Compilers

Spring 2023

Run-Time Environments

Sample Exercises and Solutions

Prof. Pedro C. Diniz

Faculdade de Engenharia da Universidade do Porto
Departamento de Engenharia Informática
pedrodiniz@fe.up.pt

Problem 1: Call Graph and Call Tree

Consider the following C program:

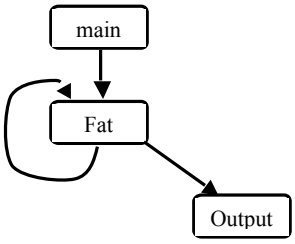
```
void Output(int n, int x){
  printf("The value of %d! is %d.\n",n,x);
}

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);
}
```

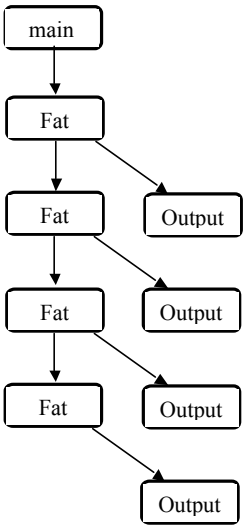
Questions:

(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.

Answers:

(a,b) Call Graph (left) call tree and call history (right)






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

Problem 2: Call Graph and Call Tree

Consider the following C program:

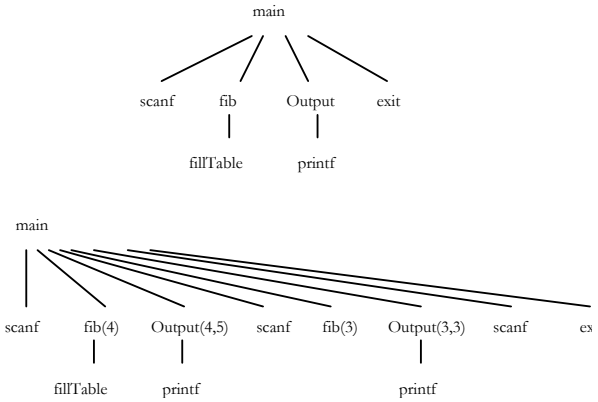
```
#include <stdio.h>
#include <stdlib.h>

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.

Answers:

(a,b) Call Graph (left) call tree and call history (right)




