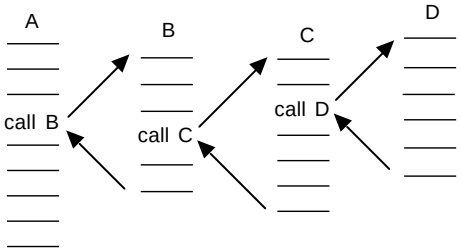
Implementing subprograms

Selected topics only. Material from the textbook:
Sections 9.1, 9.3 (except 9.3.4.2), 9.2(reading assignment).

Contents

The environment in block-structured languages

Subprogram calls are strictly nested:
the **caller** waits until the **callee** has terminated.



Every subprogram activation is represented as an activation record on the activation stack.
The activation record of the callee is placed at the top of stack, directly above the activation record of the caller.

Activation records vary in size. Sizes are usually determined at compile time, unless semidynamic arrays are supported, but every activation record contains the same kind of information about the caller and the callee.

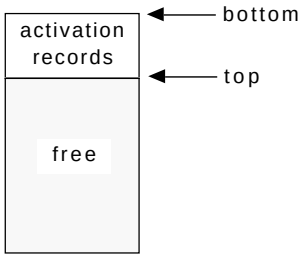
Information about the caller

- A pointer to the caller's activation record (the next record on the stack, but this pointer is necessary to handle varying record sizes); this gives us access to the chain of previous callers up to the main program.
- The return address (what the caller will do after this subprogram call has terminated).
- If this is a function, the address where the function value should be stored.

Information about the callee

- Scoping information—a pointer to the activation record of the enclosing block (this need not be the same unit as the caller).
- Local variables and constants.
- Formal parameters (copies or only pointers).
- Temporary storage (to evaluate expressions).

Memory is allocated to procedure calls in segments of stack storage, as much as is necessary to represent the callee's block.



The compiler generates a *prologue*, next the translation of the subprogram body and finally an *epilogue*.

Prologue: entering a subprogram.

- Get a segment of free stack storage, move the top-of-stack pointer up.
- Put in all data about the caller and the callee.

Epilogue: exiting a subprogram.

- Return a value (if it is a function).
- Remove the segment from the stack, move down the top-of-stack pointer.
- Jump to the return address (continue the caller).

Run-time memory is divided into three parts.

(1) Code area : main program, subprograms.

(2) Data area : the run-time stack (all variables are represented—global variables are local in the activation record of the main program).

(3) Control information :

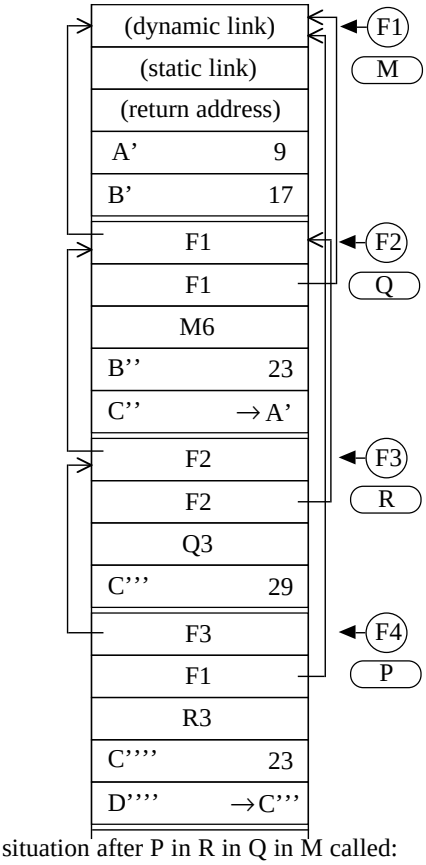
- Current instruction pointer IP indicates the instruction about to be executed.
- Current environment pointer EP shows the activation record of the current block, giving access to local and non-local data.

In the example, we will assume a simple model:
- the pointer to the caller's activation record,
- the pointer to the enclosing block,
- the return address,
- local data (if any),
- actual parameters (if any).

Return addresses will be symbolic—see the boxes on the next page.

The structure of the activation stack

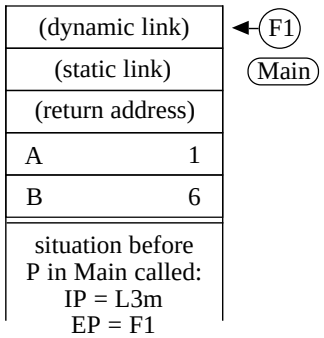
```
      program M( input, output);
         var A, B: integer;

         procedure P( C: integer;
                      var D: integer);
            begin {P}
{P1}           C := C + 2;
{P2}           D := D + 3;
{P3}           writeln('P:', A, B, C, D);
{P4}        end;

         procedure Q( var C: integer);
            var B: integer;

            procedure R( C: integer);
               begin {R}
{R1}              C := 29;
{R2}              P(B, C);
{R3}              writeln('R:', A, B, C);
{R4}           end;

            begin {Q}
{Q1}           B := 23;
{Q2}           R(A);
{Q3}           P(B, C);
{Q4}           writeln('Q:', A, B, C);
{Q5}        end;

         begin {Main}
{M1}        A := 6;
{M2}        B := 17;
{M3}        P(B, A);
{M4}        writeln('M:', A, B);
{M5}        Q(A);
{M6}        writeln('M:', A, B);
{M7}     end.
```

