Run-Time Environments
Module 4

Prepared by

Jesna J S

Assistant Professor, CSE

Topics Covered

- Source Language issues
- Storage organization
- Storage-allocation strategies.

Run-Time Environments

- To generate code, we need to connect source code (written program) with the actions that happen when the program runs.
- Example: The same variable name in source code may refer to different data objects when the program runs.
- **Allocation and deallocation of data objects are handled by the run-time support package.**
- Run-time support package: **A set of routines added to the program to help manage data objects while running.**
- This package depends on the programming language used (e.g., Fortran, Pascal, Lisp).

- Activation is simply a record or instance of a function/procedure running at a particular time.
- **Every time a function is called, a new activation is created to store and manage**:
  - ✓Local variables
  - ✓Function parameters
  - ✓Return address (where to go back after function finishes)
  - ✓Temporary data needed during function execution
- **Type** of data decides how it's represented in memory:
  - ✓**Primitive types** (characters, integers, reals) are stored directly.
  - ✓**Complex types** (arrays, strings, structures) are stored as **collections of primitive objects**.

Source Language Issues

- Procedure definition is a declaration that, in its simplest form, associates an identifier with a statement. The identifier is the procedure name, and the statement is the procedure body. A complete program will also be treated as a procedure.

a. Activation trees

- Each **execution of a procedure body is referred to as an activation of the procedure**.
- Each execution of a procedure starts at the beginning of the procedure body and eventually returns control to the point immediately following the place where the procedure was called.
- The lifetime of an activation of a procedure p is *the **sequence of steps between the first and last steps in the execution of the procedure body, including time spent executing procedures called by p***, the procedure called by them, and so on.
- If a and b are procedure activations, then their lifetimes are either non-overlapping or are nested. That is, if b is entered before a is left, then control must leave b before it leaves a.

- **A procedure is recursive if a new activation can begin before an earlier activation of the same procedure has ended.**
- A recursive procedure p need not call directly, p may call another procedure q, which may then call p through some sequence of procedure calls.
- **Activation tree is used to depict the way control enters and leaves activation.**

Activation tree represents::

1. ***Each node represents an activation of a procedure***
2. ***The root represents the activation of the main program.***
3. ***The node for a is the parent of the node for b if and only if control flows from activation a to b and,***
4. ***The node for a is to the left of the node for b if and only if the lifetime of a occurs before the lifetime of b.***

Procedure Quicksort()

Activation Tree

```
Execution begins...
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)
Execution terminated.
```

