# RUNTIME ENVIRONMENTS

# TEACHING MATERIAL

# BASIC CONCEPTS & ISSUES

**To decide on the organization of data objects, so that their addresses can be resolved at compile time.**

- **The data objects (represented by variables) come into existence during the execution of the program.**
- **The generated code must refer to the data objects using their addresses, which must be resolved during compilation.**
- **The addresses of data objects depend on their organization in memory. This is, to a great extent, decided by source language features.**

# BASIC CONCEPTS & ISSUES

We shall restrict our study to the following two types of scenarios:

1. ***Fortran like environments:*** By this we mean programming languages, in which

   a. The sizes of objects and their positions in memory are known at compile time.

   b. Recursion is not permitted.

   c. Dynamic creation of data structures is not allowed.

   We call the memory allocation strategy for such languages, ***static allocation.***

2. ***Pascal like environments:*** In which

   a. The sizes of objects and their *relative* positions in memory are known at compile time.

   b. Recursion is permitted.

   We call this ***stack allocation.*** As in Pascal, we shall also allow

   c. Data structures to be created at run time (under program control).

# BASIC CONCEPTS & ISSUES

**To illustrate the influence of source language features on allocation strategy, we shall consider the following examples.**

## EXAMPLE 1

```
      PROGRAM MAIN
      DIMENSION A(100)
      READ *, (A(I), I = 1,100)
      AVGVAL = AVG(A,100)
      STDDEV = STD(A,100)
      PRINT *, AVGVAL, STDDEV
      END
      FUNCTION AVG(B,N)
      DIMENSION B(N)
      SUM = 0
      DO 20 I = 1,N
20    SUM = SUM + B(I)
      AVG = SUM/N
      END
      FUNCTION STD(C,M)
      DIMENSION C(M)
      SUM = 0
      AV = (*) AVG(C,M)
      DO 30 J = 1,M
30    SUM = SUM + (C(J)-AV)**2
      STD = SQRT(SUM)/M
      END
```

## EXAMPLE 2

```
type t = ^integer;
var z: integer;
procedure  factorial (var r: t; x: integer);
   var y: t;
   begin if x = 0 then r^ := 1
        else begin
                new(y);
                factorial(y,x-1);
           (#)  r^ := y^ * x; writeln(z)
             end
   end;
procedure start;
var p:t, z:integer;
    begin
      z := 4; new(p);
      factorial(p, z)
(*) end;
begin (*main*)
z:=2; start;
end.
```

**(*) and (#) have been used to mark program points for future reference.**

# BASIC CONCEPTS & ISSUES

The organization of data is largely determined by answers to questions like:

1. ***Does the language permit recursion?*** There may be many incarnations of `factorial` active at the same time, each with its copy of local variables and parameters which must be allocated storage. Moreover, the number of such incarnations will be decided at run-time.

2. ***What are the parameter passing mechanisms?*** Inside `factorial`, the access mechanisms for the formal parameters `r` and `x` have to be different. This is because `r` is *call-by-reference*, whereas `x` is *call-by-value*.

3. ***How does a procedure refer to the non-local names?*** The rules of static scoping decide that the `z` being referenced in `factorial` refers to the `z` in the main program. Therefore, apart from the local variables and formal parameters, `factorial` should be able to access this variable at run-time.

4. ***Does the language permit creation of dynamic data structures?*** Space must be created at run-time, each time `new(y)` is executed.

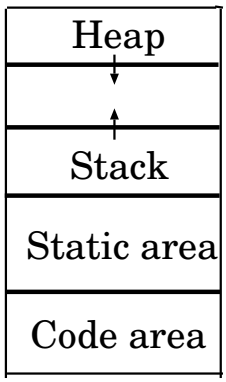
# BASIC CONCEPTS & ISSUES

## INITIAL DIVISION OF MEMORY

The memory is initially divided into:

1. *Code area:* Containing the generated target code.
2. *Static area:* Containing data whose absolute addresses can be determined at compile time. As examples:
   - In languages like Fortran, the addresses of all variables can be determined statically.
   - In languages like Pascal, the addresses of the local variables of the outermost procedure (for example `z` in `main`) can be determined at compile time.
3. *Stack area:* For data objects of procedures which can have more than one incarnations active at the same time.
4. *Heap area:* For dynamically allocated data (like objects pointed to by pointer types.)

Since the sizes of the stack and the heap are not known, they are arranged to grow in opposite directions.



# ACTIVATION RECORDS

An activation record contains the information required for a single activation of a procedure. Activation records are held in the static area for languages like Fortran and in the stack area for languages like Pascal. Typically space must be provided in the activation record for:

