Run Time Environments

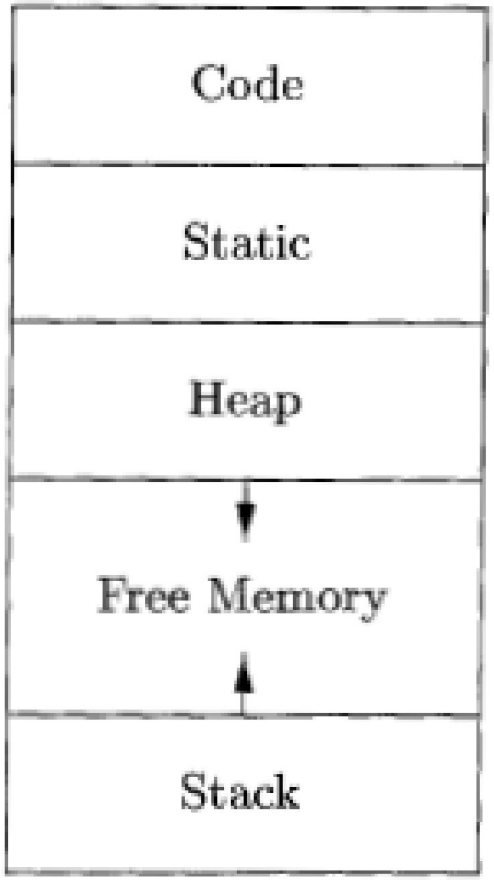
- Compiler assumes its environment in which the target program is executed.
- It should know how to coordinate with OS to get memory or to get services

Storage Organization

- Target program runs in its own logical address space.
  - Each program value has a location.
- The management and organization of this logical address space is shared between the compiler, the OS, and the target machine.
- How the logical address space is physically realized is done by the OS.
  - May be the physical memory is spread throughout the memory, interleaving memory of other programs.

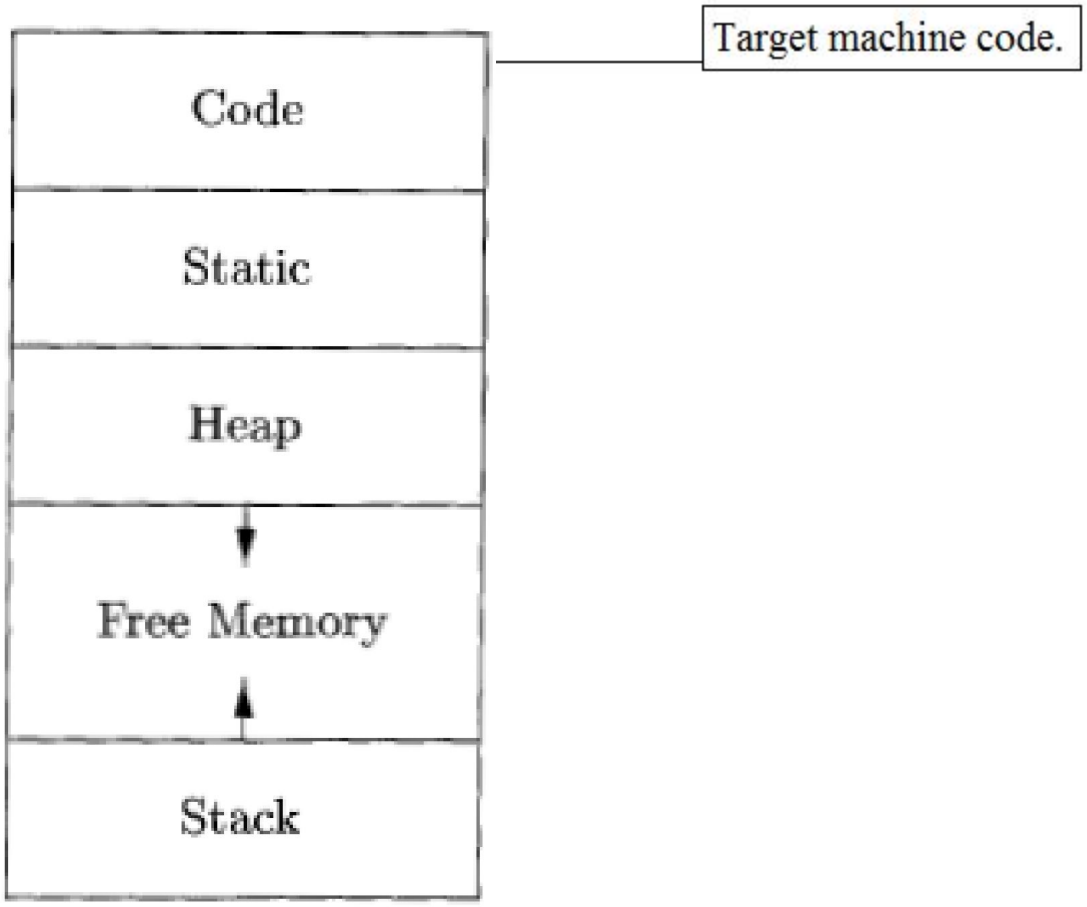
Storage Organization

The memory organization of a program at run-time :