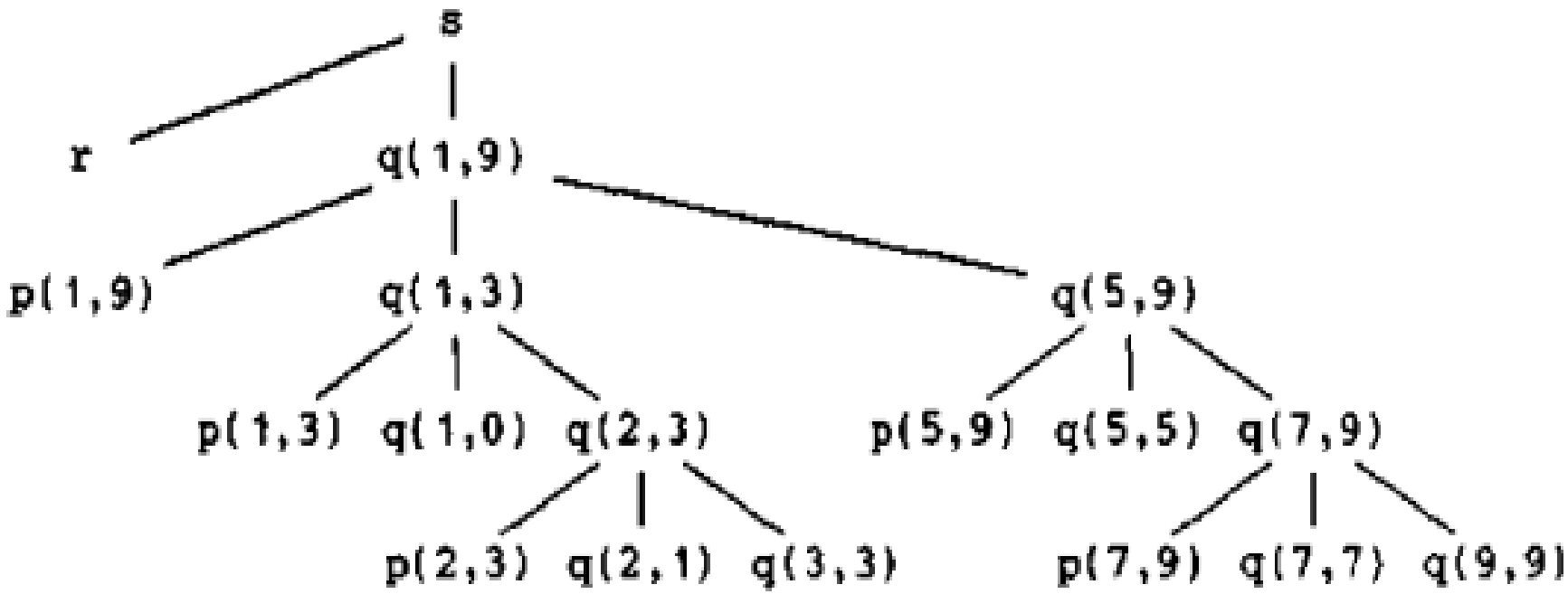
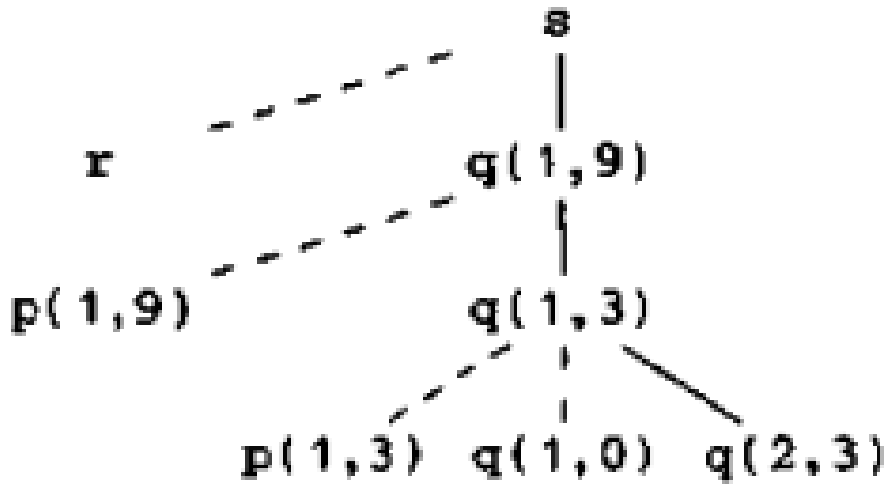


Fig. 7.2. Output suggesting activations of procedures

b. Control Stack

- It is used to keep of live procedure activations.
- **The node is pushed on to the stack when an activation begins and pop it when activation ends.**
- The contents of the control stack are related to **paths to the root of the activation tree**.
- When node n is at the top of the control stack, the stack contains the nodes along the path from n to the root.



The control stack contains nodes along a path to the root.

At this point the control stack contains the following nodes along this path to the root (the top of the stack is to the right)

s, q(1,9), q(1,3), q(2,3)

c. Scope of a declaration

- **A declaration in a programming language connects a name (identifier) to information (like type).**
- Declarations can be **explicit** (written clearly) or **implicit** (assumed by language rules).
- **Scope rules determine which declaration applies to a name in the program text**.
- Scope of a declaration: the part of the program where that declaration is valid.
- A name used inside a procedure is local if it refers to a declaration inside that procedure.
- If it refers to a declaration outside the procedure, it is called nonlocal.

- At compile time, the symbol table can be used to find the declarations that applies to an occurrence of a name.
- **When a declaration is seen, a symbol table entry is created for it.**
- As long as we are in the scope of the declaration, its entry is returned when the name in its looked up.

d. Binding of names

- The environment refers to a **function that maps a name to a storage location**, and the term state refers to a **function that maps a storage location to the value** held there.
- Environments and states are different, an assignment changes the state, but not the environment.

Fig. 7.5. Two-stage mapping from names to values.