```
main()
 scanf("%d",&n)
 fib(4)
 fillTable(4)
 Output(4,5)
 printf(" Fib of %d is %d\n",4,5);
 scanf("%d",&n)
 fib(3)
 Output(3,3)
 printf(" Fib of %d is %d\n",3,3);
```

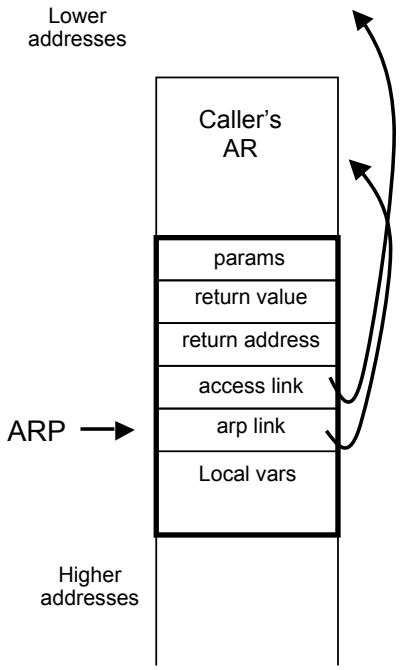
(c) In general, the ARs cannot be allocated statically whenever they involve procedures that are either directly recursive or mutually recursive. In this case there is no cycle in the call graph revealing that fib is not recursive. As a result, the AR for `fib` can be allocated statically.

Notice also that even when recursion exits it is possible to allocate some of the AR statically as long as they do not correspond to a set of mutually recursive functions in the code. A simple compiler algorithm would examine the call graph of an application and determine which portions of the call graph would not contain recursive call (i.e., would not have cycles) and would allocate statically the frames for the functions involved in this region of the call graph.

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:

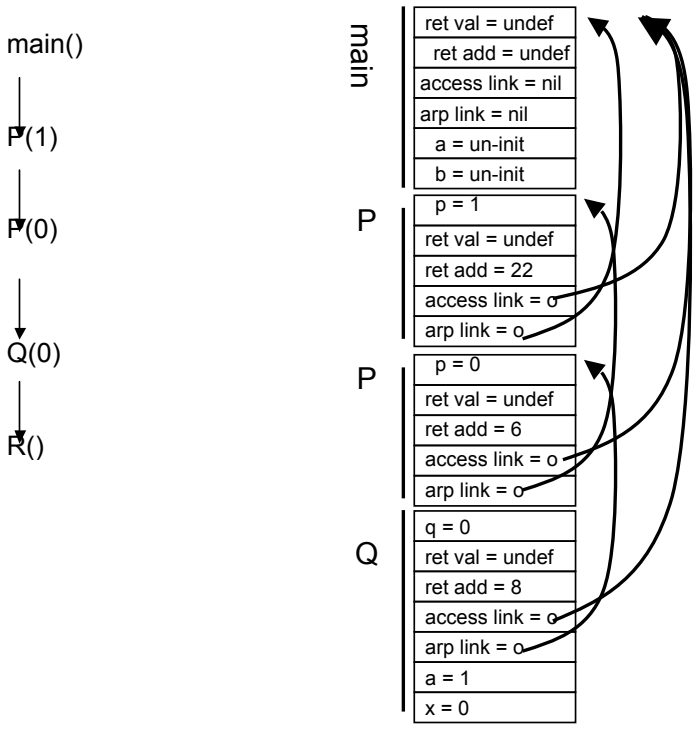
(a) Draw the call tree for the code enclosed starting with the main procedure and ignoring library functions.

(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.

Answers:

See the figure below for (a) and (b). On the left-hand-side we have the call-tree in this particular case a chain. On the right-hand-side we have the stack configuration. Note that when *P* recursively invokes itself, it copies the access-link (in this case a reference to *main*) from the previous frame of *P*. When *P* is active and invokes *Q* it also gets the access link from main as it is nested at the same level as *P* immediately within *main*.



(c) For the accesses to both P and Q we only need a single indirection via the access link so using the display does not lead to any improvement in terms of access time. For the procedure R, however, we would need 2 links to access the variables int a, and int b local to the main procedure. Only for that procedure R would the display make any difference. In practice and given that in reality R makes no accesses at all to non-local variable we actual do not need the display.

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.

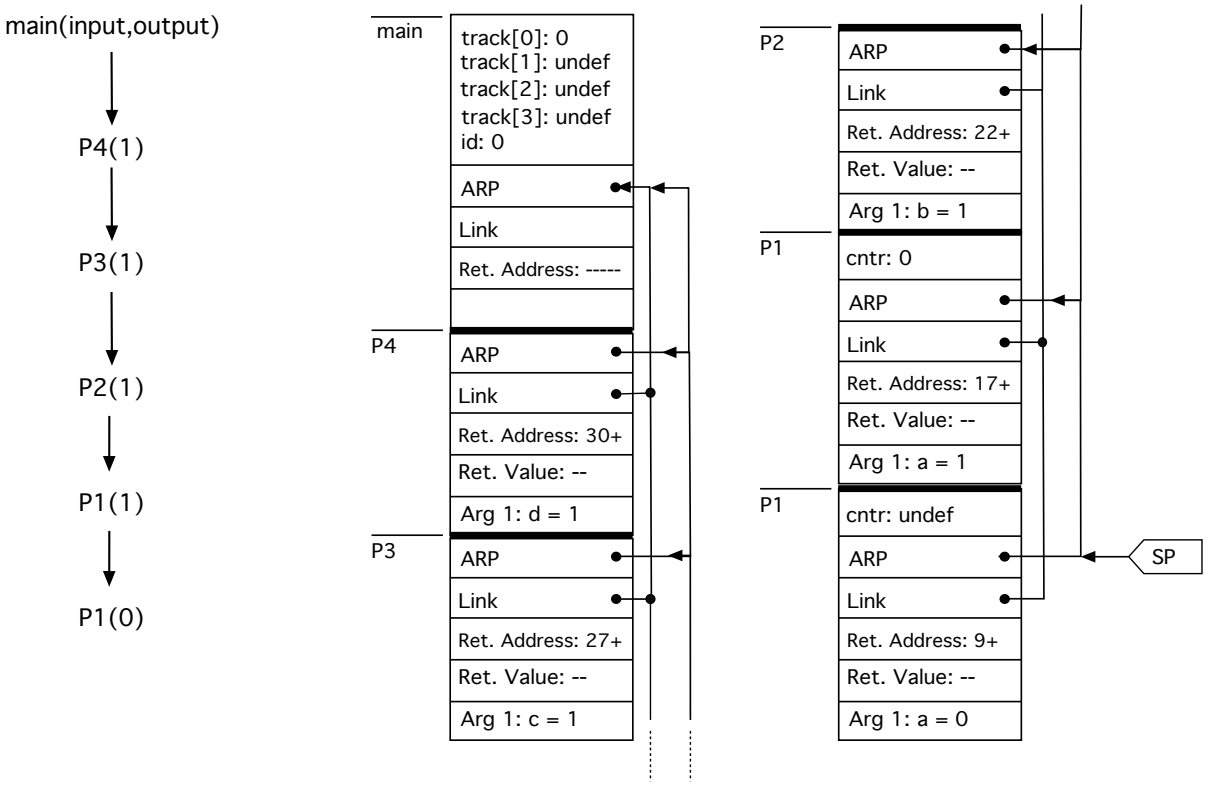
b. Assuming you are using a stack to save the activation records of all the function's invocations, draw the contents of the stack when the control reaches the line in the source code labeled as "11+" i.e., before the program executes the return statement corresponding to the invocation call at this 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?

Answers:

a. Given that there are recursive function calls (in this particular case P1), we cannot in general allocate the activation records of all these functions in a global region of the storage. However, in this particular case we observe that the recursive calls are confined to a very small portion of the call graph as P1 only calls itself. As such we could in principle allocate all the remainder procedures statically and use a simpler stack for the activations of P1.

b. The figure below depicts both the call tree and the stack configuration when the execution reaches the line 11+ in the source program.



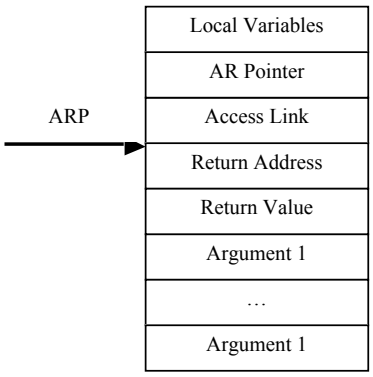
(a) Call Tree (b) Stack Layout

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:

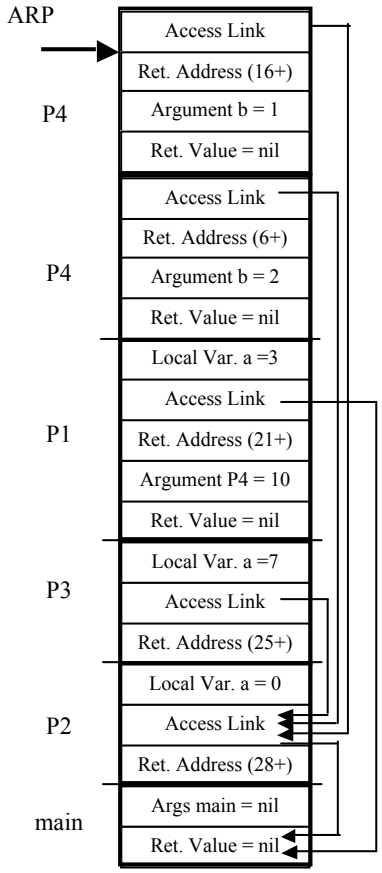
(a) Draw the set of ARs on the stack when the program reaches line 13 in procedure P4. Include all relevant entries in the ARs and use line numbers for the return addresses. Draw direct arcs for the access links and clearly label the values of local variables and parameters in each AR
(b) Explain clearly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

Solution:

(a) The figure below depicts the state of the call stack just before the program returns control after the invocation on line 13 in procedure P4. This invocation causes the two recursive invocations of the procedure P4, the second of which has an argument with the value 1 and thus will cause the program control to reach line 13. Notice that in this case the stack is growing upwards in the figure and as a result the access links are pointing downwards. Note for simplicity we have omitted the ARP links in this figure.

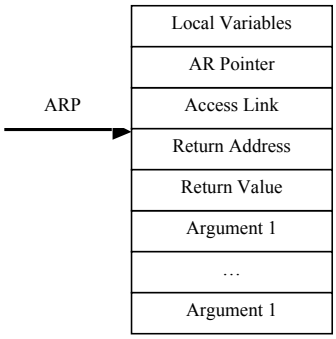


(b) All access links point to the procedure whose lexical code is the more recent (in terms of active invocation history) and immediately enclosing nesting scope. For the case of P4 the immediate nesting scope of P4 is P2 and thus both recursive invocations need to have their access link entries pointing to P2 single AR on the stack.

Problem 6: Activation Records Stack Layout

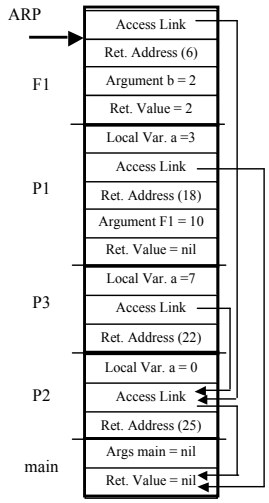
Under the assumption that the AR are allocated on the stack with the individual layout as shown below, draw the set of active AR on the stack for the code in the figure below just prior to the return from the function F1. Include all relevant entries in the ARs and use line numbers for the return addresses. Draw direct arcs for the access links and clearly label the values of local variables and parameters in each AR.

Draw the set of ARs on the stack when the program reaches line 7 in procedure p1.




