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 {through system calls}.
- Main concerns of this lecture are
  - Procedure calls {how recursion is supported?!}
  - Memory (storage) management

7.1 Storage Organization

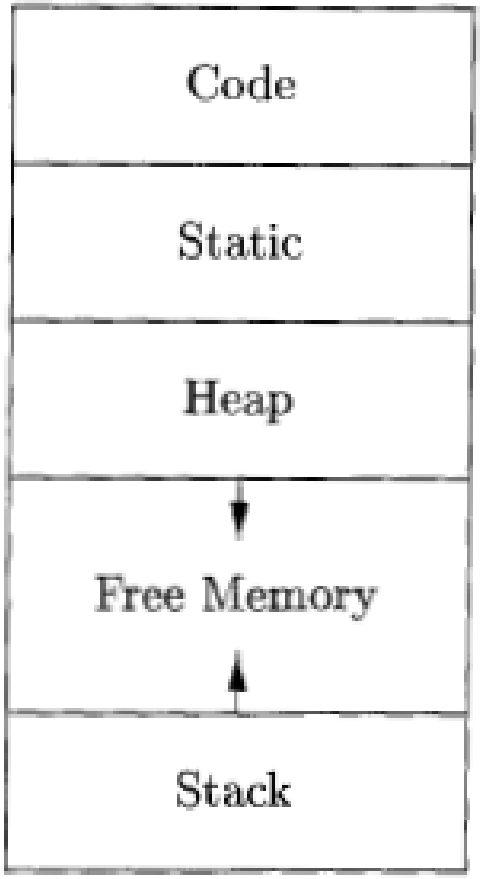
- Target program runs in its own logical address space.
  - Each program value has a location.
- The management and organization of this logical address space is shared between the compiler, the OS, and the target machine.
- How the logical address space is physically realized is done by the OS.
  -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

Code and Data Area in Memory

- Most programming languages distinguish between code and data
- Code consists of only machine instructions and normally does not have embedded data
  - Code area normally does not grow or shrink in size as execution proceeds
    - Unless code is loaded dynamically or code is produced dynamically
      - As in Java – dynamic loading of classes or producing classes and instantiating them dynamically through reflection
  - Memory area can be allocated to code statically
    - We will not consider Java further in this lecture
- Data area of a program may grow or shrink in size during execution