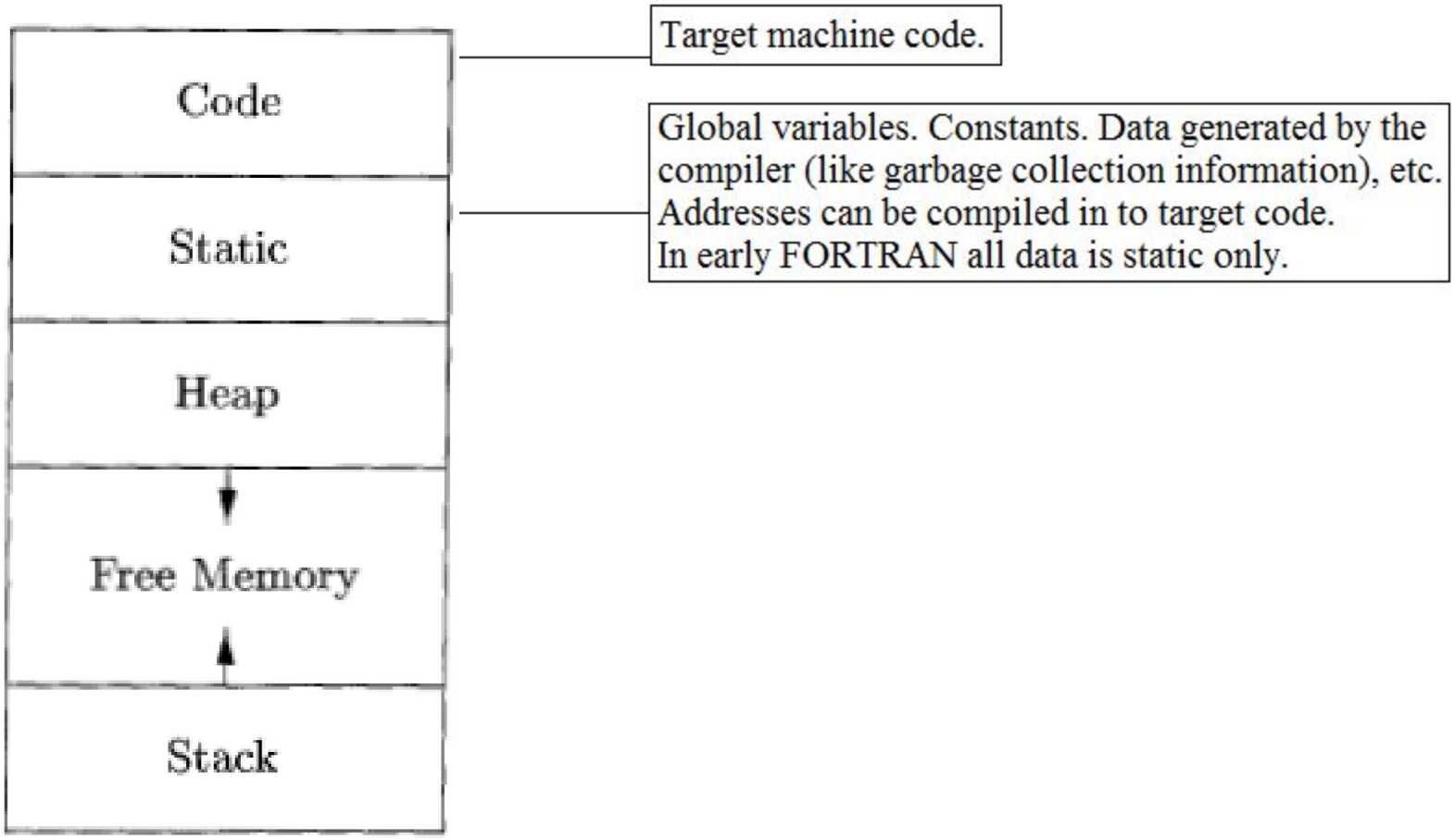
Storage Organization

The memory organization of a program at run-time :