```
01: program main(input,  output);
02:  procedure P1(function g(b: integer):integer);
03:  var a: integer;
04:   begin (* P1 *)
05:     a := 3;
06:     writeln(g(2))
07:   end; (* P1 *)
08:  procedure P2;
09:    var a: integer;
10:    function F1(b: integer): integer;
11:   begin (* F1 *)
12:      F1 = a + b;
13:   end; (* F1 *)
14:  procedure P3;
15:   var a: integer;
16:   begin (* P3 *)
17:    a := 7;
18:    P1(F1)
19:   end (* P3 *)
20:  begin (* P2 *)
21:   a := 0;
22:   P3
23:  end; (* P2 *)
24:  begin (* main *)
25:    P2
26:  end. (* main *)
```

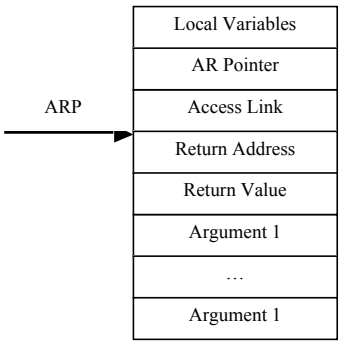
Answers:



Problem 7: Activation Records 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:

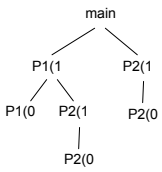
(a) Draw the call tree starting with the invocation of the main program.
(b) Draw the set of ARs on the stack when the program reaches line 16 in procedure P2. Include all relevant entries in the ARs and use line numbers for the return addresses. Draw direct arcs for the access links and clearly label the values of local variables and parameters in each AR.
(c) For this particular example would there be any advantage of using the Display Mechanism?

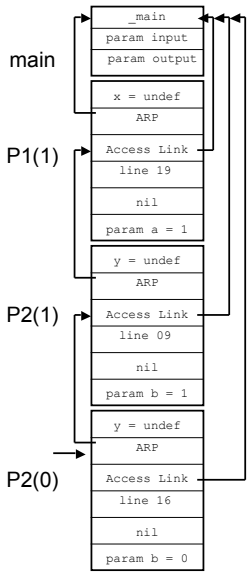