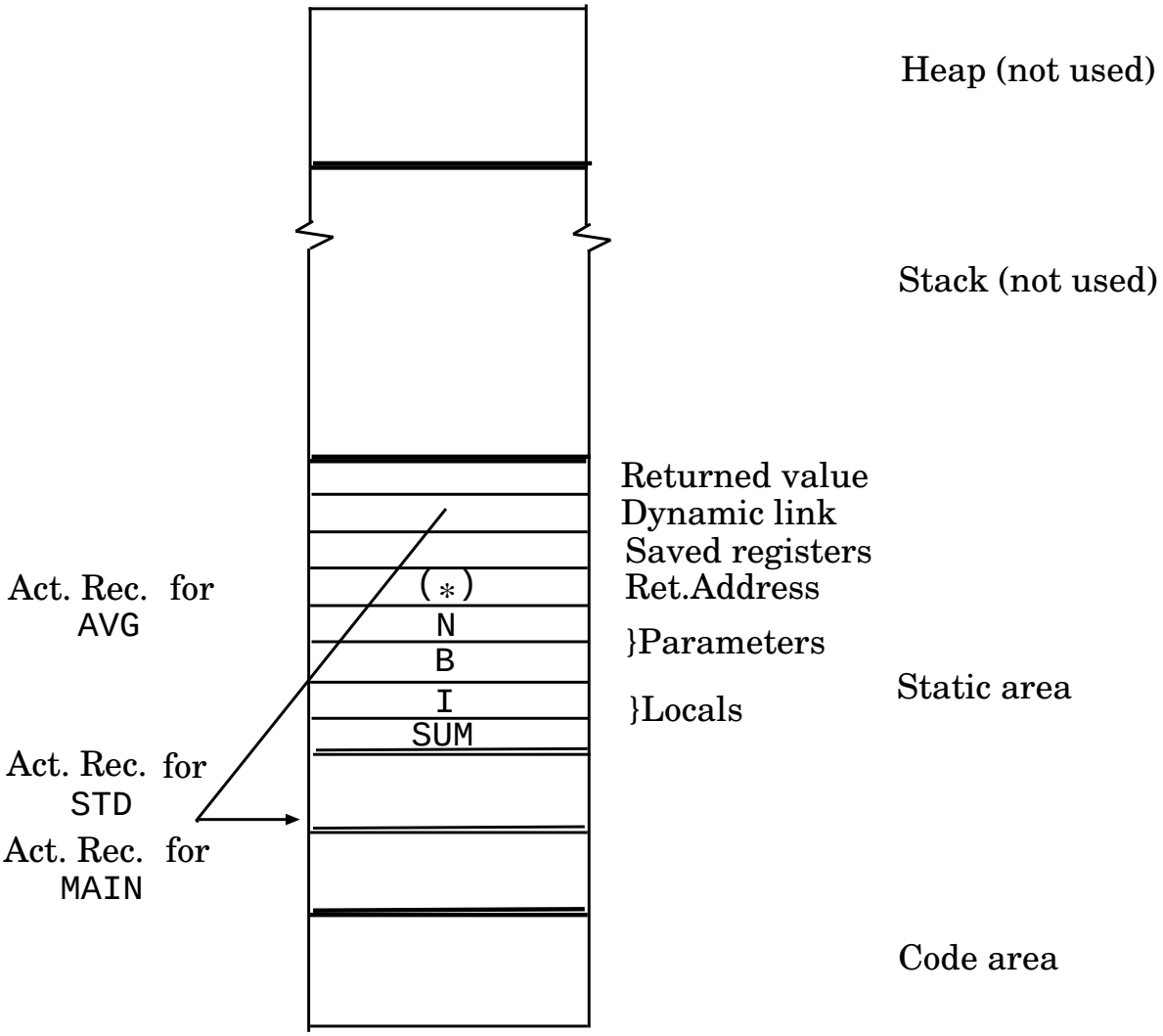
1. ***Local variables:*** Part of the local environment of the procedure.
2. ***Parameters:*** Also part of the local environment.
3. ***Return address:*** To return to the appropriate control point of the calling procedure.
4. ***Saved registers:*** If the called procedure wants to use the registers used by the calling procedure, these have to be saved before and restored after the execution of the called procedure.
5. ***Static link:*** For accessing the non-local environment. The static link of a procedure points to the latest activation record of the immediately enclosing procedure. Static links may be avoided by using a display mechanism.

   A static link is not required for Fortran like languages.
6. ***Dynamic link:*** Pointer to activation record of calling procedure.
7. ***Returned value:*** Used to store the result of a function call.

# ACTIVATION RECORDS

## EXAMPLES OF ACTIVATION RECORDS

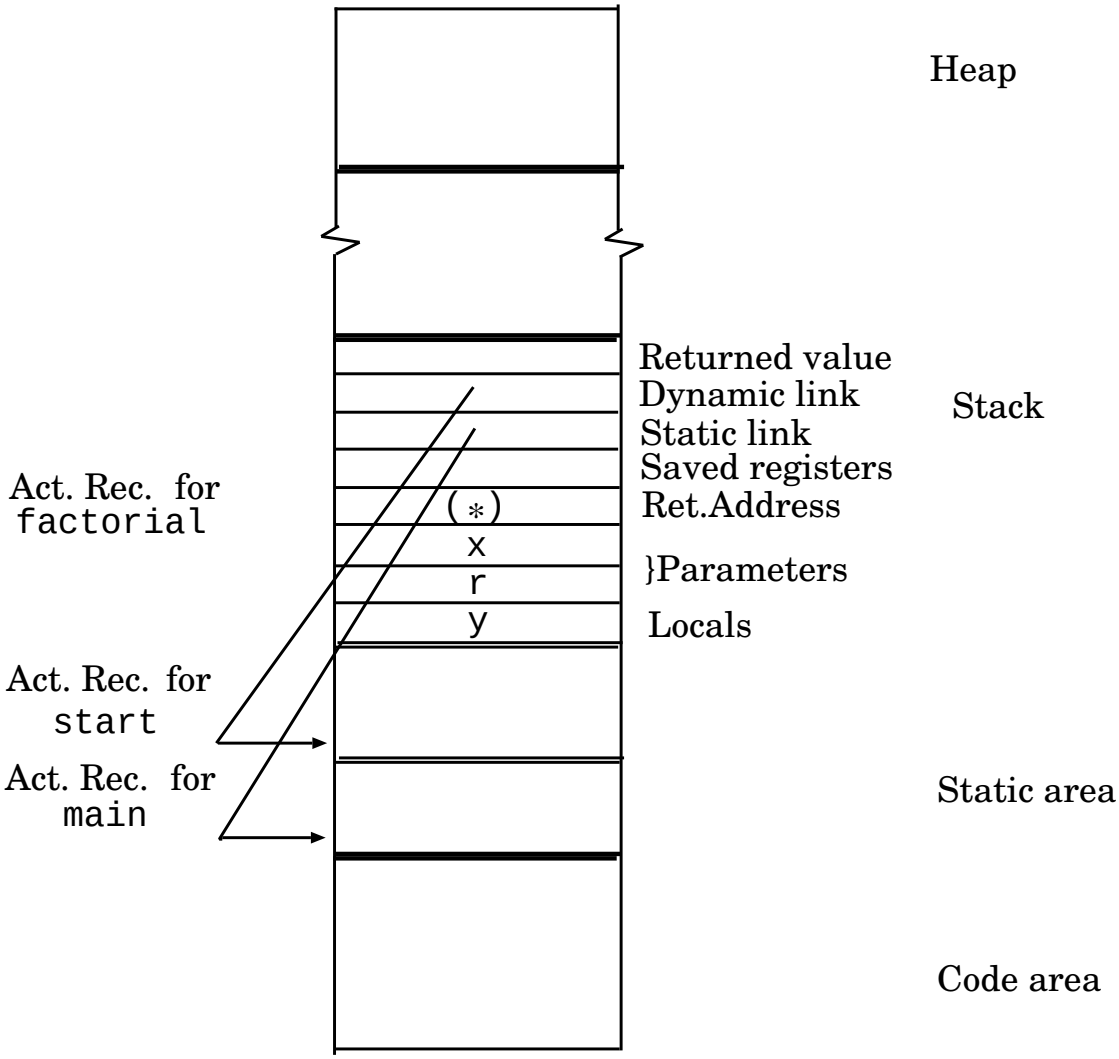
For the Fortran program shown earlier, the activation record of `AVG` during its invocation from `STD` is shown below.



