

Compiler Design

Spring 2011

Run-Time Environments

Sample Exercises and Solutions

Prof. Pedro C. Diniz

USC / Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, California 90292
pedro@isi.edu

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;
}

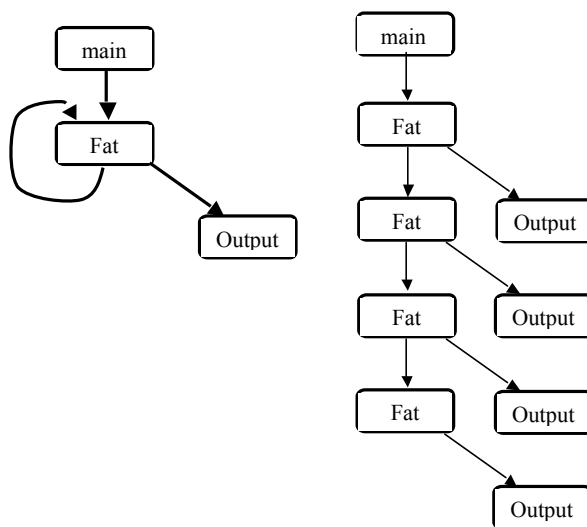
void main(){
    Fat(4);
}
```

Questions:

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

Solution:

(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

- 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;

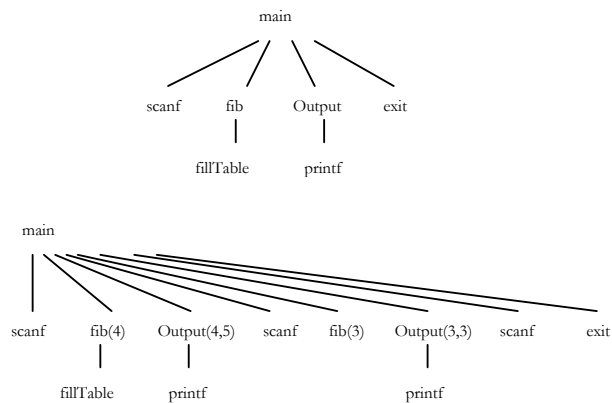
        while(1){
            scanf("%d",&n);
            if((n <= 0) || (n >= 1024))
                exit(0);
            k = fib(n);
            Output(n,k);
        }
    }
}
```

Questions:

- Show its call graph, *i.e.* caller-callee relationship for user defined procedures/functions.
- 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'.
- 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.

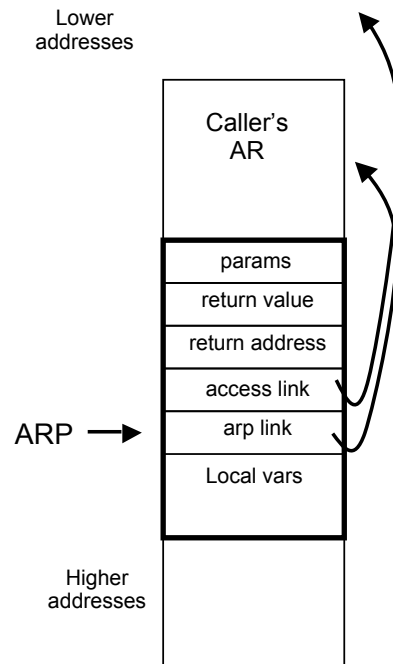
Problem 3: Activation Records and Run-Time Environments