```
01: program main(input,  output);
02:    procedure P1(a: integer));
03:      var x: integer;
04:      begin
05:     if (a <> 0) then
06:       begin
07:         P1(a-1);
08:         P2(a);
09:       end
10:      end;
11:      procedure P2(b: integer);
12:        var y: integer;
13:        begin
14:       if(b <> 0) then
15:          P2(b-1);
16:        end;
17:  begin (* main *)
18:    P1(1);
19:    P2(1);
20:  end.
```

Answers:

(a) and (b) See the call tree on the left and the stack organization on the right below where the values of the access link and activation records pointers are explicit.





(c) None at all. Given that there are no nested procedures the only non-local variables that a given procedure needs to access are the global variables (like in C). So, you only need to have the ARP (or Frame-Pointer – FP) and a second GP or Global Pointer registers. Most architectures do support these two registers in hardware.

Problem 8: Activation Records Stack Layout

In the PASCAL language one can define functions and procedures that are local to other procedures, *i.e.,* they are only visible within the scope of that immediately enclosing procedure. This is analogous to the use of nested blocks in the C language. For example, in the code below procedure `f3` is only visible inside the code of procedure `f2`. But neither inside the code of `f1` nor inside the procedure `main`.

```
01:  procedure main
02:    integer a, b, c;
03:    procedure f1(a,b);
04:      integer a, b;
05:      call f2(0,b,a);
06:    end;
07:    procedure f2(x,y,z);
08:      integer x, y, z;
09:      procedure f3(m,n);
10:          integer m, n;
11:          ...
12:      end;
13:      procedure f4(m,n);
14:        integer m, n;
15:         ...
16:      end;
17:      call f3(y,z);
18:      call f4(c,x);
19:    end;
20:  ...
21:  call f1(a,b);
22:  end;
```

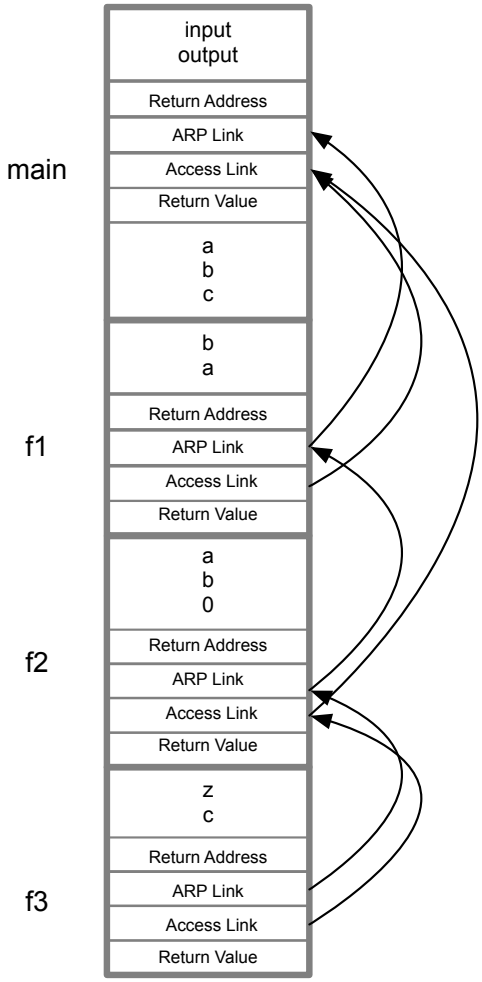
For this code answer the following questions:

a) On line `17` and `18`, which variables are used as the actual values of the parameters of both calls to `f3` and `f4` respectively.
b) Draw (using simplified Activation Records layout) when the control flow of the program reaches line `11` (i.e., inside procedure `f3`). Draw the links regarding the ARs and the Access Links used to access non-local variables. Explain how the compiler can generate code to access the local variable x of the procedure `f2` in the body of procedure `f3`.

Answers:

a) The arguments for the call to procedure `f3` on line `17` includes variables `y` and `z` which are local variables to procedure `f2`. The arguments for the call to procedure `f4` on line `18` includes variables `c` and `x` which are local variables to `main` and procedure `f2`.

b) See figure below where only the Access Link and ARP Link are shown.



Problem 9: Procedure Storage Organization

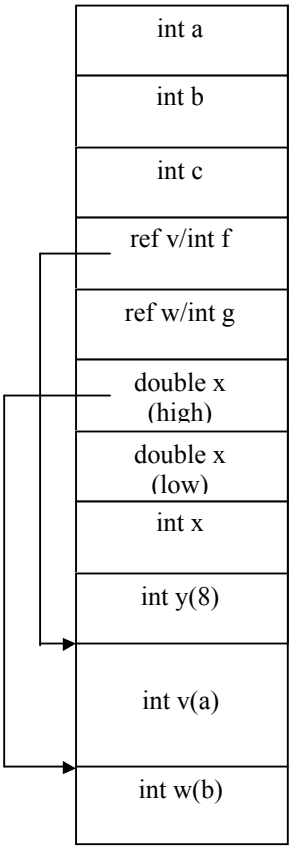
Consider the following C program and assume that the ARs follow the same general layout as discussed in problem 2 above. Assuming that an integer requires 4 bytes and a double data type 8 bytes, derive a layout for the local variables for the AR of the enclosing procedure P detailing the sharing of storage space between variables.