# ACTIVATION RECORDS

## EXAMPLES OF ACTIVATION RECORDS

For the Pascal program, the activation record of `factorial` during its first invocation is shown below.

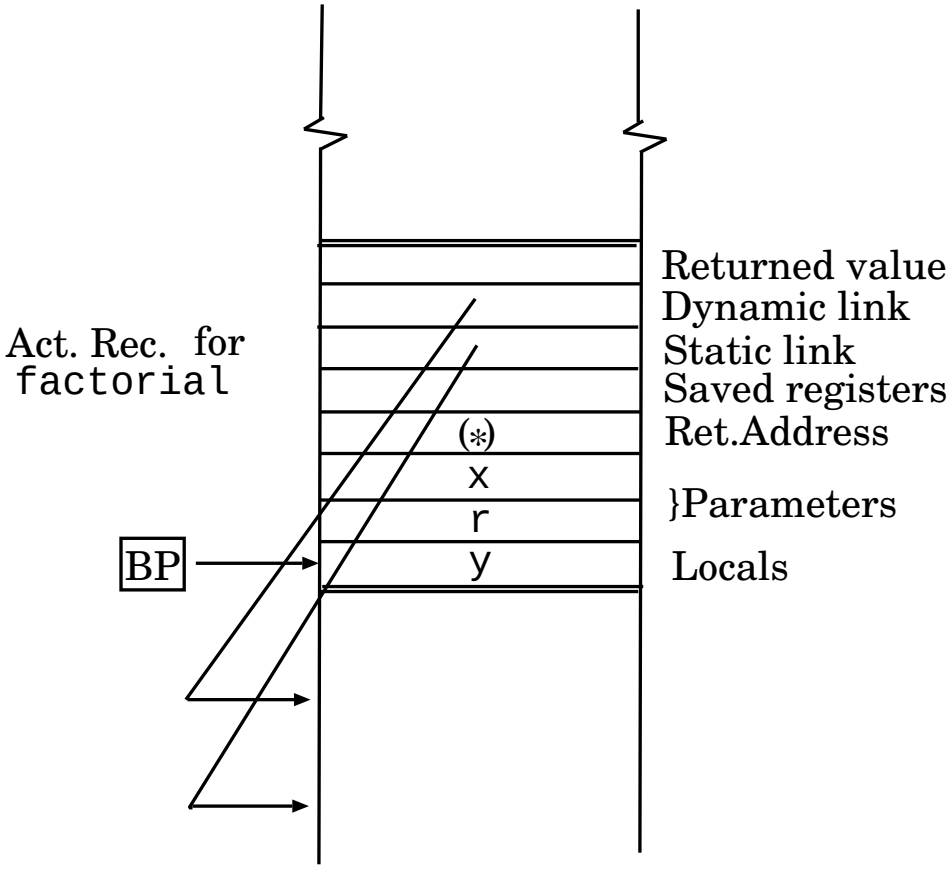



This is one of many possible organizations of an activation record. We shall see a different organization later.

# ACTIVATION RECORDS

## ACCESSING INFORMATION IN ACTIVATION RECORD

Within an activation record, any information is referred to using an offset relative to a pointer (BP) pointing to the base of the activation record. Thus the address of the local variable `y` is BP (contents of register BP) and the parameter `x` has the address BP+2.




Some machines use a dedicated register for holding the base pointer.

# ACTIVATION RECORDS

## ACCESS TO NON-LOCAL VARIABLES

**An outline of a program fragment with a non-trivial nesting structure:**

```
procedure S;
  var a,x: _
     procedure R;
        var i: _
        procedure T;
           var m,n: _
           body of T;
        body of R;
     procedure E;
        body of E;
     procedure Q;
        var k,v: _
        function P(y,z:_): _
            var i,j: _
            body of P;
        body of Q;
  body of S;
```

# ACTIVATION RECORDS

Assuming that the language uses ***static scoping*** rules, the names which can be referred to (visible) at a program point are

1. The names declared locally in the procedure, of which this program point is a part.
2. The names of all enclosing procedures.
3. The names declared immediately within such procedures.

An additional rule is that a name declared in more closely nested procedure overrides the same name declared in an outer procedure.