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 *)

```

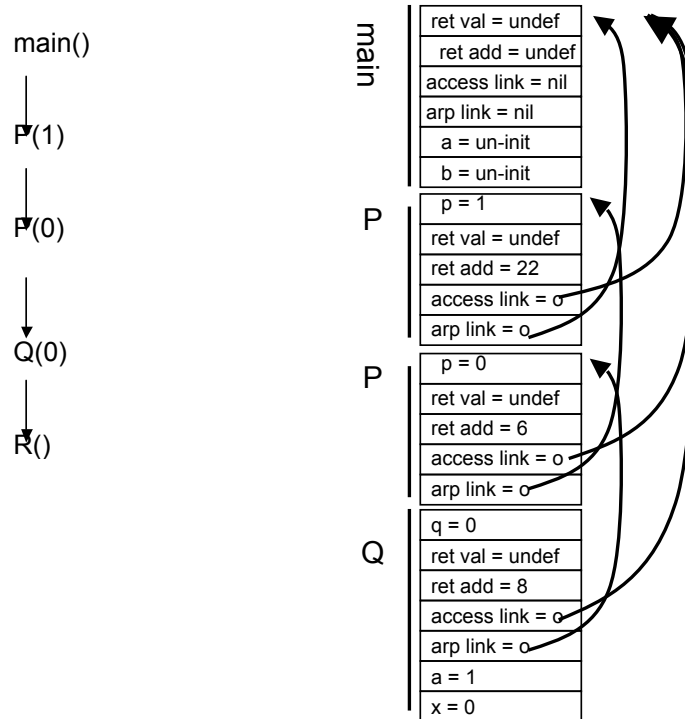


Questions:

- Draw the call tree for the code enclosed starting with the main procedure and ignoring library functions.
- 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.
- Do you think using the display mechanism in this particular case would lead to faster access to non-local variables? Please explain.

Solution:

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*.

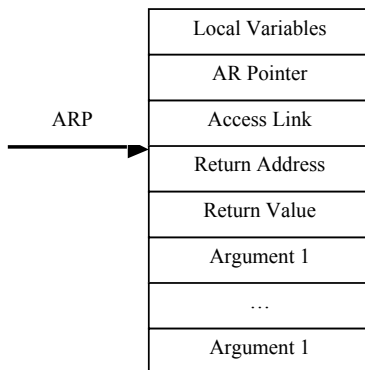


- (c) For the accesses to both *P* and *Q* we only need a single indirection via the access link so using the display does not lead to any improvement in terms of access time. For the procedure *R*, however, we would need 2 links to access the variables `int a`, and `int b` local to the *main* procedure. 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: 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:

- 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
- Explain clearly the values of the access link fields for the instantiation of the procedure P4.



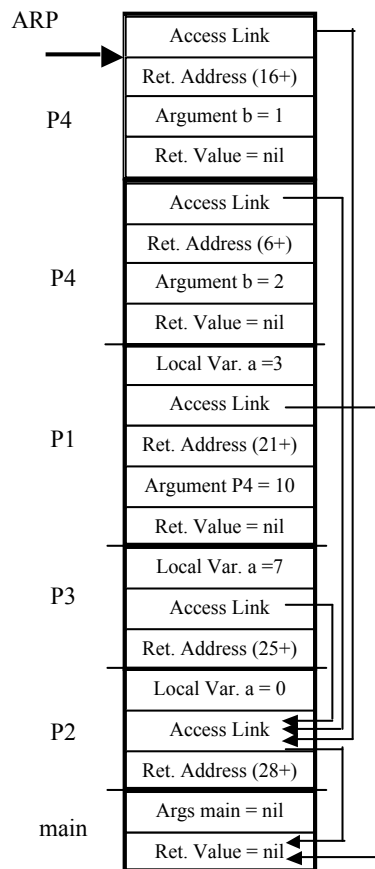
```

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.

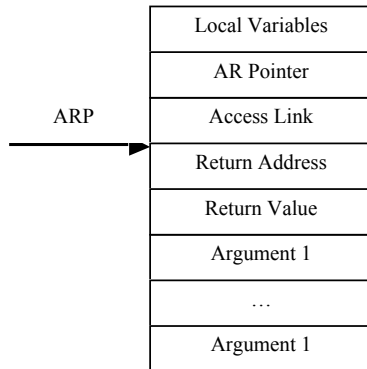


- (b) All access links point to the procedure whose lexical code is the more recent (in terms of active invocation history) and immediately enclosing nesting scope. For the case of P4 the immediate nesting scope of P4 is P2 and thus both recursive invocations need to have their access link entries pointing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

Problem 5: 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.