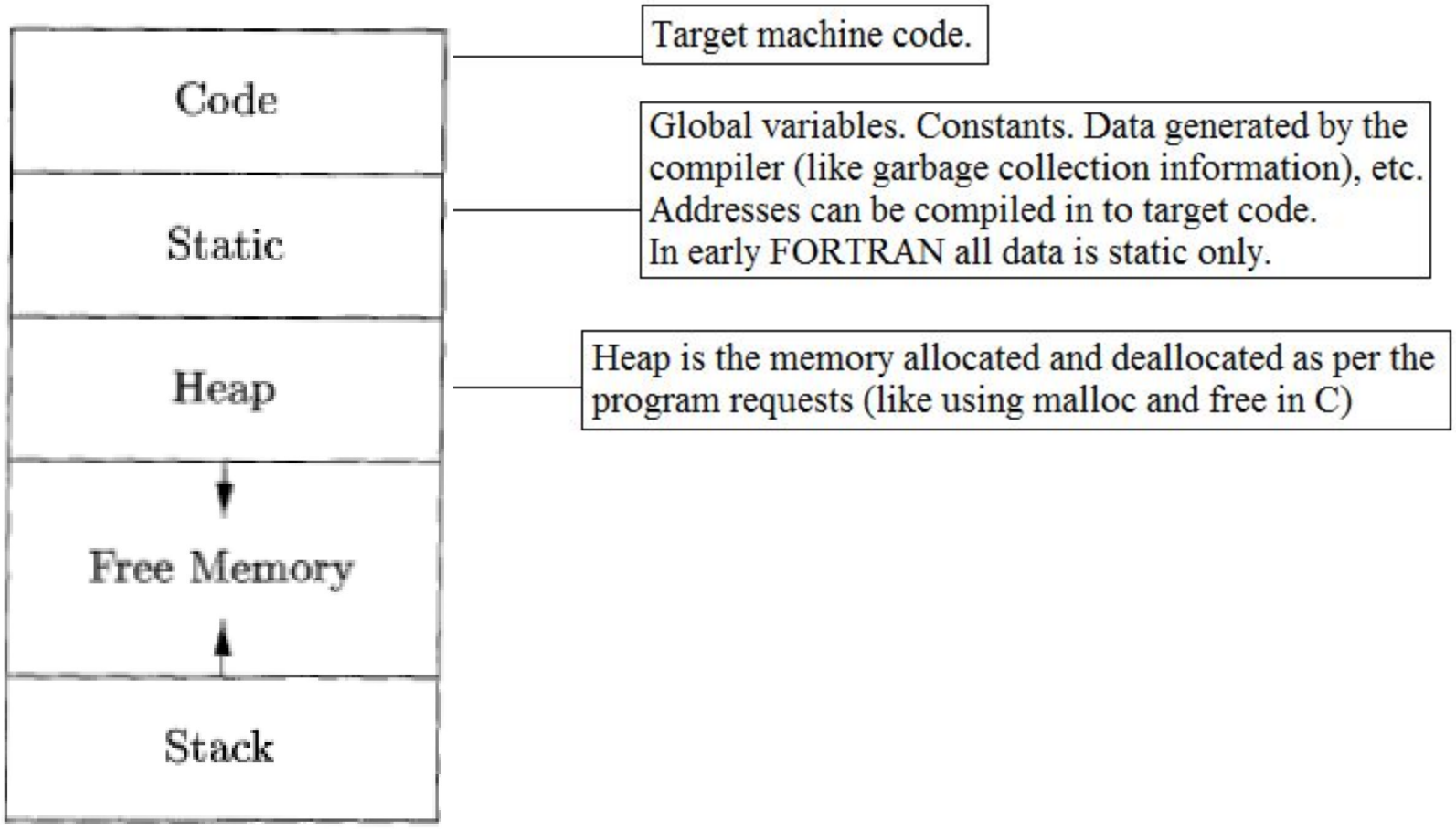
Storage Organization

The memory organization of a program at run-time :