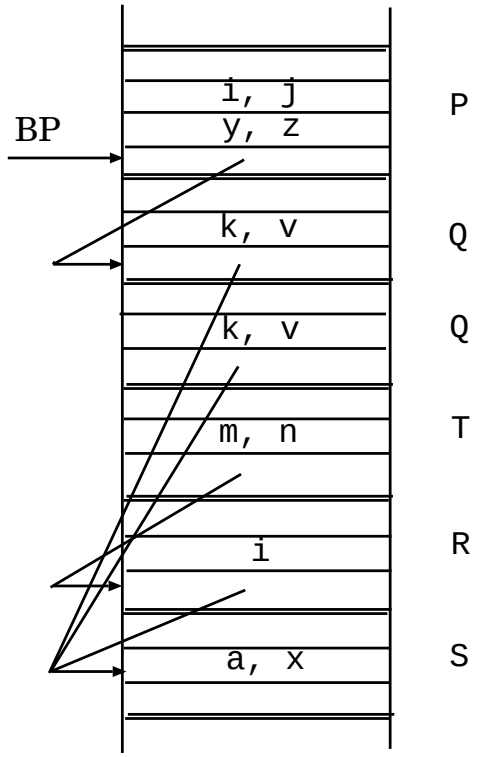
For example, the names visible in the body of `T` are:

1. `T`, `m`, `n` – names declared in `T`.
2. `R` and `S` – enclosing procedure names.
3. `i`, `a`, `x`, `E` and `Q` – names declared immediately within `R` and `S`.

# ACTIVATION RECORDS

## STATIC LINK

**The non-local variables of a procedure are accessed by chasing a chain of pointers starting from the base of the activation record of the procedure.**
**Consider a call `S` → `R` → `T` → `Q` → `Q` → `P`.**




***Note:***

- `P` finds its local variables in its own activation record.
- To access the variables of `Q`, it goes down the static link once.
- To access the variables of `S`, it goes down the static link twice.

# ACTIVATION RECORD

## SETTING UP THE STATIC LINK

**How does one set up the static link during a procedure call? It depends on the relative *levels* of the calling and called procedure.**

## LEVEL OF A PROCEDURE

**The level of a procedure is calculated using the following rules:**

1. **The level of the main program is 1.**
2. **If the level of an enclosing procedure is $l$, then the level of the enclosed procedure is $l + 1$.**

## EXAMPLE

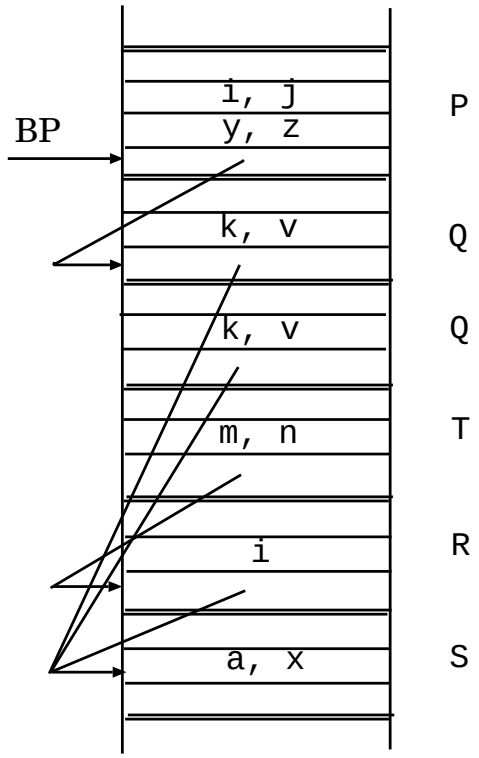
**The level of procedures in the previous example (ignoring the procedures outside `S`) is:**

`S` — **1**
`R` — **2**
`T` — **3**
`E` — **2**
`Q` — **2**
`P` — **3**

# ACTIVATION RECORD

## RULE FOR SETTING STATIC LINK

**If the calling procedure is at level $l_1$ and the called procedure is at level $l_2$, then starting from the activation record of the calling procedure the static link is traversed $l_1 - l_2 + 1$ times. The static link of the called procedure should point to the base of the activation record thus reached.**



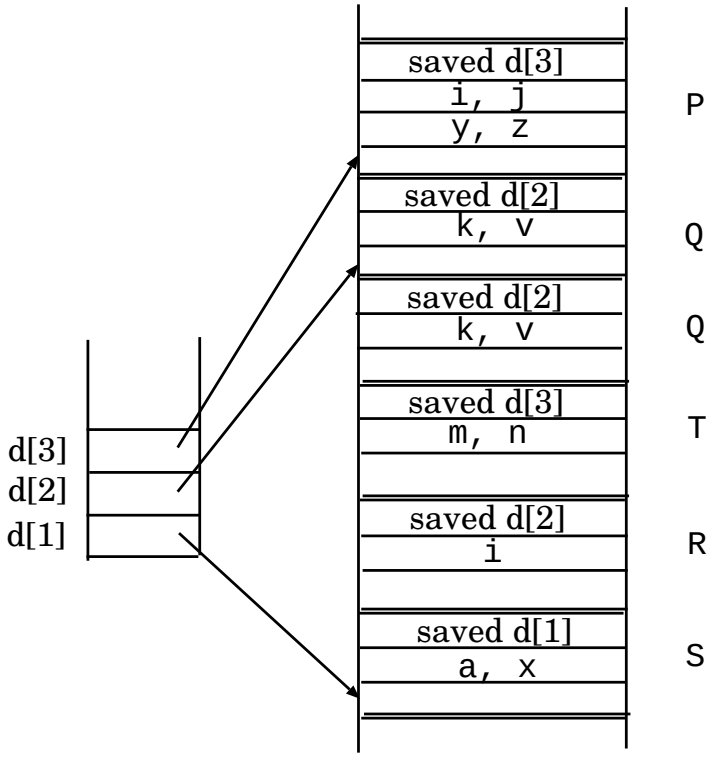

**When procedure `T` (level 3) calls procedure `Q` (level 2), *by traveling down the static link twice (3-2+1), the base of* `S` *is reached. This is where the static link of* `Q` *should point to.***

# ACTIVATION RECORD

## DISPLAYS

As we have seen before, accessing a non-local variable involves going down the static link, possibly more than once. This process can be made more efficient if, instead of chasing pointers, one uses an *array* of pointers to activation records. This array is called a ***display.***

In the previous example, for the calling sequence `S` $\rightarrow$ `R` $\rightarrow$ `T` $\rightarrow$ `Q` $\rightarrow$ `Q` $\rightarrow$ `P`, the display would look like:




From the body of `P`, the base of the activation record of the (unique) procedure at level 2 (in this case `Q`) would be given by d[2].