- **When an environment associates storage location s with a name x, we say that x is bound to s; the association itself is referred to as a binding of x**.
- A binding is the dynamic counterpart of a declaration.

STATIC NOTION	DYNAMIC COUNTERPART
definition of a procedure	activations of the procedure
declaration of a name	bindings of the name
scope of a declaration	lifetime of a binding

Storage Organization

Subdivision of Run-Time memory

- The compiler obtain a block of storage from the OS for the compiled program to run it.
- This run-time storage might be subdivided to hold:
    1. **The generated target code**
    2. **Data objects**
    3. **A counterpart of the control stack to keep track of procedure activations.**
- The size of the generated target code is fixed at compile time, so the compiler can place it in a static it in a statically determined area, perhaps in the low end of memory.
- Also the size of some data objects may also be known at compile time, and placed in a statically determined area.

- A separate area of run-time memory, called a heap, holds all other information.
- Implementations of languages in which the lifetimes of activations cannot be represented by an activation tree might use the **heap to keep information about activations**.
- The size of stack and the heap can change as the program executes, so these are placed at opposite ends of memory where they can grow toward each other as needed.

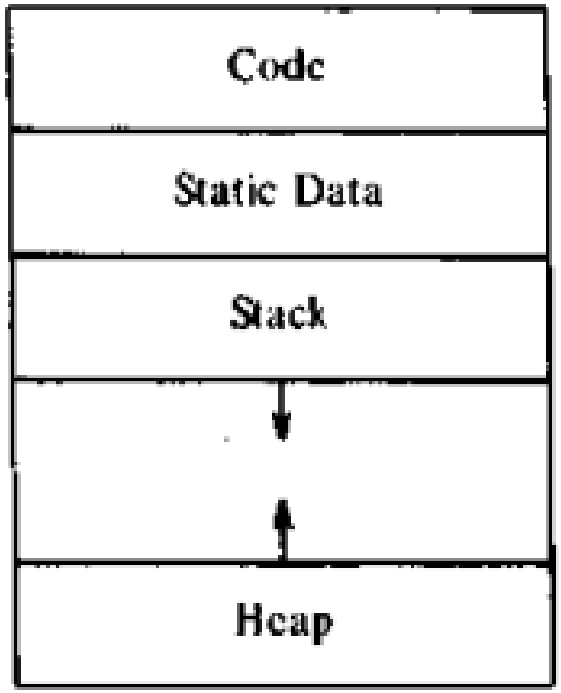


Fig: Typical subdivision of run-time memory into code and data areas

Activation Records

- **Information needed by a single execution of a procedure is managed using a contiguous block of storage called an activation record** or frame, consisting of the collection of fields shown in the figure below

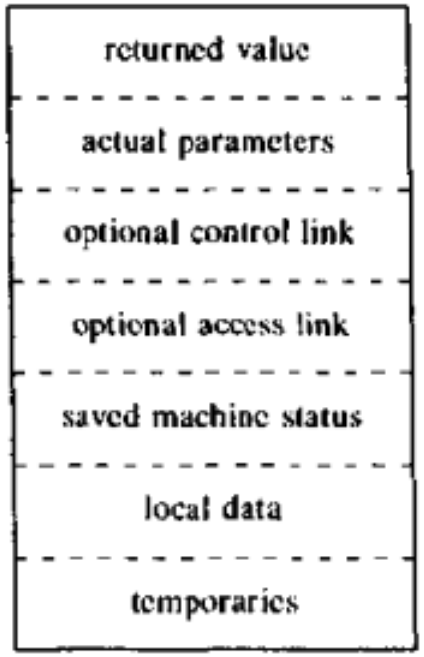


Fig. 7.8. A general activation record.

The purpose of the fields of an activation record is as follows, starting from the field for temporaries.

1. Temporary values, such as those arising in the evaluation of expressions, are stored in the field for temporaries.
2. The field for local data holds data that is local to an execution of a procedure. The layout of this field is discussed below.
3. The field for saved machine status holds information about the state of the machine just before the procedure is called. This information includes the values of the program counter and machine registers that have to be restored when control returns from the procedure.

4. The optional *access link* is used in Section 7.4 to refer to nonlocal data held in other activation records. For a language like Fortran access links are not needed because nonlocal data is kept in a fixed place. Access links, or the related "display" mechanism, are needed for Pascal.