```
B0: {
     int a, b, c
     … assign value to a and b
B1:    {
         int v(a), w(b)
         double x;
B2:      {
          int x, y(8)
          ….
         }
     }
B3:  {
       int f, g;
        ….
     }
```

Answers:

In this solution B3 is disjoint from both B1 and B2 and thus the space used for the local variables of the blocks B1 and B2 can be shared with the space required for B3. This last block B3 only requires 8 bytes for the two variables f and g, which is shared with space for the references to the v and w arrays.

Regarding the space for arrays v and w, as they have parameterizable lengths they are decoupled as a pointer (ref v and ref w respectively) pointing to the end of the AR area. All other data is known statically (i.e., at compile time).

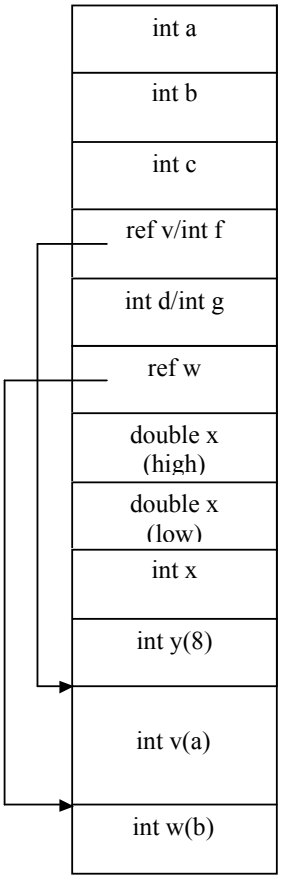


Problem 10: Procedure Storage Organization

Consider the following C program and assume that the ARs follow the same general layout as discussed in problem 2 above. Assuming that an integer requires 4 bytes and a double data type 8 bytes, derive a layout for the local variables for the AR of the enclosing procedure P detailing the sharing of storage space between variables.

```
B0: {
   int a, b, c
   … assign value to a and b
B1:      {
      int v(a), d, w(b)
     double x;
B2:        {
          int x, y(8)
          ….
     }
               }
B3:       {
     int f, g;
      ….
               }
```

Solution:



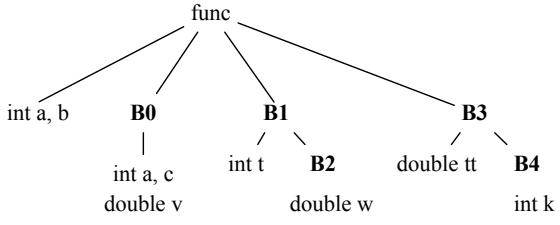
Problem 11: Run-Time Environment and Storage Allocation

a) Describe the basic structure of an Activation Record (AR).
b) Where are ARs stored at run-time. Why?
c) Why do you need an AR for? What features of a language requires explicit AR and why?
d) What is the purpose of the access link? What language mechanism requires its use? Why?
e) For the code structure shown below determine the location in the AR of each local variable in each block. Assume that the first local variable begins at the zero offset and that integer values take 4 bytes and double values use 8 bytes.

```
void func(){
    int a, b;
    B0: {
            int a, c;
            double v;
        }
    B1: {
            int t;
            B2:{
              double v;
            }
        }
    B3: {
            double tt;
            BB4:{
                  int k;
            }
        }
}
```

Answers:

a) The AR has a field for each function/procedure parameter, return address, return value, the ARP link and access link as well as local variables.
b) The AR are allocated and maintained on the stack. This is because their life span is connected to the activation or lifetimes of the corresponding procedures and function, which exist or form a stack over time following the *caller-callee* relationship.
c) To capture the execution context of a procedure/function. Recursion is the basic language mechanism that requires an AR for each invocation, as there can be multiply active instances of the local variables.
d) The access link is used to access non-local variables in languages supporting lexical scoping.
e) The key issue in allocating the space in the AR is to determine which scope blocks are simultaneously active. To capture this, we can organize the nesting of these scopes in a tree as shown in the figure below and them it is obvious what space can be reuse when each of the scope are no longer active.



int a			4 bytes
int b			4 bytes
int a	int t	double tt	4 bytes
int c	double w	double tt	4 bytes
double v	double w	int k	4 bytes
double v			4 bytes

Problem 12: Run-Time Environment and Storage Allocation

a) Describe the basic structure of an Activation Record (AR).
b) Where are ARs stored at run-time. Why? Can they be stored in a static area?
c) What is the purpose of the Frame Pointer link (FP Link) and the Access Link (AL)? What language mechanism requires their use? Why?

Answers:

a) Describe the basic structure of an Activation Record (AR).

This run-time structure has field to hold the values of the functions' arguments, the return address, the previous stack AR frame pointer and the Access Link pointer (to be explained below). It also holds fields to hold the values of the function local variables and in some cases temporary variables.

b) Where are ARs stored at run-time. Why? Can they be stored in a static area?

They can be stored in a static compile-time defined area as long as there are no recursive calls. In the case of imperative languages with recursion like Pascal or C they need to be allocated on the stack as there as multiple active instances of the local variables in each function. For the case of functional languages, where the life of some of the function's variables outlive the function in which they were created, they need to be allocated on the heap.

c) What is the purpose of the Frame Pointer link (FP Link) and the Access Link (AL)? What language mechanism requires their use? Why?

The Frame-Pointer (FP) or Stack-Pointer is the field in the activation record (AR) that points to the previous invoked function or procedure. The chain of FP values thus highlights the currently active call chain in the call-tree. The Access Link is a field in the AR that indicates where the find the non-local variables the code of the function the AR corresponds to uses to access them. This AL is only used in languages that have lexical nested scopes such as Pascal. In C there is only two scoping levels, local and global and this field is not needed.

Problem 13: Implementing Object-Oriented Language

Consider the following prototypical Object-oriented program that supports both single and multiple inheritance. Here the class Cpoint inherits from both Point and Cthing.