# ACTIVATION RECORD

## MANIPULATING DISPLAYS

When setting up d[i] to point to the base of the current procedure at level i, the old d[i] is saved within the activation record of the procedure itself.

While new activation record of a procedure (at nesting depth) is being set up:

1. Save the value of d[i] in the new activation record.
2. Set d[i] to point to the new activation record.

Just before the activation of the procedure ends:

1. Reset d[i] to its old value.

# PARAMETER PASSING

The common methods to associate an actual parameter with a formal parameter are:

1. ***Call by value:*** The calling procedure evaluates the actual parameter and places the resulting value in the activation record of the called procedure.

2. ***Call by reference:*** The actual parameter is a variable. The calling procedure places the address of this variable in the activation record of the called procedure. A reference to the formal parameter, then, becomes an indirect reference through the address passed to the called procedure.

If a procedure `P` declared as

```
P(x:_, var y:_)
```

is called as

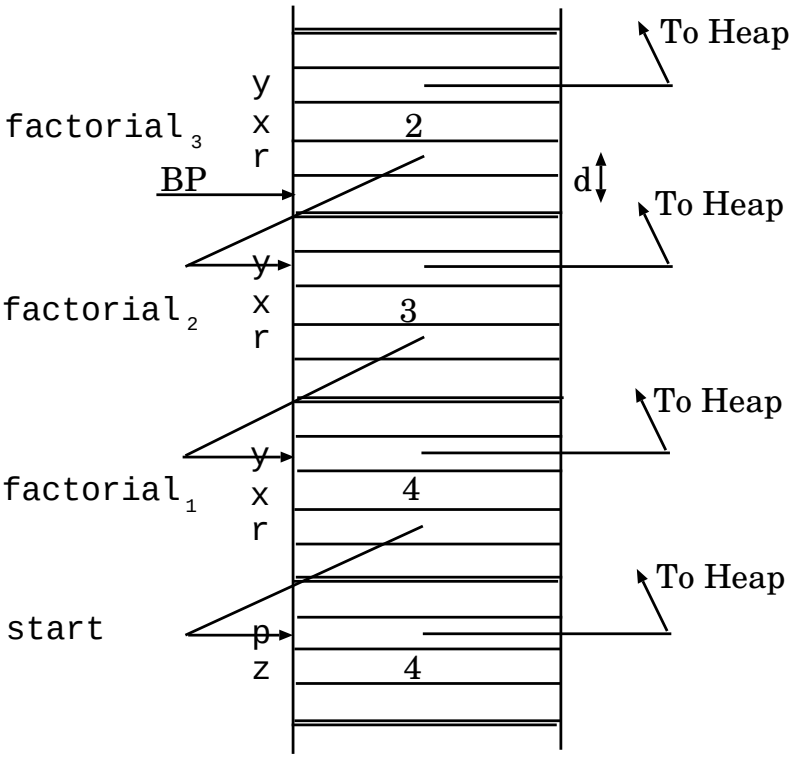
```
P(e,a)
```

where `e` is an expression and `a` a variable, then `e` is evaluated and the ***value*** of `e` is placed in the location for `x` in the procedure `P`. Whereas the ***address*** of `a` is placed in the location for `y`. Inside `P`, any reference to `x` translates to a reference to the location for `x`, while a reference to `y` translates to an indirect reference to `a` through the address in the location for `y`.

# PARAMETER PASSING

## EXAMPLE

In the `factorial` example, the parameters are passed in the manner shown:




When $\texttt{factorial}_2$ calls $\texttt{factorial}_3$, the value of `x-1` (2 in this case) is evaluated and stored in the location for `x` in $\texttt{factorial}_3$. However, the address of `y` in $\texttt{factorial}_2$ is stored in the location for `r` in $\texttt{factorial}_3$.

A reference to `x` now translates to *(BP+$d_x$) while a reference to `r` translates to *(*(BP+$d_r$)), where $d_x$ and $d_r$ are the offsets of `r` and `x` in the activation record.

# PARAMETER PASSING

Some other methods of parameter passing are:

3. *Copy restore:* The actual parameters are evaluated, and the values passed to the called procedure. After execution, the current values of the formal parameters are copied back into the addresses of the actual parameters, if it is meaningful to do so. ***Note that both these activities are done by the caller.***

4. *Call by name:* The effect of the call is that of substituting procedure body for the procedure call, with the actuals literally substituted for the formals. Name clashes, if any, are resolved by renaming.

   This strategy is actually implemented by passing a closure consisting of:

   a. A pointer to the code to evaluate the actual parameter.

   b. The environment to resolve accesses to the variables in the actual parameter.

# PARAMETER PASSING

It would be appropriate to compare call by reference with the copy restore method. The second method seems to be more secure than the first, particularly in Fortran like contexts.
Consider the Fortran subroutine shown below:

```
SUBROUTINE SWITCH (N)
N = 3
RETURN
END
```

Assuming that the parameter passing mechanism is call by reference, if this subroutine is called as `CALL SWITCH(2)`, then (depending on the implementation) it may have the unexpected side effect of modifying the data area for literals.
However, if the parameter passing mechanism is copy restore, then the caller would know that the subroutine has been called with a constant. Therefore it would not copy back the modified actual parameter into the formal parameter.