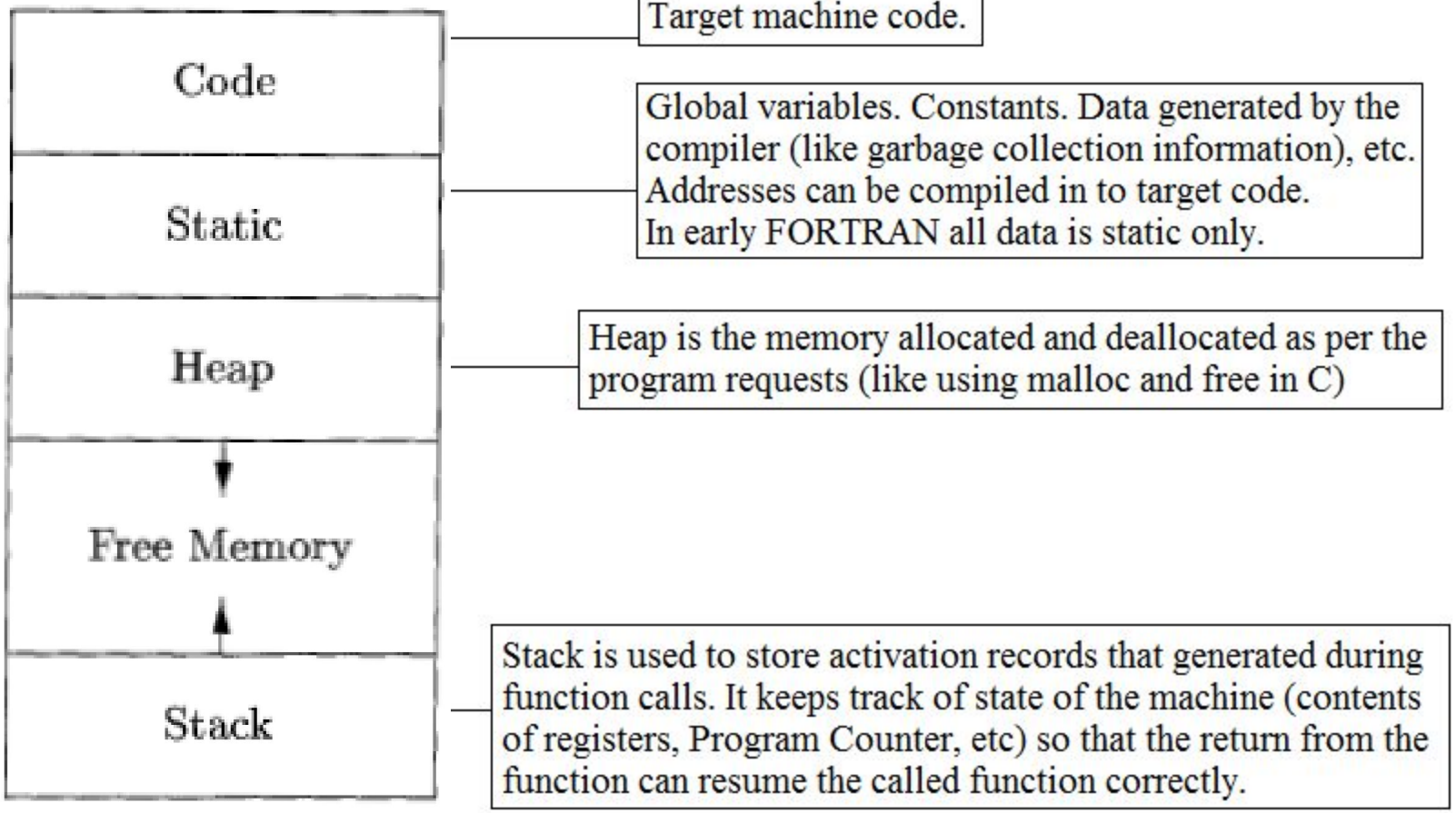
Storage Organization

The memory organization of a program at run-time :



Storage Organization

The memory organization of a program at run-time :



Garbage collection

- Heap memory will be wasted/becomes useless, if not properly released.

- Garbage collection automatically searches for these type of memory wastages and recovers them.
  - Modern languages, often, supports automatic garbage collection, despite it being a difficult task.

- Storage comes in blocks.
  - Block is a contiguous space, whose address is the address of the first byte.
- Many consecutive blocks might be used for some program components (like arrays).

Static vs. Dynamic Allocation

- **Static**: Compile time, **Dynamic**: Runtime allocation
- Many compilers use some combination of following
  - Stack storage: for local variables, parameters and so on
  - Heap storage: Data that may outlive the call to the procedure that created it
- Stack allocation is a valid allocation for procedures since procedure calls can be nested

- Code occupies the lowest portion
- Global variables are allocated in the static portion
- Remaining portion of the address space, stack and heap are allocated from the opposite ends to have maximum flexibility

Activation Record

Storage space needed for variables associated with each activation of a procedure – *activation record* or *frame*