```
Class Point {
   int x, y;
   void draw();
   void d2o();
}
Class CThing {
   Color c;
   void rev();
}
Class CPoint extends Point, CThing {
   void draw()
}
```

For this code excerpt answers the following questions:

a) Draw the object and class for three objects, one of each class, corresponding to the object declaration below:

```
Point p;
CThing c;
CPoint cp;
```

b) Explain how the compiler will generate code to access the various fields of each declared object and the need for the use of the trampoline function (and for which method invocation it is needed).

c) Consider now a different scenario. Here a class, named Point3D inherits from Point and the CPoint class now inherits both from CThing and Point3D as shown below:

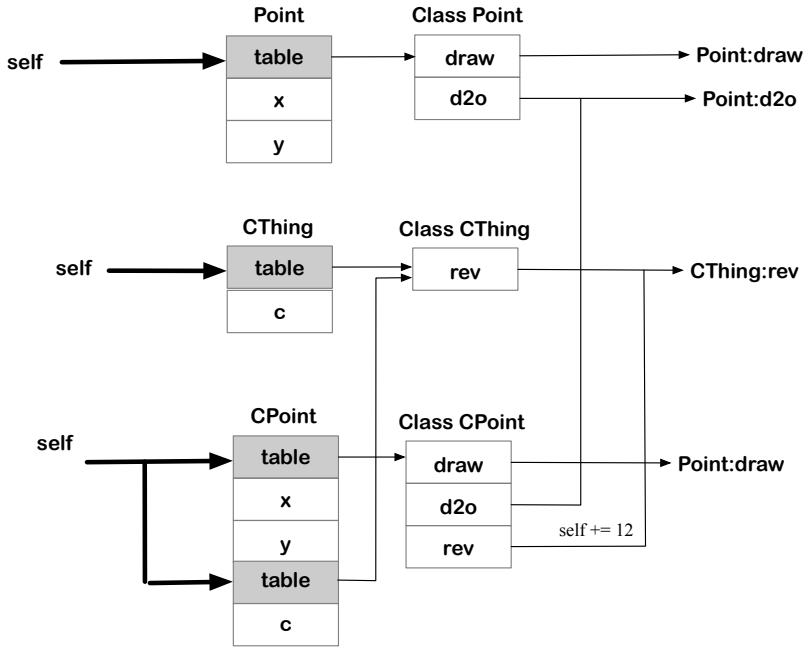
```
Class Point {
   int x, y;
   void draw();
   void d2o();
}