# COMPILING PROCEDURE CALLS

During a procedure call, there are many possible ways of dividing the bookkeeping work between the caller and the callee. One possible way is:

During a procedure call

*Caller*

1. Makes space on the stack for a return value.
2. Puts the actual parameters on the stack.
3. Sets the static link.
4. Jumps to the called procedure. Return address is saved on the stack.

*Callee*

1. Sets the dynamic link.
2. Sets the base of the new activation record.
3. Saves registers on the stack.
4. Makes space for local variables on the stack.

Note that if we follow this sequence, the base pointer does not actually point to the base of the activation record, but somewhere in the middle. The actual parameters are now referenced using a negative offset, and the local variables with a positive offset.

During a return

*Callee*

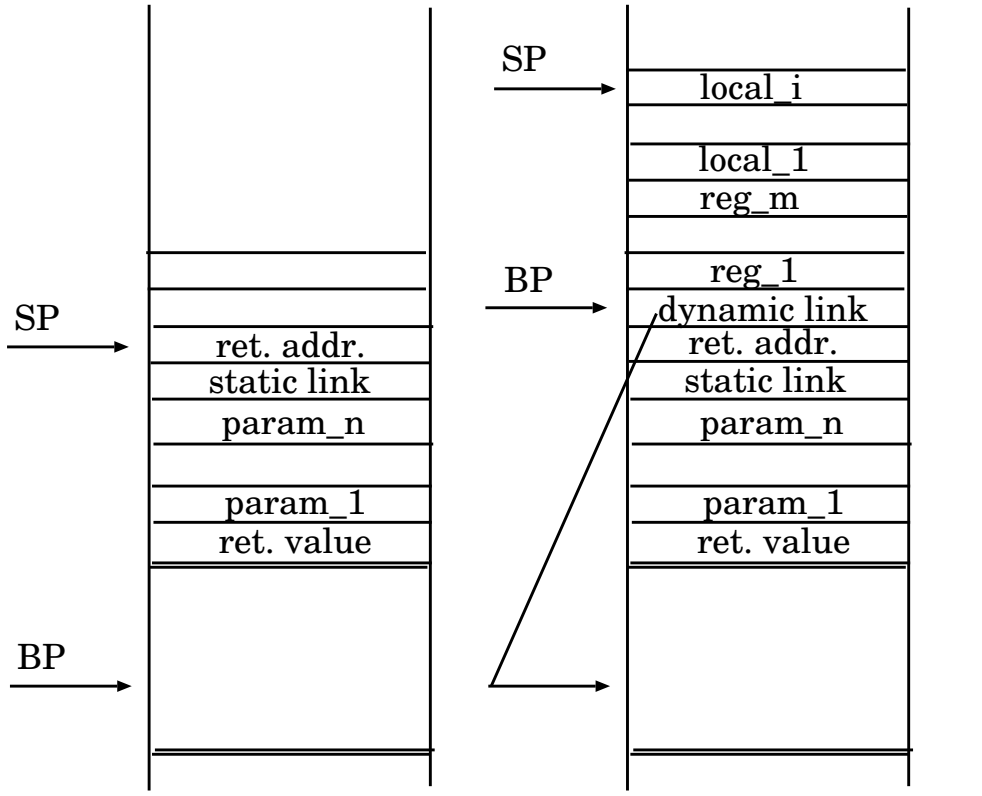
1. Restores registers.
2. Sets the base pointer to the activation record of the calling procedure.
3. Returns to the caller.