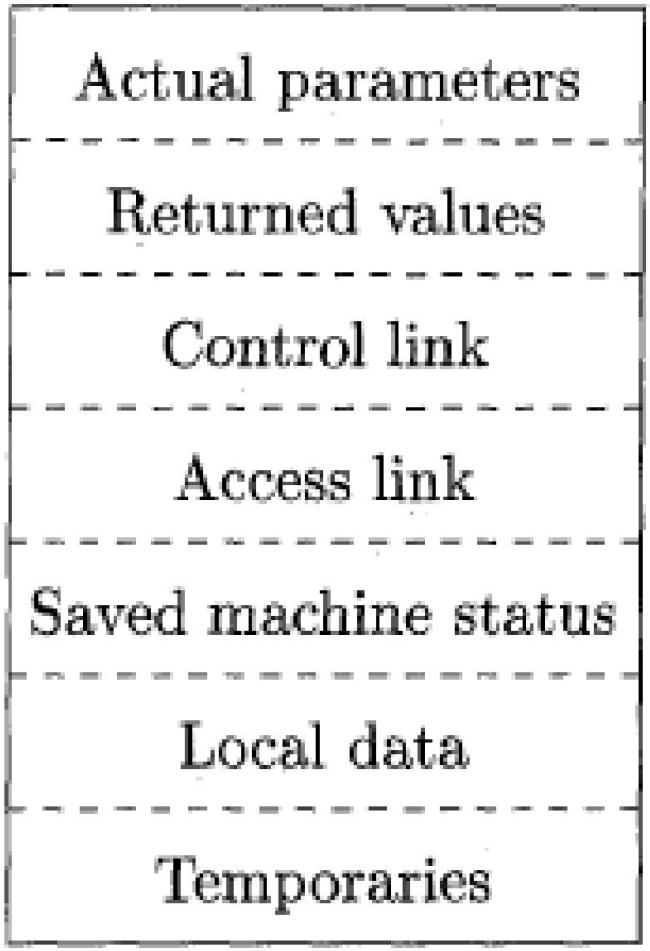
Typical activation record contains

Parameters passed to the procedure

- Bookkeeping information, including return values

- Space for local variables
- Space for compiler generated local variables to hold sub-expression values

A General Activation Record



1. Temporary values, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers.

2. Local data belonging to the procedure whose activation record this is.

3. A saved machine status, with information about the state of the machine just before the call to the procedure. This information typically includes the *return address* (value of the program counter, to which the called procedure must return) and the contents of registers that were used by the calling procedure and that must be restored when the return occurs.

4. An "access link" may be needed to locate data needed by the called procedure but found elsewhere, e.g., in another activation record.

5. A *control link*, pointing to the activation record of the caller.

6. Space for the return value of the called function, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.

7. The actual parameters used by the calling procedure. Commonly, these values are not placed in the activation record but rather in registers, when possible, for greater efficiency. However, we show a space for them to be completely general.

Location for Activation Record

- Depending upon language, activation record can be created in the static, stack or heap area
- Creation in Static Area:
  - Early languages, like FORTRAN
  - Address of all arguments, local variables etc. are preset at compile time itself

    To **pass** parameters, values are copied into these locations at the time of invoking the procedure and copied back on return
  - There can be a single activation of a procedure at a time
  - Recursive procedures cannot be implemented

Location for Activation Record

- Creation in Stack Area:
  - Used for languages like C, Pascal, Java etc.
  - As and when a procedure is invoked, corresponding activation record is pushed onto the stack
  - On return. entry is popped out
  - Works well if local variables are not needed beyond the procedure body

- For languages like LISP, in which a full function may be returned, activation record created in the heap

Processor Registers

- Also a part of the runtime environment
- Used to store temporaries, local variables, global variables and some special information
- Program counter points to the statement to be executed next
- Stack pointer points to the top of the stack

- Frame pointer points to the current activation record
- Argument pointer points to the area of the activation record reserved for arguments