5. The optional *control link* points to the activation record of the caller.

6. The field for actual parameters is used by the calling procedure to supply parameters to the called procedure. We show space for parameters in the activation record, but in practice parameters are often passed in machine registers for greater efficiency.

7. The field for the returned value is used by the called procedure to return a value to the calling procedure. Again, in practice this value is often returned in a register for greater efficiency.

Compile time layout of local data

- **Run-time data** is stored in **contiguous blocks** of memory.
- **Amount of storage** depends on **data type**:
  - ✓**Simple types** (e.g., character, integer, real): stored in natural size.
  - ✓**Aggregates** (e.g., arrays, records/structures): stored as a whole in a **single block of memory**.
- **Fixed-length data**: Size known at compile time; allocated directly.
- **Variable-length data**: Only a pointer/reference is stored; actual size known at run time.
- **Relative address (or offset) is computed for each variable relative to the beginning of its storage block (like activation record).**
- Offset = Distance from the start of the block to the variable's position.

- Some systems require data alignment (e.g., integers aligned to addresses divisible by 4).
  - Padding: Extra space added to ensure alignment.
  - Example: If a 4-byte integer is required, but previous data ends at an odd byte, 3 padding bytes may be added.
  - Aligning data makes memory access faster and efficient.
  - Non-aligned access may cause errors or performance issues.
  - Compiler decides the position and padding during compile time.
  - If space is tight, packing is done to minimize unused space, but may need extra work at runtime to align data when used.

Storage allocation strategies

- A different storage-allocation strategy is used,
    1. **Static allocation lays out storage for all data objects t compile time.**
    2. **Stack allocation manages the run-time storage as a stack**
    3. **Heap allocation allocates and deallocates storage as needed at run time from a data area known as a heap**

1. Static allocation

In static allocation

- Names (variables) are given fixed storage locations at compile time.
- No need for run-time support for storage management.
- Each time a procedure runs, the same storage locations are used for its variables.

- Local variables retain their values between activations (repeated calls) of a procedure.
- If a procedure is called again, the local variables have the same values as before, unless changed.

At compile time, the compiler:

- Calculates how much **space** to allocate for each variable.
- Decides the **position** (offset) of each variable within an **activation record**.

- Once decided, this layout is **fixed** for the program.
- The **target code** knows the addresses of data to work on **at compile time**.

Limitations of Static Allocation:

1. **Fixed size**: Size and position of data must be **known at compile time**.
2. **No recursive procedures**: Cannot handle recursive calls because **each activation** would need **separate copies** of local variables, but static allocation reuses the same.
3. **No dynamic data structures**: Cannot create **dynamic structures** (like linked lists) as **storage cannot be assigned at runtime**.