*Caller*

1. Restores stackpointer to the location containing the returned value.

# COMPILING PROCEDURE CALLS

## EXAMPLE

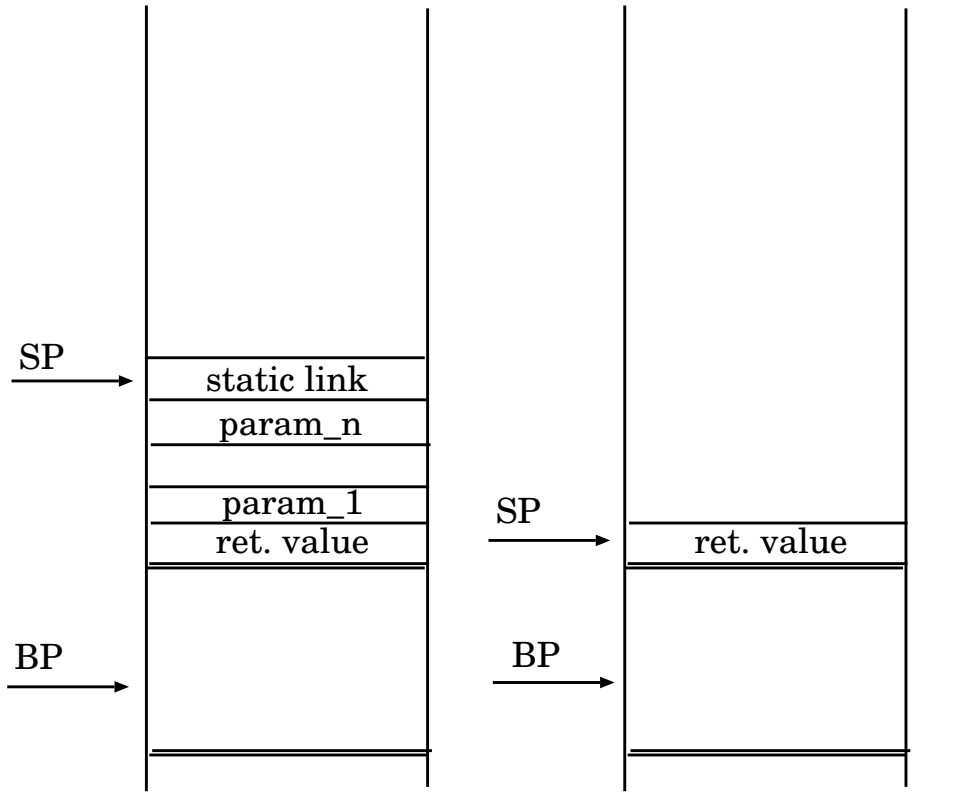



Procedure call:
after caller's activities

Procedure call:
after callee's activities

# COMPILING PROCEDURE CALLS

## EXAMPLE




Procedure call:
after callee's activities

Procedure call:
after caller's activities

# COMPILING PROCEDURE CALLS

## MACHINE SUPPORT FOR COMPILATION

Modern processors have instructions which facilitate compilation. For example the Motorola 68000 processor has:

1. LINK BP, #disp:
   - Pushes the base pointer on the stack.
   - Sets the base pointer to the top of the stack.
   - Makes disp amount of space on the top of the stack.

   disp is usually negative because, in 68000, the stack grows towards decreasing memory locations.
2. UNLK BP:
   - Resets the stack pointer to the base pointer.
   - Restores the base pointer of the caller.
3. JSR P: Jumps to the procedure P, after having pushed the return address on the stack.
4. MOVEM D0–D7, (SP)/ MOVEM (SP), D0–D7: Saves/restores registers D0–D7 on/from the stack.
5. PEA BP + x: Pushes the effective address of the location BP + x. Useful for passing parameters by call-by-reference.

# COMPILING PROCEDURE CALLS

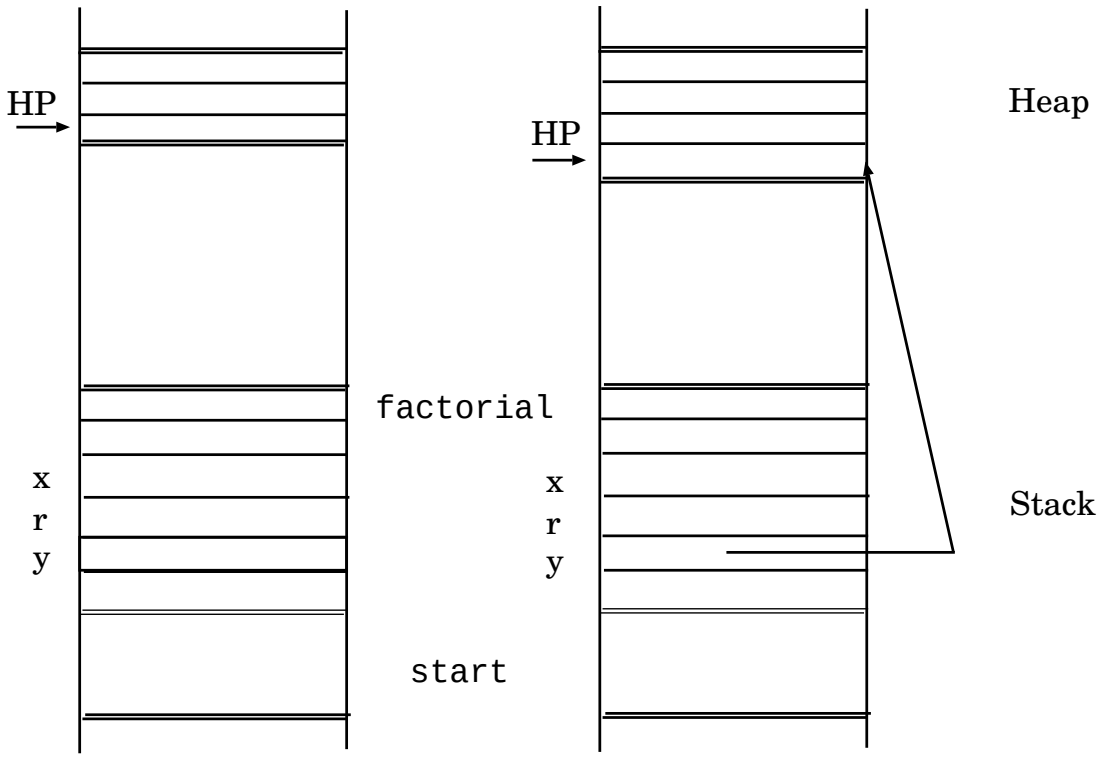
**Shown below are the relevant parts of the M68000 code generated for the recursive call to `factorial`. It is assumed that**

1. **The current values of the variables reside in their home locations.**
2. **No registers need to be saved.**
3. **Each data item requires one location.**
4. **No space needs to be allocated for a return value.**

```
factorial: LINK BP, #-1     /* for y */
           ...              /* code for call to factorial */
           PEA BP+1         /* push address of y */
           MOVE 3(BP), R1   /* push value of x-1 */
           SUB #1, R1
           MOVE R1, -(SP)
           MOVE BP, R1    /* set up static link */
           MOVE #1, R2
     LOOP: CMP R2, 0
           BEQ END
           MOVE 2(R1), R1
           SUB #1, R2
           JMP LOOP
      END: MOV R1, -(SP)
           JSR FACTORIAL
           ADD #3 SP        /* clean up act. rec. of callee */
           ...
           UNLK BP
           RSR
```

# HANDLING DYNAMIC DATA STRUCTURES

A pointer variable `p` is allocated memory on the stack. On execution of a `new(p)`, memory is allocated from the heap and the home location of the pointer variable (on the stack) is made to point to the allocated location in the heap. Any reference to `p^` is an indirect reference to the heap location through the address on the stack.