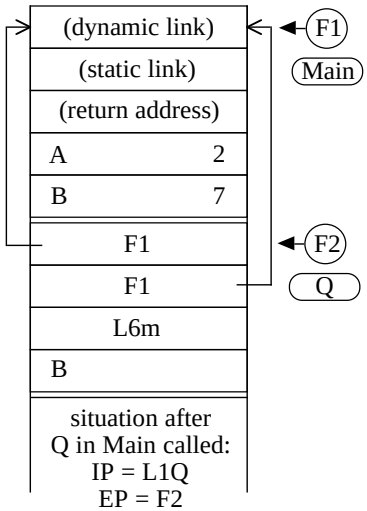



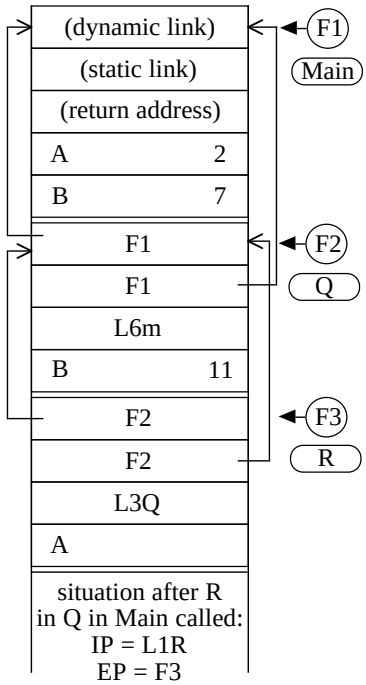
situation after P in R in Q in M called:
IP = P1, EP = F4

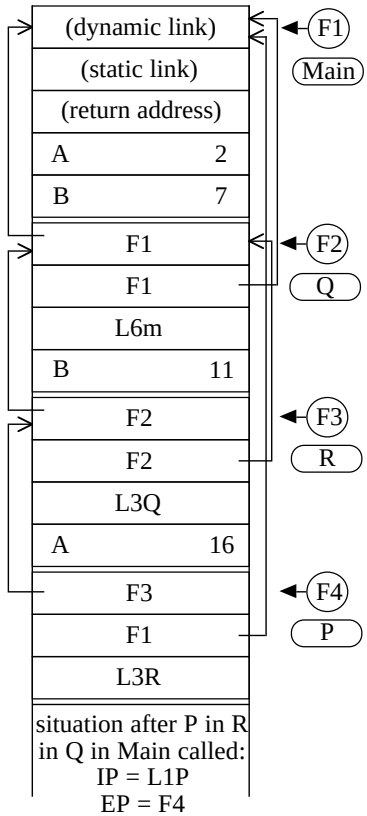
Another example

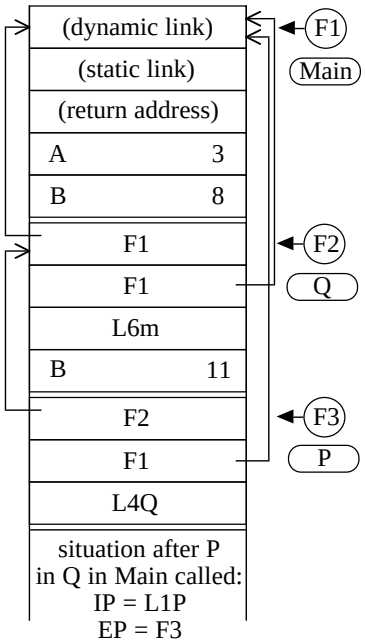
```
program Main;
      var A, B: integer;

      procedure P;
            begin {P}
{L1P}             A := A + 1;
{L2P}             B := B + 1;
{L3P}       end;

      procedure Q;
            var B: integer;

            procedure R;
                  var A: integer;
                  begin {R}
{L1R}                   A := 16;
{L2R}                   P;
{L3R}                   write(A, B);
{L4R}             end;
            begin {Q}
{L1Q}             B := 11;
{L2Q}             R;
{L3Q}             P;
{L4Q}             write(A, B);
{L5P}       end;

      begin {Main}
{L1m}       A := 1;
{L2m}       B := 6;
{L3m}       P;
{L4m}       write(A, B);
{L5m}       Q;
{L6m}       write(A, B);
{L7m} end.
```