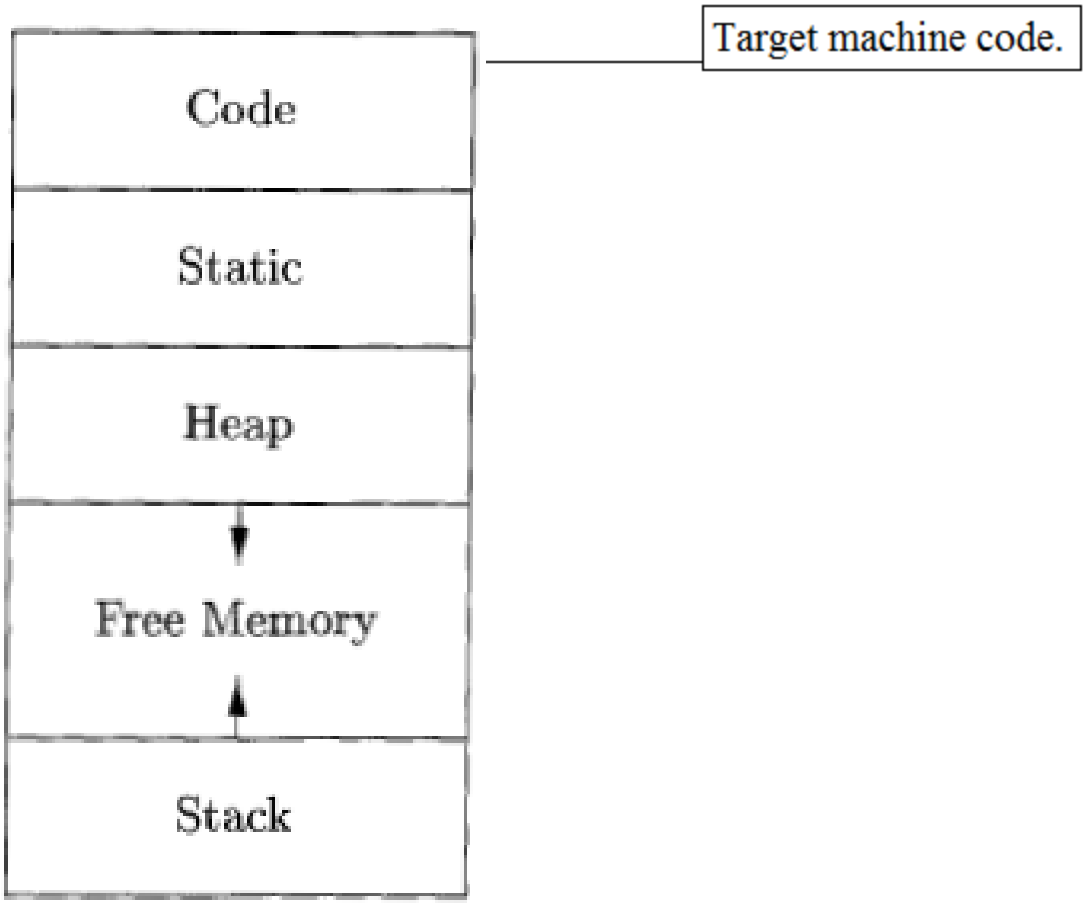
Storage Organization

The memory organization of a program at run-time :



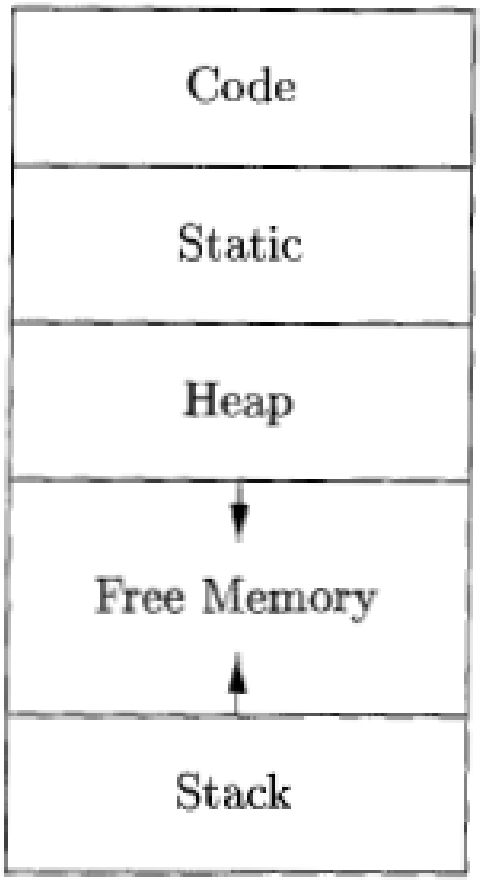
Storage Organization

The memory organization of a program at run-time :



Storage Organization

The memory organization of a program at run-time :

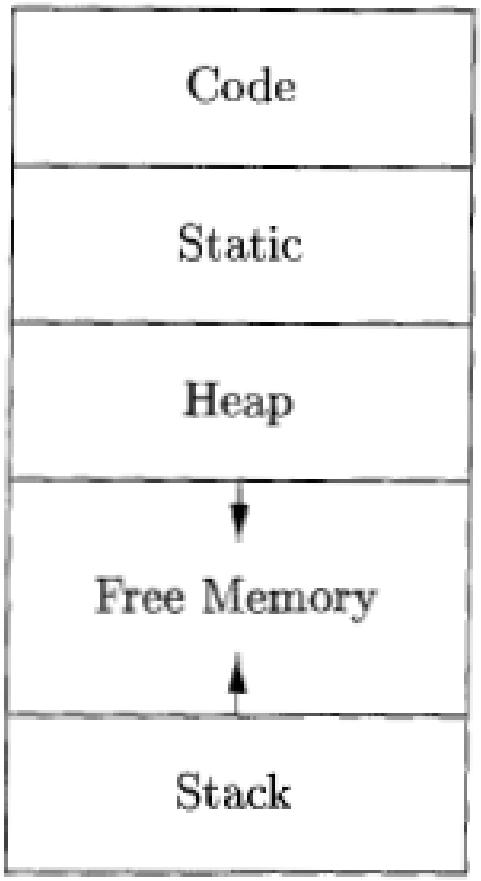


Target machine code.