Class CThing {
   Color c;
   void rev();
}
```

```
Class Point3D extends Point {
     int z;
     void draw();
}

Class CPoint extends CThing, Point3D {
   int d;
      void tang();
}
```

What would the object layout for object of class CPoint look like? What changes would you need to do to the code of the inherited methods? Identify the basic issues of the object record layout and discuss some alternatives.

Answers:

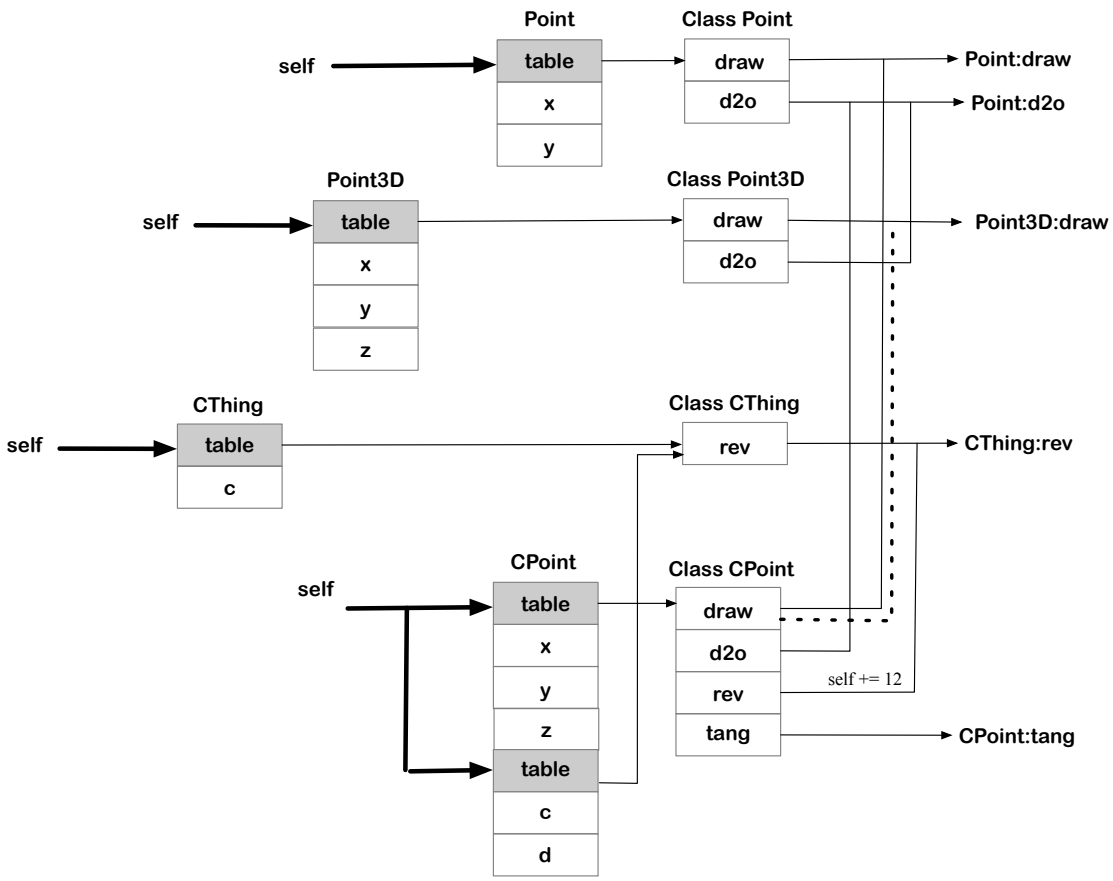
a) As discussed in class with an inheritance that defines a tree-like inheritance graph we would have a way to combine the object records and the corresponding class method tables as shown below.



b) For the methods draw and d2o from the Point class, the access to the object fields is straightforward. The self pointer value can be used as the offset of the fields. As an example, x and y will be located at offsets 4 and 8 respectively under the assumption that each integer field takes up 4 bytes. As for the rev method defined in the CThing class, it can also directly use the self pointer value for the objects from class CThing. The draw method when executed over object of class CPoint also does not need to have the self pointer adjusted since the fields x and y are located at the same offset as the original draw method (from class Point) expects them. The only issue is for method rev define over objects of class CPoint. Here, the self pointer is way off and needs to be adjusted so that when the rev method executes it finds the c field at offset 4 of the already adjusted self pointer. In this case, the adjustment corresponds to 12 bytes as it has to "jump over the layout of the x, y and a class table reference corresponding to the layout of object of the Point class. This is the reason of the need to have the trampoline function.

c) In this particular case, the inheritance graph is still a tree. So, the layout of an object in terms of its data can still be organized correctly (where the offsets are preserved). In this case we need to define which data members of which of the Point3D to be laid out first followed by the data fields of the CThing class and last followed by the fields of CPoint. We can assume it is the first class that appears in the inheritance list. Regarding the method table there are complications. As you can see in this example both classes Point and Point3D define the draw method. So, which is seen in class CPoint? In various languages this is a syntactical error, but for the purposes of this exercise we assume that methods inherited first cannot be later redefined. In this specific example, the draw method in the class CPoint will correspond to the method first defined in the class Point and not the from the derived class (can you see why?).

The figure below depicts a possible object layout as well as the corresponding classes' method tables along with the trampoline functions.



The situation would be complicated if class CThing also inherited from Point, thus creating an inheritance graph that would be non-tree like as then both fields of these classes would "collide" in the object "record" definition. While there are flexible solutions to this issue, they are expensive from a static and dynamic execution standpoint.