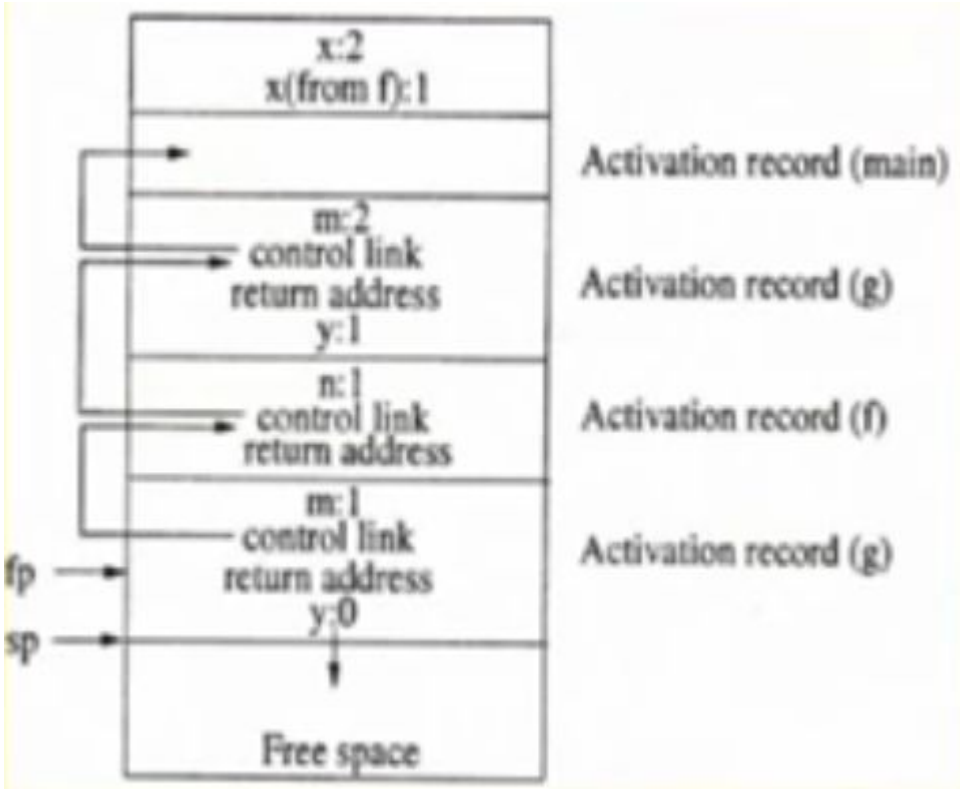
Environment Types

- Stack based environment without local procedures common for languages like C
- Stack based environment with local procedures – followed for block structured languages like Pascal

Environment without Local Procedures

- For languages where all procedures are global
- Stack based environment needs two things about activation records
  - Frame pointer: Pointer to the current activation record to allow access to local variables and parameters
  - Control link / Dynamic link: Kept in current activation record to record position of the immediately preceding activation record

```
int x = 2;
void f( int n ) {
    static int x = 1;
    g(n);
    x--;
}
void g( int m ) {
    int y = m – 1;
    if (y > 0) {
        f(y);
        x--;
    }
}
main() {
    g(x); return 0;
}
```



Activation Record Creation

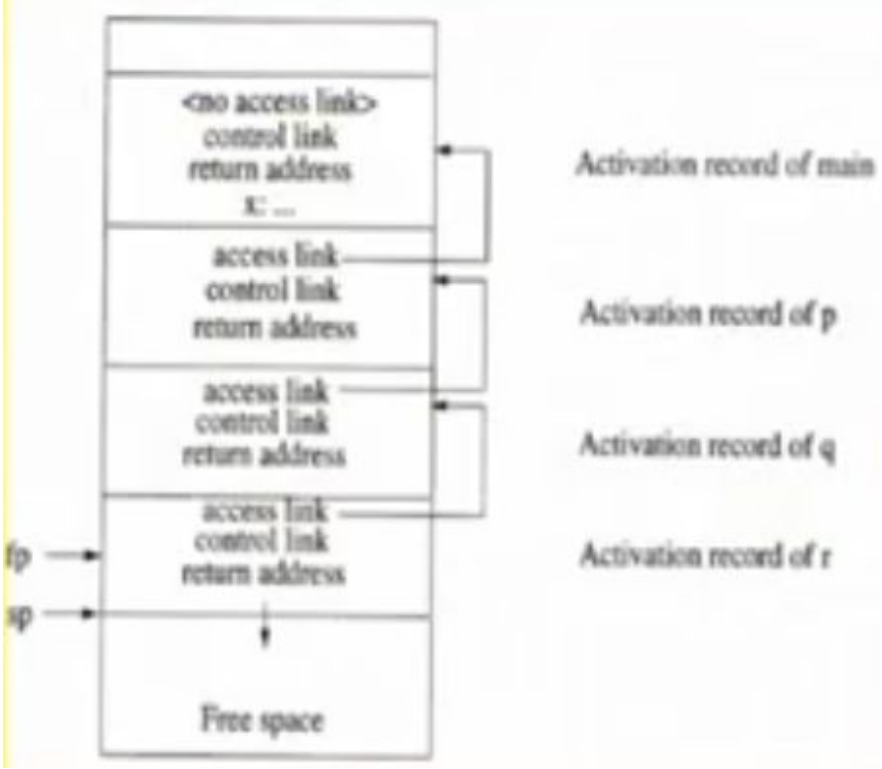
At a call	
Caller	**Callee**
1. Allocate basic frame 2. Store parameters 3. Store return address 4. Save caller-saved registers 5. Store self frame pointer 6. Set frame pointer for child 7. Jump to child	1. Save callee saved registers, state 2. Extend frame for locals 3. Initialize locals 4. Fall through to code
At a return	
Caller	**Callee**
1. Copy return value 2. Deallocate basic frame 3. Restore caller-saved registers	1. Store return value 2. Restore callee-saved registers, state 3. Unextend frame 4. Restore parent's frame pointer 5. Jump to return address