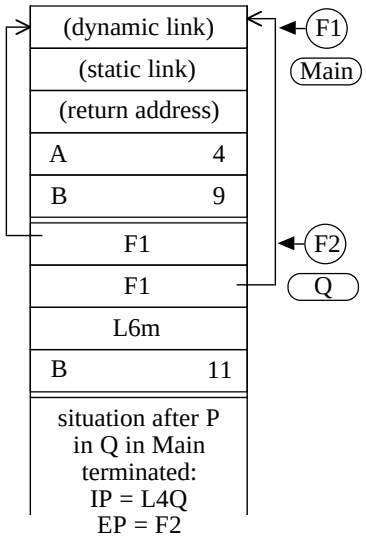













Static chain

Variables represented on the stack are not accessed by name. In a language with static scoping, a variable must, however, be located by moving up the chain of static nesting.

An address of variable V on the stack is composed of two numbers that tell us

- how many activation records up the chain is the record R that contains V,
- how far is V from the beginning of R.

In the situation when Q in Main has been called:

Main.Q.B	(0, 3)
Main.A	(1, 3)
Main.B	(1, 4)

In the situation when P in R in Q in Main has been called:

Main.Q.R.A	unavailable!
Main.Q.B	(1, 3)
Main.A	(2, 3)
Main.B	(2, 4)

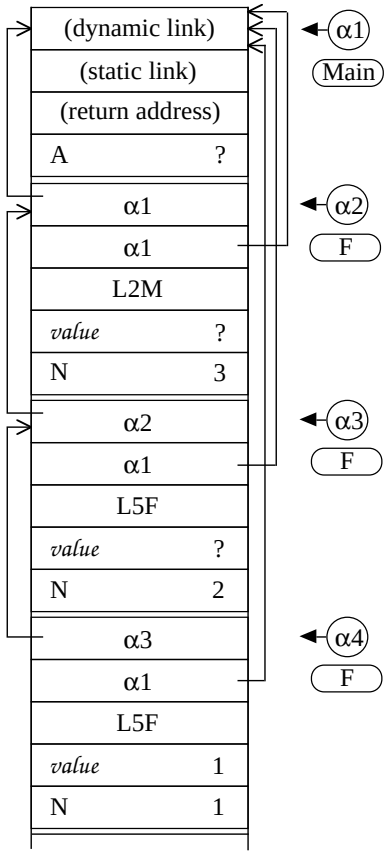
Functions and the stack

```
program Main;
var A: integer;
function F(N: integer): integer;
  begin
    if N <= 1 then
      F := 1
    else
      F :=
        F(N-1)
        * N
  end;
begin
  A :=
    F(3);
  writeln(A)
end.
```

Assigning locations to program fragments must be a little more elaborate...


```
L1F     if N <= 1 then
L2F     value := 1
L3F     goto L7F
L4F     F(N-1)
L5F     →  temp
L6F     value := temp * N
L7F
```

```
L1M     F(3)
L2M     →  temp
L3M     A := temp
L4M     writeln(A)
L5M
```

F in F in F in Main returns with 1: IP = L3F, EP = α4

Variable-length data

Such data (e.g., local arrays) must also be stored on the stack, but we prefer addressing to be determined at compile time.

We use additional indirection in addressing: we separate access mechanisms from pure data.

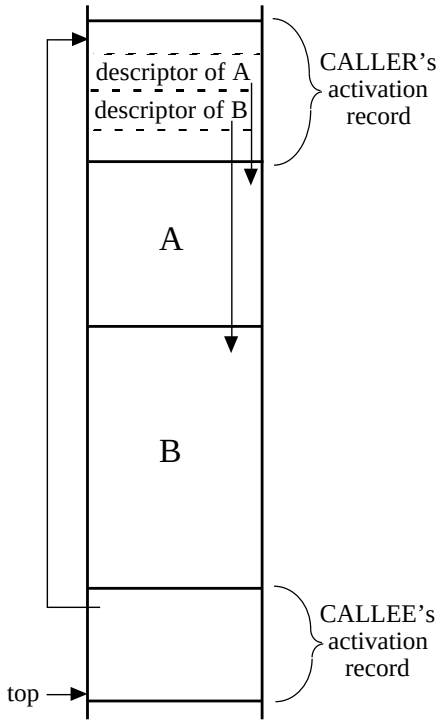
- An array is represented in the activation record by a descriptor.
- Array elements are stored after the caller's activation record, closer to the top of the stack. No size restriction is necessary (other than the amount of free storage in general).

A descriptor contains:

° bounds for the subscripts in the array (they may be needed for run-time checking),

° base—a pointer to the beginning of the data area.

To access an element, find its address: add to base the distance from the beginning (row-major or column-major).



Summary