Global variables. Constants. Data generated by the compiler (like garbage collection information), etc.
Addresses can be compiled in to target code.
In early FORTRAN all data is static only.

Storage Organization

The memory organization of a program at run-time :



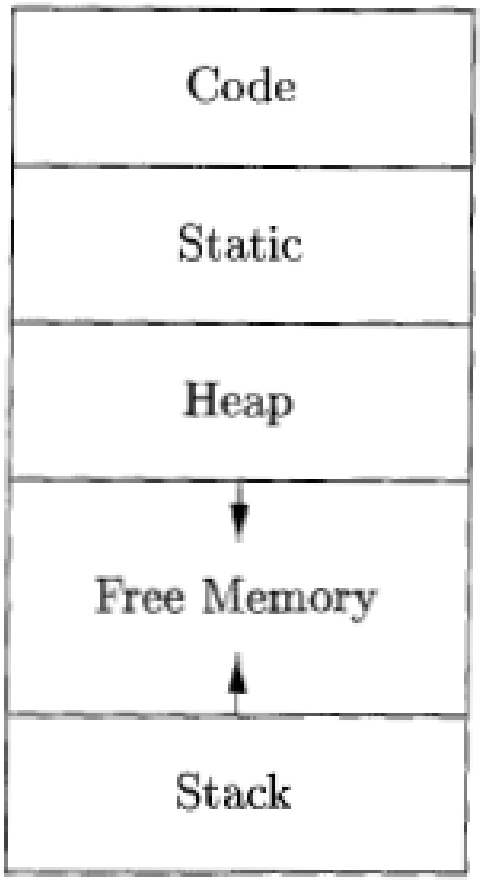
Target machine code.

Global variables. Constants. Data generated by the compiler (like garbage collection information), etc.
Addresses can be compiled in to target code.
In early FORTRAN all data is static only.

Heap is the memory allocated and deallocated as per the program requests (like using malloc and free in C)

Storage Organization

The memory organization of a program at run-time :



Target machine code.

Global variables. Constants. Data generated by the compiler (like garbage collection information), etc. Addresses can be compiled in to target code. In early FORTRAN all data is static only.

Heap is the memory allocated and deallocated as per the program requests (like using malloc and free in C)

Stack is used to store activation records that generated during function calls. It keeps track of state of the machine (contents of registers, Program Counter, etc) so that the return from the function can resume the called function correctly.

Garbage collection

- Heap memory will be wasted/becomes useless, if not properly released.
- Programmer can often forget to release the dynamic memory.
- 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).
  - Often wastage (in the last block, a portion) is there. {Need to do more work if we do not want to waste.}

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

7.2 Stack Allocation of Space

- Languages that use functions, procedures, etc must use a part of its run-time memory as stack memory.
  - Activation records – local variables.
- A function or procedure can be compiled on its own.
  - Uses relative addresses for local variables.
  - These are mapped to absolute addresses at run-time. This is possible because of stack memory.

7.2.1 Activation Trees

Stack allocation would not be feasible if procedure calls, or *activations* of procedures, did not nest in time. The following example illustrates nesting of procedure calls.

7.2.1 Activation Trees

Stack allocation would not be feasible if procedure calls, or *activations* of procedures, did not nest in time. The following example illustrates nesting of procedure calls.

Example 7.1 : Figure 7.2 contains a sketch of a program that reads nine integers into an array a and sorts them using the recursive quicksort algorithm.

Sketch of a quicksort program

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Figure 7.2: Sketch of a quicksort program

At run-time

- partition(1,9) returned 4
- So, a[1] through a[3] is in one block, a[5] to a[9] in the second block.