Environment with Local Procedures

For supporting local procedures, variables may have various scopes

- To determine the definition to be used for a reference to a variable, it is needed to access non-local, non-global variables
- These definitions are local to one of the procedures nesting the current one – need to look into the activation records of nesting procedures
- Solution is to keep extra bookkeeping information, called *access* ***link*** pointing to the activation record for the defining environment procedure

```
program chaining;
 procedure p;
 var x: integer;
 procedure q;
    procedure r
    begin
       x := 2;
       ...
       if ... Then p;
    end {of r}
 begin
    r;
 end {of q}
 begin
    q;
 end {of p}
 begin {of main}
    p;
 end.
```



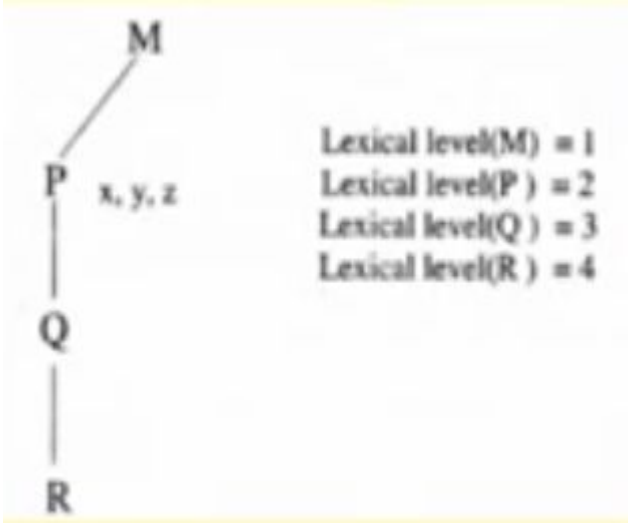
Current procedure r

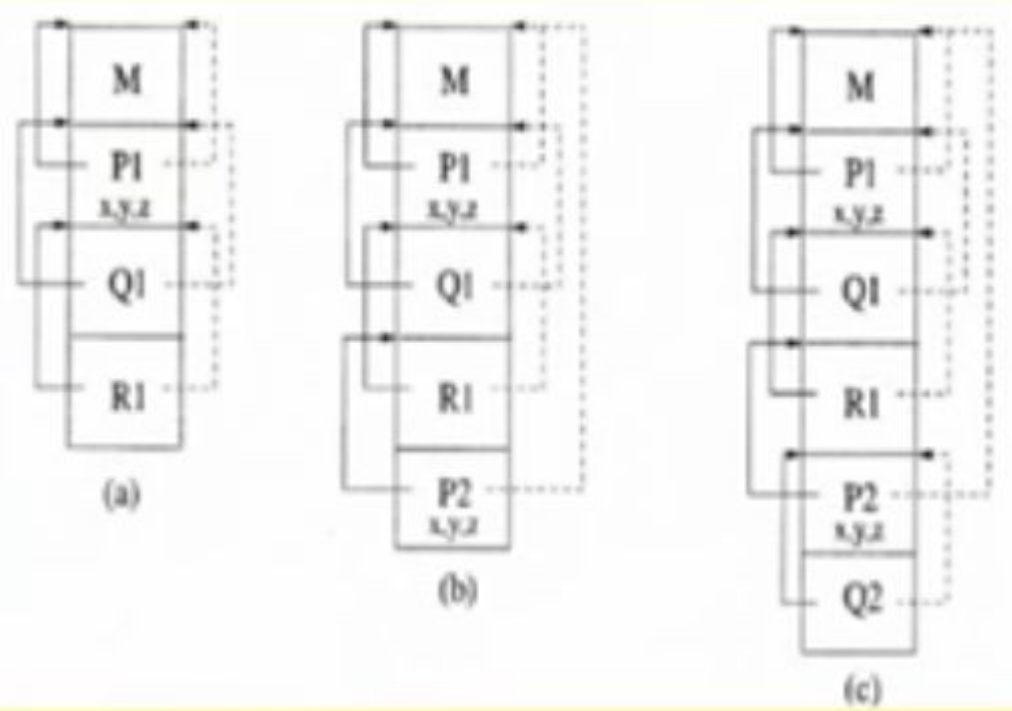
- To locate definition of x, it has to traverse through the activation records using access links
- When the required procedure containing definition of x is reached, it is accessed via offset from the corresponding frame pointer