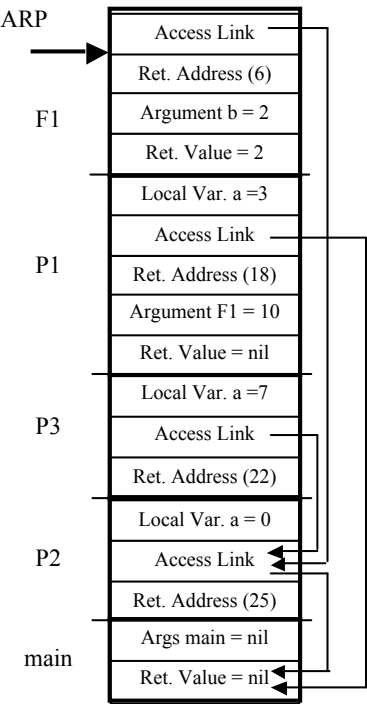


```

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 *)

```

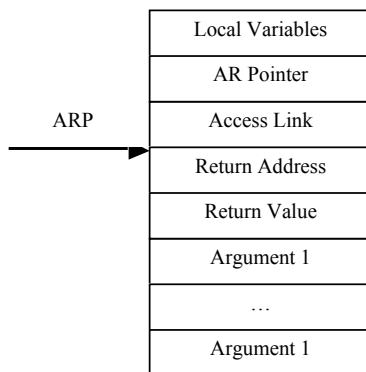
Solution:



Problem 6: 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:

- Draw the call tree starting with the invocation of the main program.
- 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.
- For this particular example would there be any advantage of using the Display Mechanism?



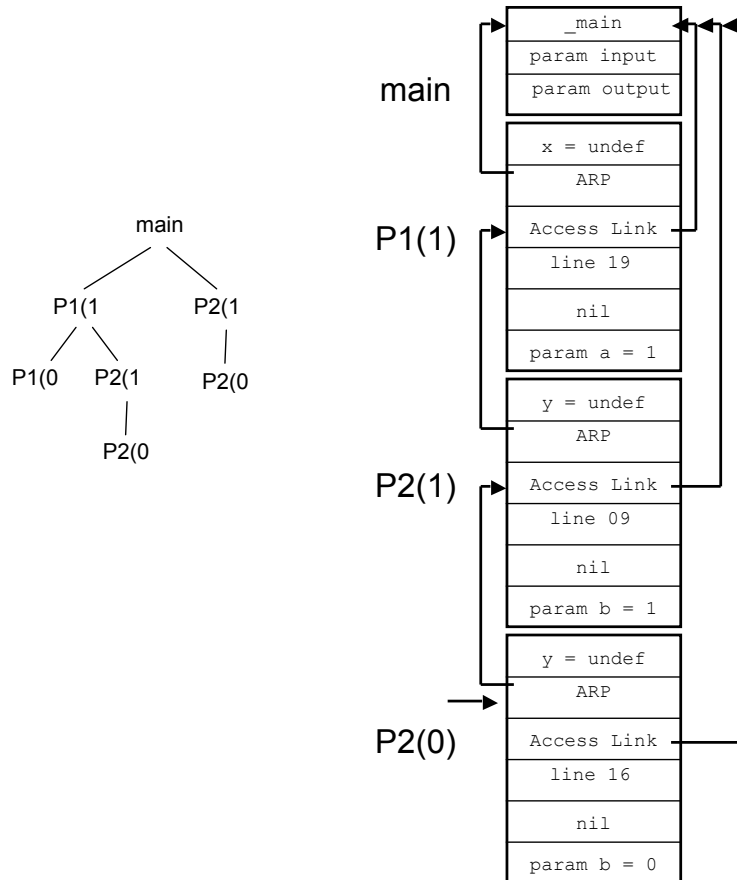
```

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.