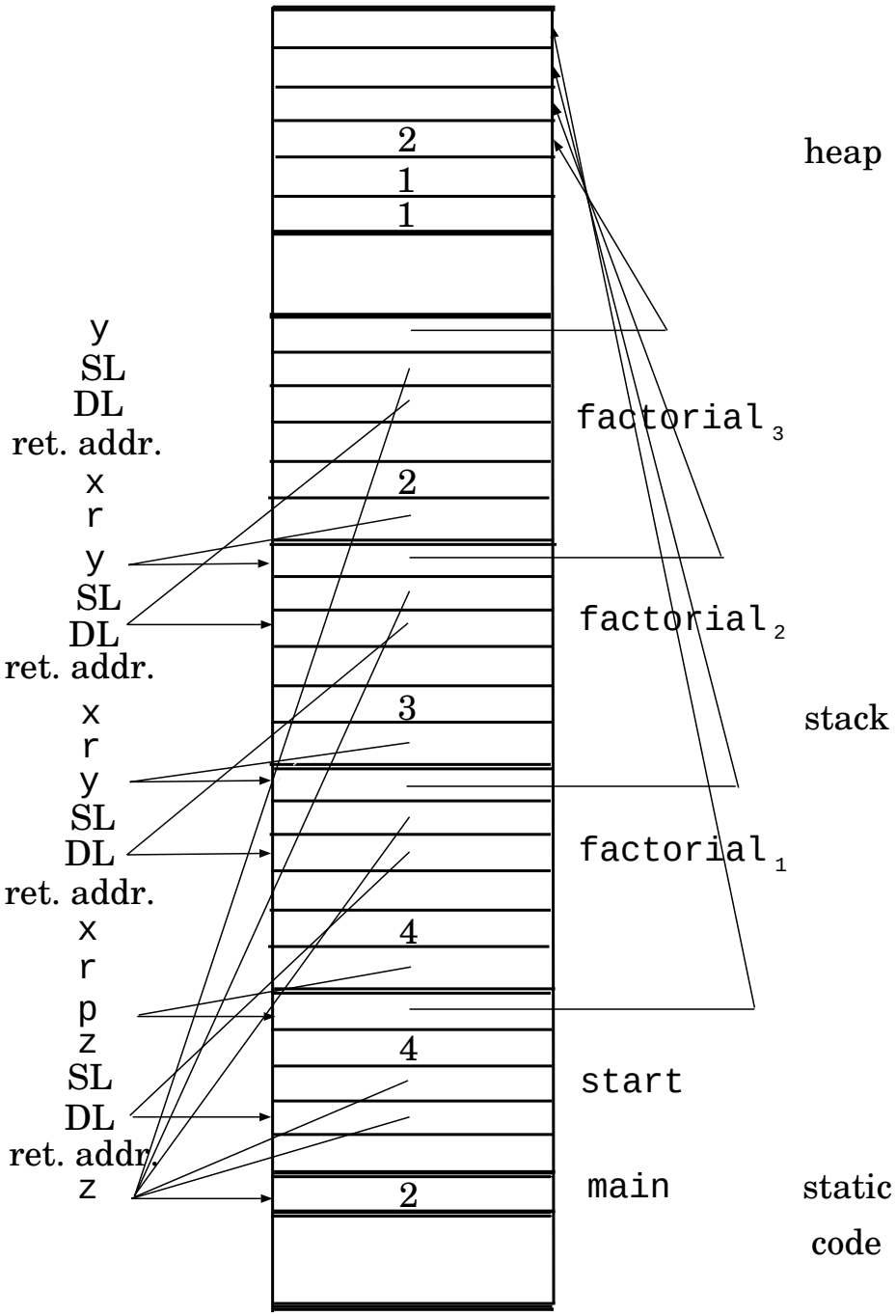



This example shows the memory configuration before and after the execution of `new(y)` in `factorial`. A new location has been released from the heap and the contents of `y` has been made to point to it.

On running out of heap space, the run-time environment might employ the services of a *garbage collector*.

# CONCLUSION

**We summarize by showing the entire memory after completion of the third call to `factorial`.**



# GRADED EXERCISES

1. **What is printed by the the program shown below, assuming (a) call-by-value, (b) call-by-reference, (c) copy-restore linkage (d) call-by-name?**

```
program main(input,output);
   procedure p(x, y, z);
      begin
         y := y + 1;
         z := z + x
      end;
   begin
      a := 2;
      b := 3;
      p(a+b, a, a);
      print a
   end.
```

2 **Some languages like C and Algol allow a programmer to define blocks as shown below.**

   a. **How would you implement blocks? In particular, for this program, how would you access the variables at (*) at runtime?**

   b. **Can you allocate storage for all the blocks in a procedure at the point of entering the procedure body? If so, describe how?**

```
main()
  {int a = 0;
   int b = 0;
     {int b = 1;
        {int a = 2;
         printf(``%d\n'', a, b);}
        {int b = 3;
   (*)   printf(``%d\n'', a, b);}
      printf(``%d\n'', a, b);}
   printf(``%d\n'', a, b);}
```

**3. Consider the Pascal program shown below:**

```
var x : integer;
   procedure p(x: integer; var y:integer);
      procedure q(x: integer; var y : integer);
         procedure t(x: integer; var y : integer);
            var r : integer;
            begin (*t*)
               r := y;
               p(x-1,r)
            end;   (*t*)
         begin (*q*)
            t(x,y)
         end;  (*q*)
      begin (*p*)
         if x = 0 then x := 1 else q(x,y)
      end   (*p*)
```

```
begin (*main*)
   x := 2; p(2,x)
end.  (*main*)
```

a. **Assuming an implementation using static links, show the runtime stack when it has grown to its maximum height.**

b. **Generate M68000 code for the program.**

c. **Repeat a. using display instead of static links.**