;
08:    if(a > 0)
09:      cnt := P1(a-1);
10:    else
11:      track[id] = a;
12:      P1 := cnt;
13:  end;
14:  procedure P2(b: integer);
15:    begin (* P2 *)
16:      id := 0;
17:      P1(b)
18:    end;
19: procedure P3(c: integer);
20:   begin (* P3 *)
21:    id := 3;
22:    P2(c)
23:  end;
24:  procedure P4(d: integer);
25:   begin (* P4 *)
26:     id := 4;
27:     P3(d);
28:   end;
29: begin (* main *)
30:   P4(1)
31: end.
```

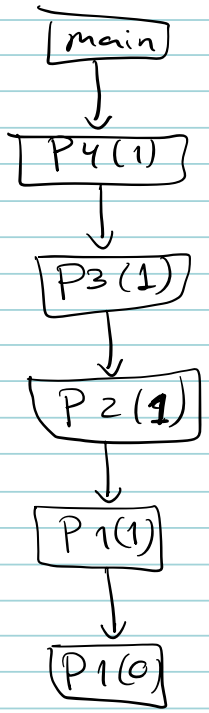
Questions:

a. Show the call tree for this particular program and discuss for this particular code if the Activation Records (ARs) for each of the procedures P1 through P4 can be allocated statically or not. Explain why or why not.

b. Assuming you are using a stack to save the activation records of all the function's invocations, draw the contents of the stack when the control reaches the line in the source code labeled as "11+" i.e., before the program executes the return statement corresponding to the invocation call at this line. For the purpose of indicating the return addresses include the designation as "N+" for a call instruction on line N. For instance, then procedure P2 invokes the procedure P1 in line 17, the corresponding return address can be labeled as "17+" to indicate that the return address should be immediately after line 17. Indicate the contents of the global and local variables to each procedure as well as the links in the AR. Use the AR organization described in class indicating the location of each procedure's local variable in the corresponding AR.

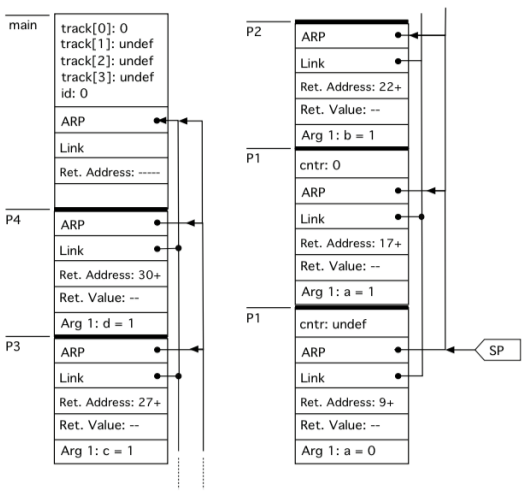
c. For this particular code do you need to rely on the *Access Links* on the AR to access non-local variables? Would there be a substantial advantage to the use of the *Display* mechanism?

a)



a) Apenas não é possível alocar as AR's de forma estática para P1. Podemos alocar de forma estática para todos os outros processos e usar uma stack mais simples para P1.

b)



(b) Stack Layout

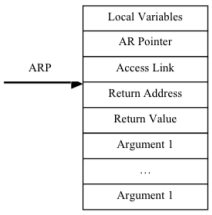
c)

c. Given that these are only accesses to local variables within each procedure or global variables (which are allocated in a specific static data section) and there are no accesses to other procedure's local variables, there is no need to use the Access Links (*Access*) in the Activation Records (*AR*) and hence no need to use and maintain the display access mechanism.

Problem 5: Activation Records and Stack Layout

Under the assumption that the AR are allocated on the stack with the individual layout as shown below, and given the PASCAL code on the right-hand-side answers the following questions:

(a) Draw the set of ARs on the stack when the program reaches line 13 in procedure P4. Include all relevant entries in the ARs and use line numbers for the return addresses. Draw direct arcs for the access links and clearly label the values of local variables and parameters in each AR
(b) Explain clearly the values of the access link fields for the instantiation of the procedure P4.




```
01: program main(input,  output);
02:    procedure P1(procedure g(b: integer));
03:      var a: integer;
04:       begin (* P1 *)
05:           a := 3;
06:           g(2);
07:       end; (* P1 *)
08:    procedure P2;
09:      var a: integer;
10:      procedure P4(b: integer);
11:        begin
12:          if(b = 1) then
13:              writeln(b);
14:          else
15:              P4(b-1);
16:        end; (* P4 *)
17:       procedure P3;
18:         var a: integer;
19:         begin
20:           a := 7;
21:           P1(P4)
22:         end (* P3 *)
23:       begin (* P2 *)
24:        a := 0;
25:        P3
26:       end; (* P2*)
27: begin (* main *)
28:   P2
29  end. (* main *)
```