At run-time

- partition(1,9) returned 4
- So, a[1] through a[3] is in one block, a[5] to a[9] in the second block.

In this example, as is true in general, procedure activations are nested in time. If an activation of procedure p calls procedure q, then that activation of q must end before the activation of p can end. There are three common cases:

[1] Recall we use "procedure" as a generic term for function, procedure, method, or subroutine.

1. The activation of q terminates normally. Then in essentially any language, control resumes just after the point of p at which the call to q was made.

1. The activation of q terminates normally. Then in essentially any language, control resumes just after the point of p at which the call to q was made.

2. The activation of q, or some procedure q called, either directly or indirectly, aborts; i.e., it becomes impossible for execution to continue. In that case, p ends simultaneously with q.

1. The activation of q terminates normally. Then in essentially any language, control resumes just after the point of p at which the call to q was made.

2. The activation of q, or some procedure q called, either directly or indirectly, aborts; i.e., it becomes impossible for execution to continue. In that case, p ends simultaneously with q.

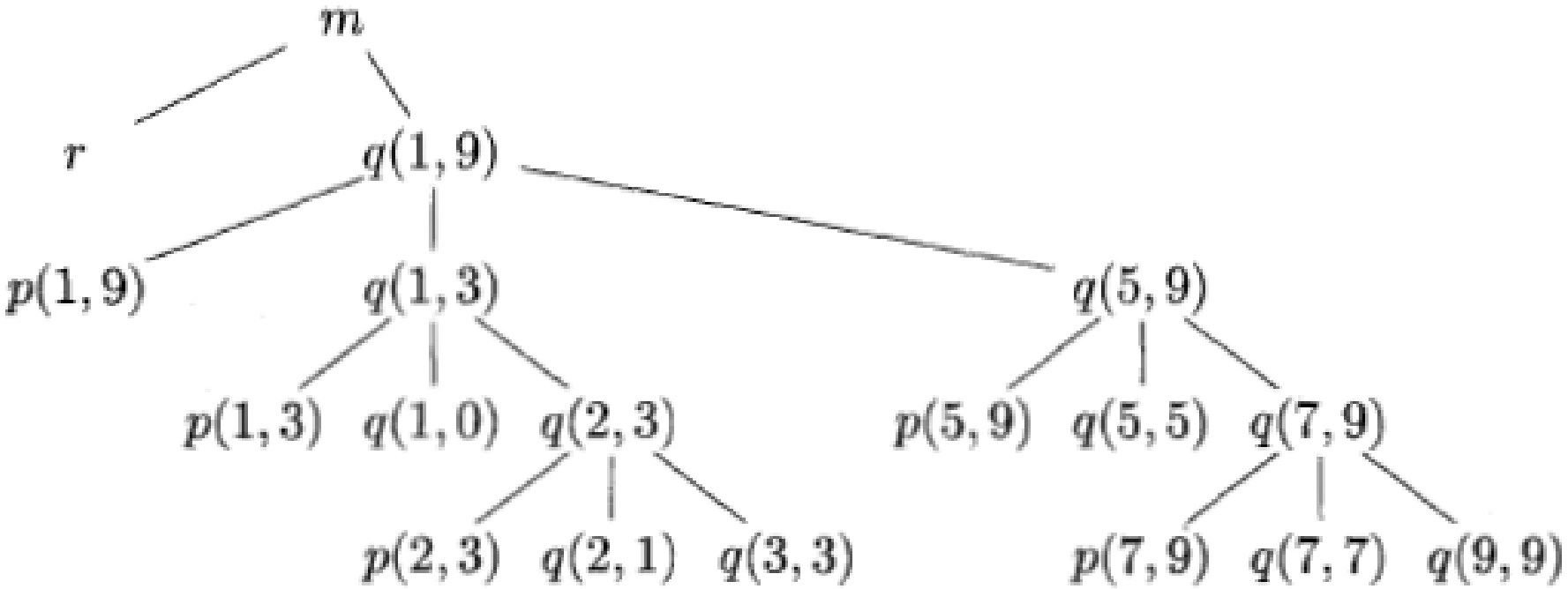
3. The activation of q terminates because of an exception that q cannot handle. Procedure p may handle the exception, in which case the activation of q has terminated while the activation of p continues, although not necessarily from the point at which the call to q was made. If p cannot handle the exception, then this activation of p terminates at the same time as the activation of q, and presumably the exception will be handled by some other open activation of a procedure.