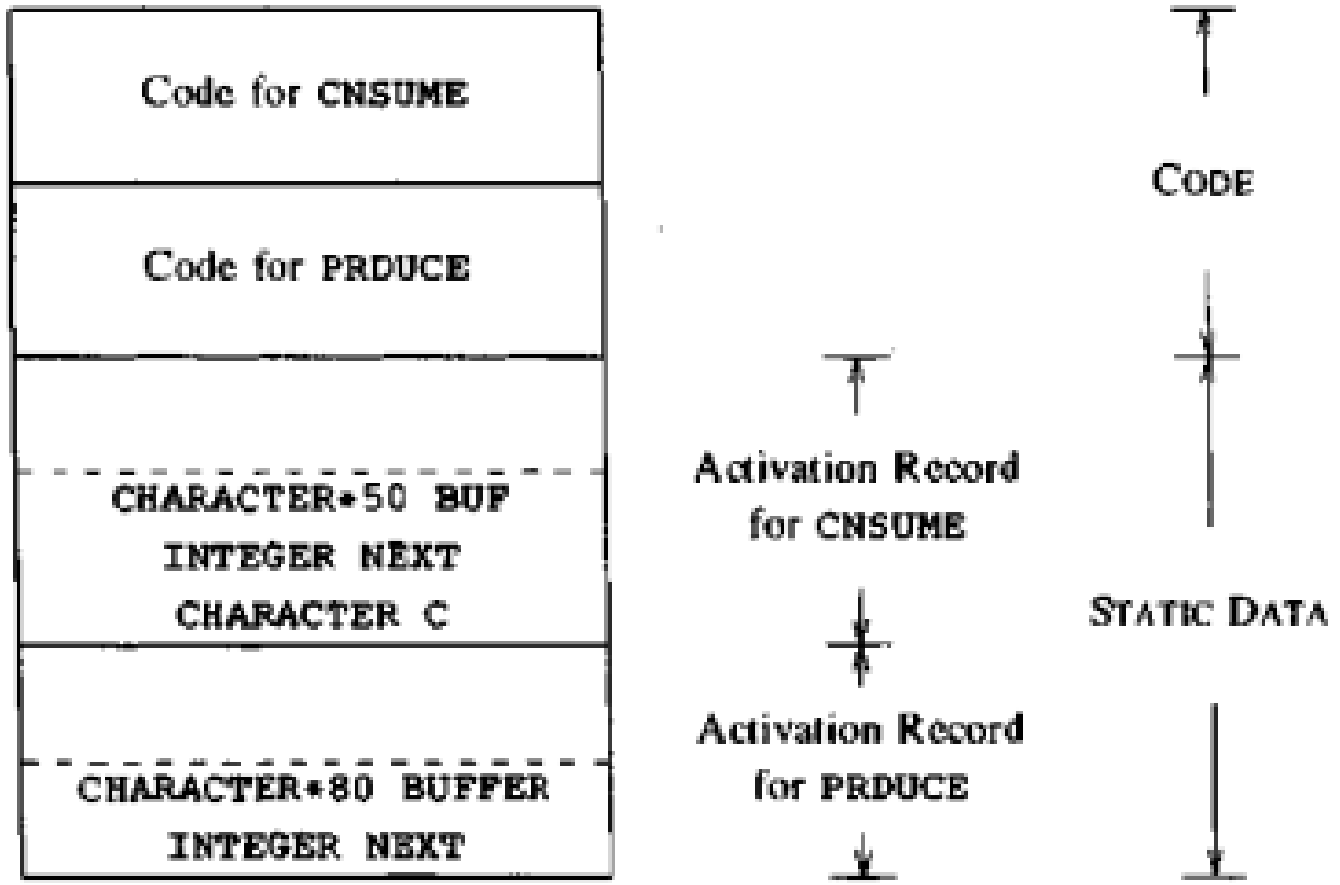


Fig. 7.11. Static storage for local identifiers of a Fortran 77 program.

2. Stack allocation

- All compilers for languages that use procedures, functions or methods as units of user functions define actions manage at least part of their runtime memory as a stack run- time stack.
- Each time a procedure called, space for its local variables is pushed onto a stack, and when the procedure terminates, space popped off from the stack.

Calling Sequence

- ✓A calling sequence is a set of instructions generated to handle procedure (function) calls in a program.
- ✓It allocates an activation record (stack frame) and fills it with necessary information.
- ✓A return sequence is used to restore machine state so that the caller can continue execution after the callee (called function) finishes.
- ✓Calling sequence code may be shared between the caller and callee.
- ✓No fixed rule on how to divide tasks between caller and callee — it depends on compiler design, language, machine, and OS.

- Fields in activation records that are fixed-size and known early are placed in the middle of the record.
- These include:
  - ✓Control link (points to caller’s activation record)
  - ✓Access link (for accessing non-local data)
  - ✓Machine status (state info like registers)
  - ✓Decision to include these links is made at compiler construction time.

✓Space for temporary variables is allocated in activation records.

✓Though final size is known at compile time, front-end (parser) may not know it.

✓Optimizations may reduce the number of temporaries.

- Each call has its **own parameters** (arguments).
  - ✓Caller evaluates parameters and passes them to callee.
  - ✓Parameters and return value fields are placed near activation records for easy access.
  - ✓In run-time stack, callee's activation record is placed above caller’s record.
  - ✓Caller can use offsets to access these parameters without knowing full layout of callee’s record.