```

Solution:

- (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 an second GP or Global Pointer registers. Most architectures do support these two registers in hardware.

Problem 7: 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.

```

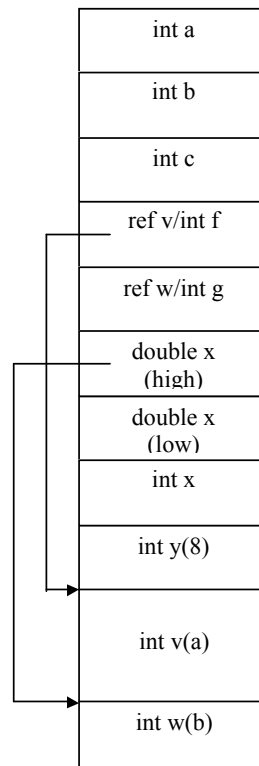
B0: {
    int a, b, c
    ... assign value to a and b
B1: {
    int v(a), w(b)
    double x;
B2: {
    int x, y(8)
    ....
    }
    }
B3: {
    int f, g;
    ....
    }

```

Solution:

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 8: Procedure Storage Organization

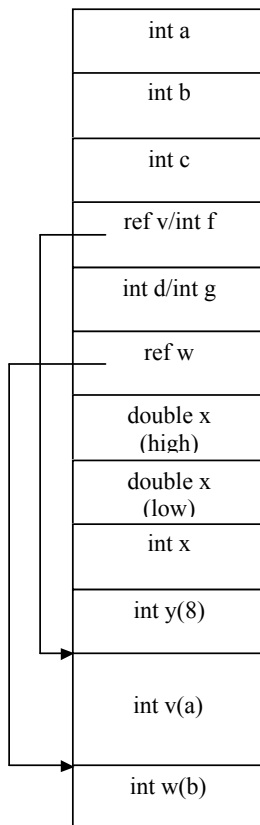
Consider the following C program and assume that the ARs follow the same general layout as discussed in problem 2 above. Assuming that an integer requires 4 bytes and a double data type 8 bytes, derive a layout for the local variables for the AR of the enclosing procedure P detailing the sharing of storage 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 9: Run-Time Environment and Storage Allocation

- Describe the basic structure of an Activation Record (AR).
- Where are AR stored at run-time. Why?
- Why do you need an AR for? What features of a language requires explicit AR and why?
- What is the purpose of the access link? What language mechanism requires its use? Why?
- 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.

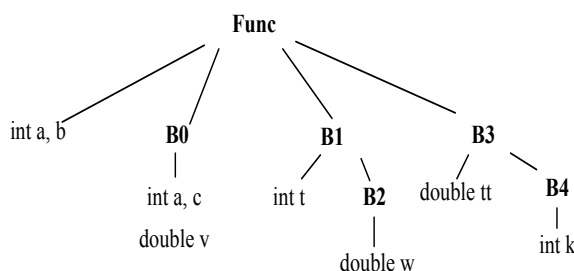
```

void func(){
    int a, b;
    B0: {
        int a, c;
        double v;
    }
    B1: {
        int t;
        B2: {
            double v;
        }
    }
    B3: {
        double tt;
        B4: {
            int k;
        }
    }
}

```

Solution:

- 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.
- 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.
- 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.
- The access link is used to access non-local variables in languages supporting lexical scoping.
- 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 then 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(H)	4 bytes
int c	double w(H)	double tt(L)	4 bytes
double v(H)	double w(L)	int k	4 bytes
double v(L)			4 bytes

Problem 10: 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?

Solution:

- 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 are multiple active instances of the local variables in each function. For the case of functional languages, where the life of some of the functions 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 highlight the currently active call chain in the call-tree. The Access Link is a field in the AR that indicates where to 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.