- We can see activations of procedures as a hierarchy (tree) called the **activation tree**
- Each node corresponds to one activation,
  - The root is the action of the “main”
  - At a node for activation of *p*, the children are activations for procedure calls called by *p*.

Activation for Quicksort

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

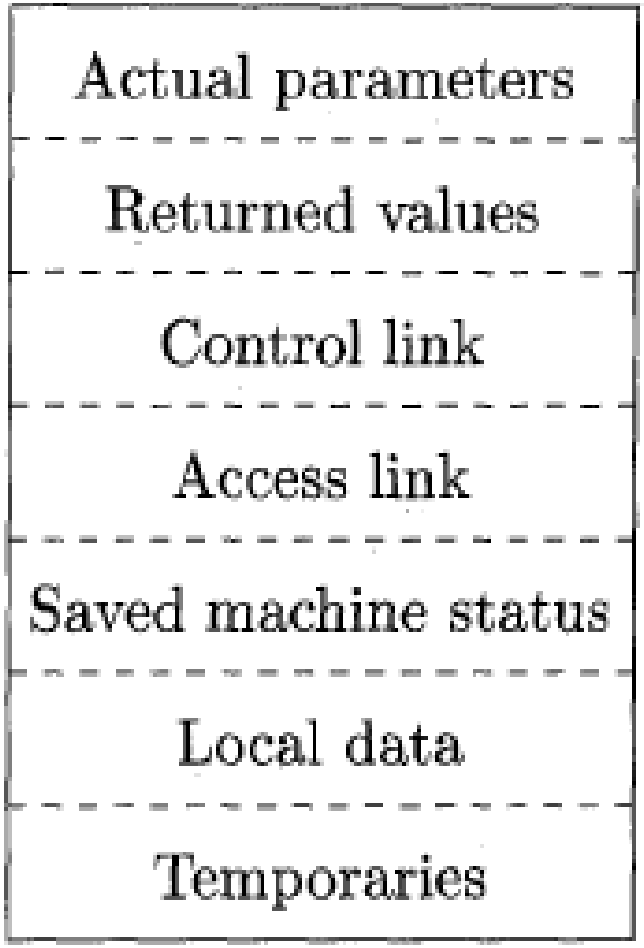
Activation tree representing calls during an execution of quicksort



Activation records …

- Procedure calls and returns are usually managed by a run-time stack called the control stack.
- Each live activation has an activation record (sometimes called a frame)
- The root of activation tree is at the bottom of the stack
- The current execution path specifies the content of the stack with the last activation has record in the top of the stack.

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.

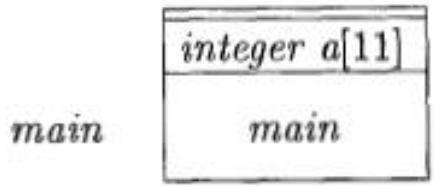
4. An "access link" may be needed to locate data needed by the called procedure but found elsewhere, e.g., in another activation record. Access links are discussed in Section 7.3.5.

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

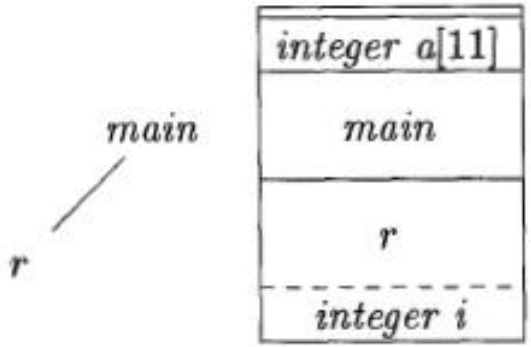
6. Space for the return value of the called function, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.