Compiler's Responsibility

- Proper code to access the correct definitions:
  - Find difference *d* between the lexical nesting level of declaration of the name and the lexical nesting level of the procedure referring to it
  - Generate code for following *d* access links to reach the right activation record
  - Generate code to access the variable through offset mechanism

```
program M;
    procedure P;
        var x, y, z;
        procedure Q;
            procedure R;
            begin
                ... z = P; ...
            end R;
        begin
            ... y = R; ...
        end Q;
    begin
        ... x = Q; ...
    end P;
begin
    ... P; ...
end M;
```





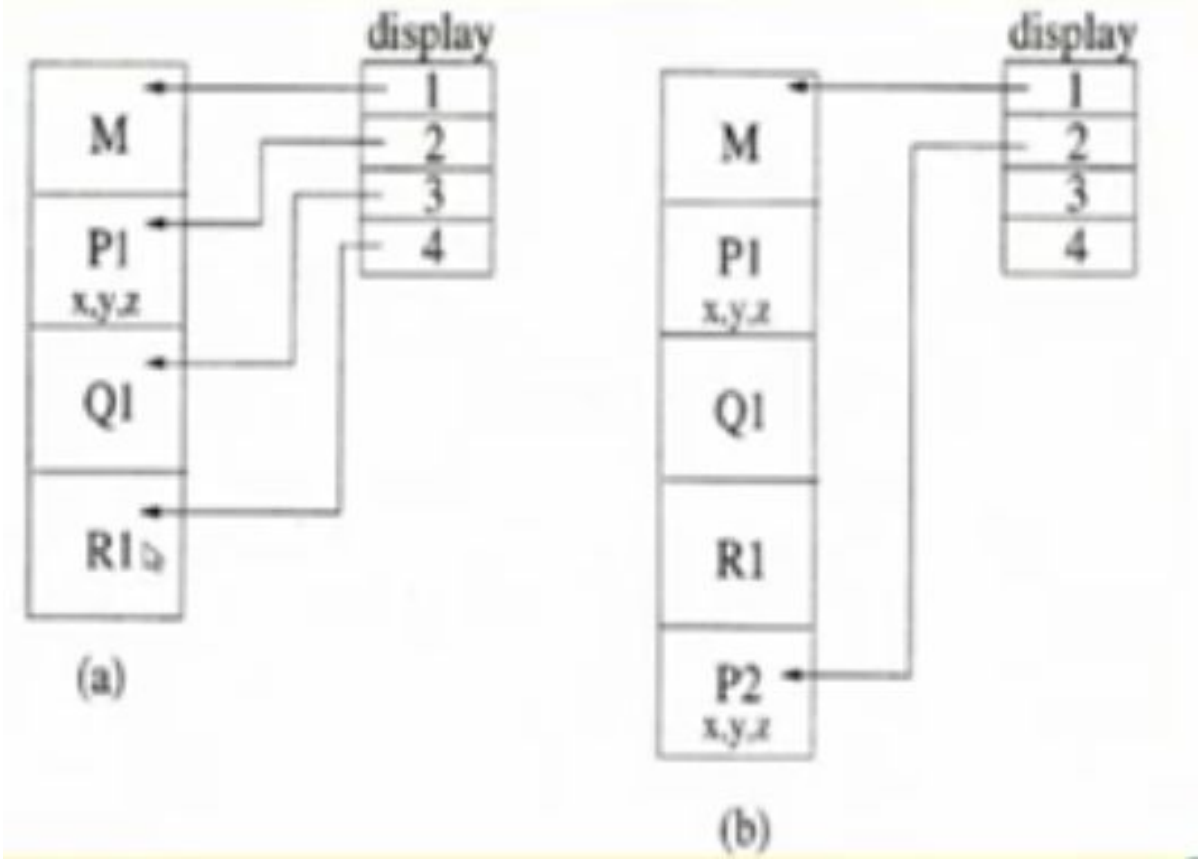
Display

- Difficulty in non-local definitions is to search by following access links
- Particularly for virtual paging environment, certain portion of the stack containing activation records may be swapped out,

  access may be very slow
- To access variables without search, *display* is used

Display

- When a procedure P at nesting depth i is called, following actions are taken:
  - Save value of d[i] in the activation record for P
  - Set d[i] to point to new activation record
- When a procedure P finishes:
  - d[i] is reset to the value stored in the activation record of P



- Maximum nesting depth 4, so 4 entries in the display
- In Fig (a), M has called P, P has called Q and Q has in turn called R

Compiler knows that x is in procedure P at lexical level 2

- Code is generated to access second entry of the display to reach the activation record of P directly
- Same is in Fig (b)

Materials:

NPTEL Compiler Course