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.

Quicksort example …

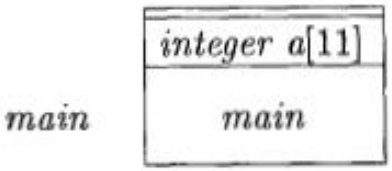


(a) Frame for *main*

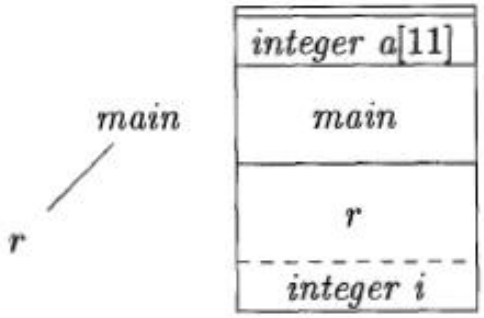


(b) r is activated

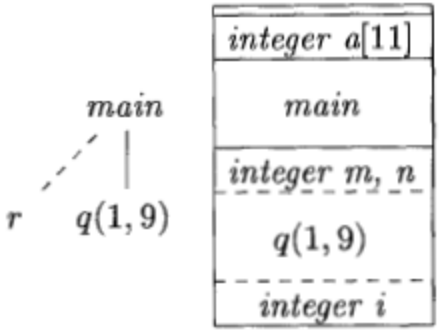
Quicksort example …



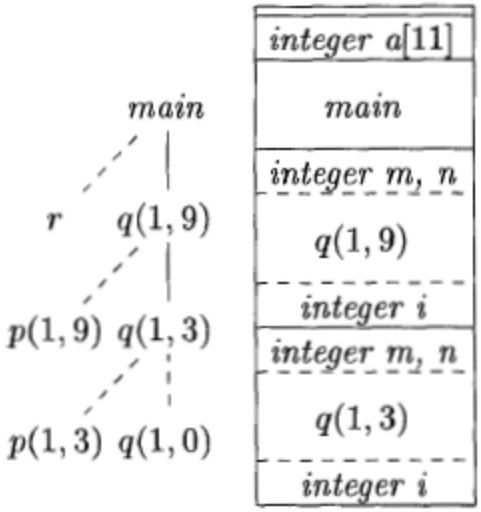
(a) Frame for *main*



(b) r is activated



(c) r has been popped and $q(1,9)$ pushed



(d) Control returns to $q(1,3)$

Exercises

- Considering the quicksort

Exercise 7.2.1 : Suppose that the program of Fig. 7.2 uses a *partition* function that always picks $a[m]$ as the separator v. Where m is the first element of the list.

Draw the activation tree when the numbers $9, 8, 7, 6, 5, 4, 3, 2, 1$ are sorted.

Exercise 7.2.3: In Fig. 7.9 is C code to compute Fibonacci numbers recursively. Suppose that the activation record for f includes the following elements in order: (return value, argument n, local s, local t); there will normally be other elements in the activation record as well. The questions below assume that the initial call is $f(5)$.

a) Show the complete activation tree.

b) What does the stack and its activation records look like the first time $f(1)$ is about to return?

! c) What does the stack and its activation records look like the fifth time $f(1)$ is about to return?

```
int f(int n) {
    int t, s;
    if (n < 2) return 1;
    s = f(n-1);
    t = f(n-2);
    return s+t;
}
```

Figure 7.9: Fibonacci program for Exercise 7.2.3

7.2.3 Calling Sequences

- “Calling sequence” consists of code that allocates an activation record on the stack and enters information into its fields.
- A “return sequence” is similar code that does the necessary restoration of the state of the machine so that the **caller** can continue (after the return of the called (**callee**)).

- The code in a calling sequence is divided between the caller and callee. {this division is influenced by OS, target m/c, source lang., etc}
- For a procedure, the portion of the callee depends only on the callee, hence is generated once only.
- Whereas, for caller, for the same procedure, its portion can differ based on where the call occurs. So as many as number of times the procedure is called that many times it needs to be generated.

An example division

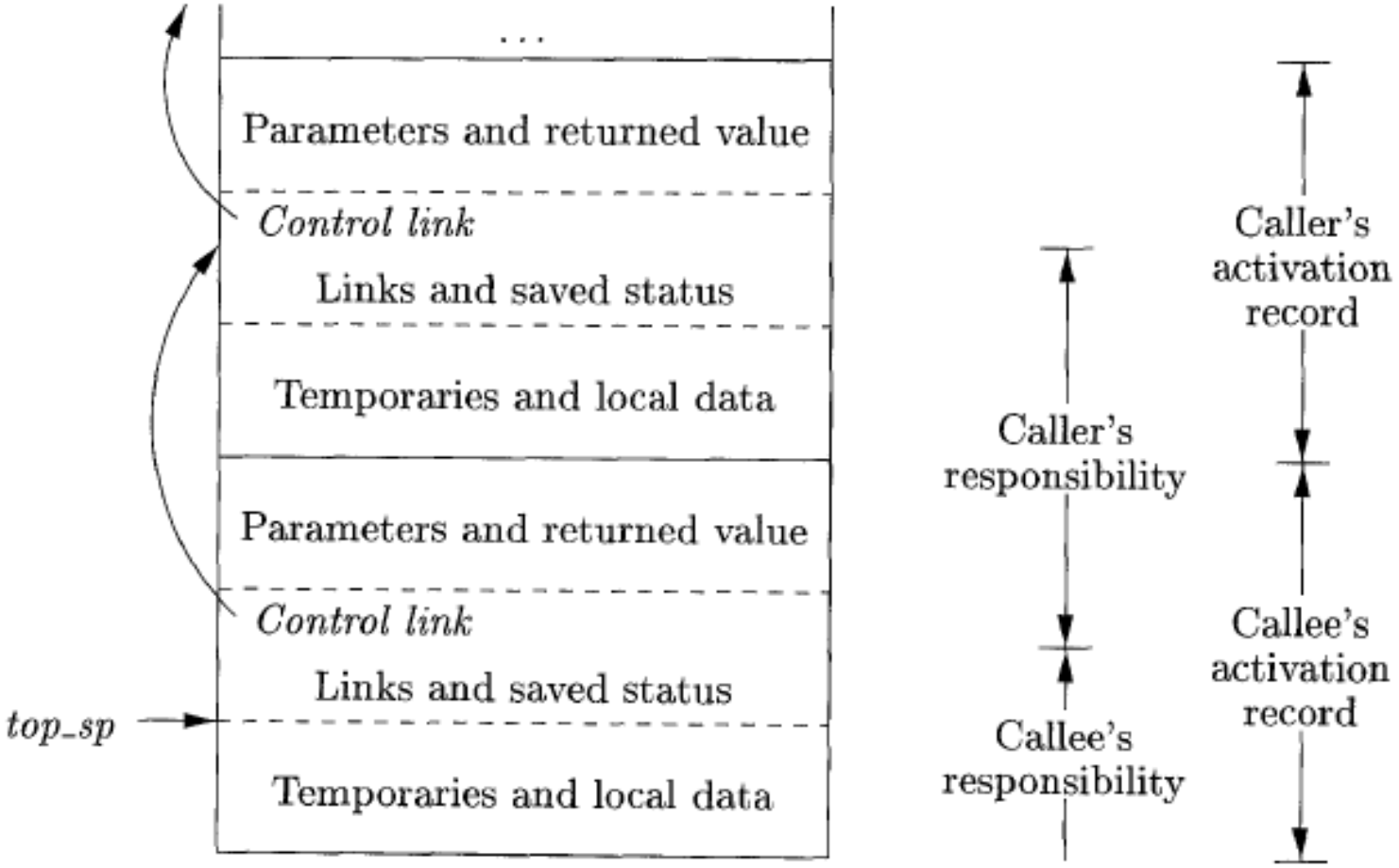


Figure 7.7: Division of tasks between caller and callee

- Actual implementation of calling sequence and return sequence has many details.
- One can read Dragon book 7.2, 7.3
-