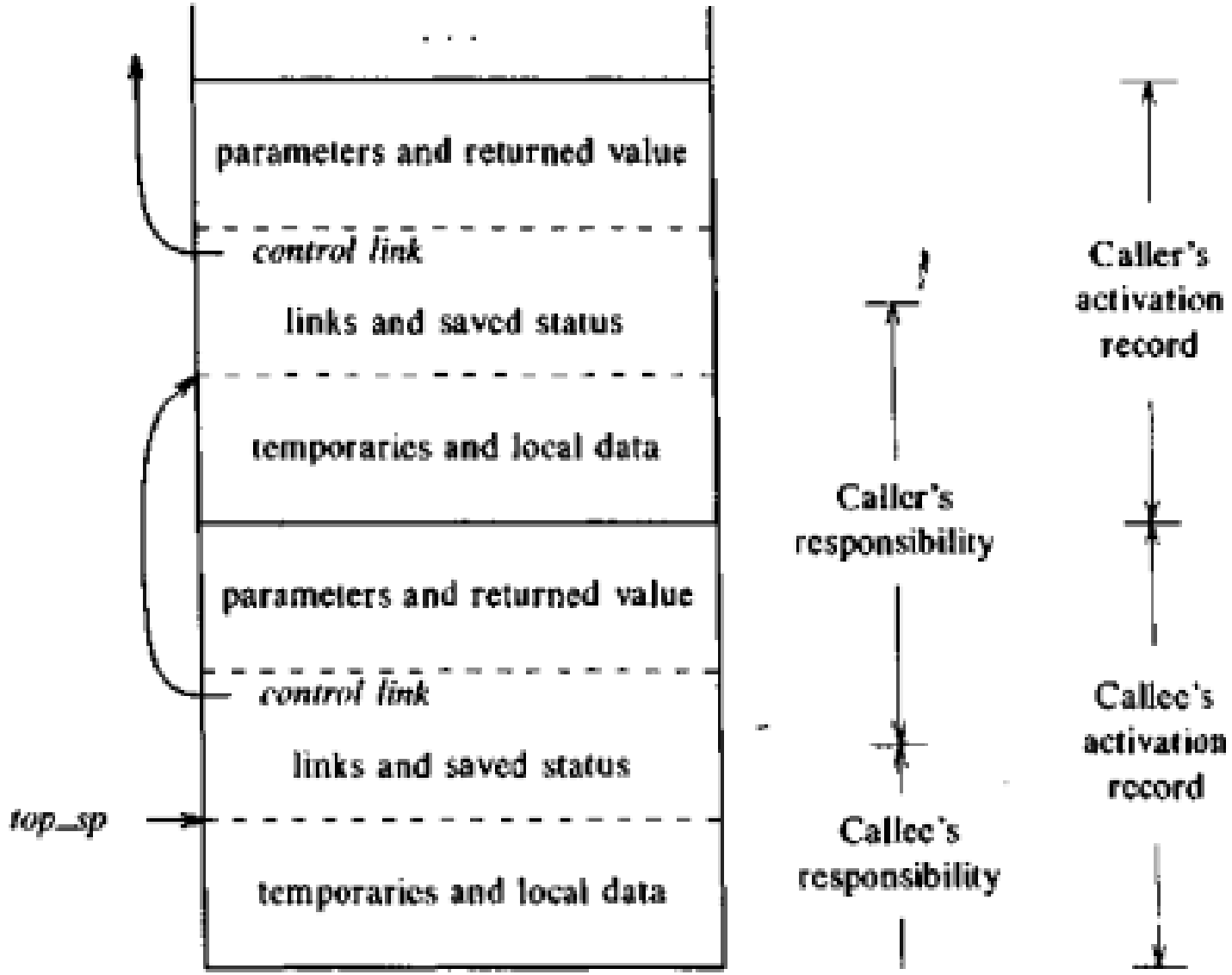


Fig. 7.14. Division of tasks between caller and callee.

The code for the callee can access its temporaries and local data using offsets from *top_sp*. The call sequence is:

1. The caller evaluates actuals.
2. The caller stores a return address and the old value of *top_sp* into the callee's activation record. The caller then increments *top_sp* to the position shown in Fig. 7.14. That is, *top_sp* is moved past the caller's local data and temporaries and the callee's parameter and status fields.
3. The callee saves register values and other status information.
4. The callee initializes its local data and begins execution.

A possible return sequence is:

1. The callee places a return value next to the activation record of the caller.
2. Using the information in the status field, the callee restores *top_sp* and other registers and branches to a return address in the caller's code.
3. Although *top_sp* has been decremented, the caller can copy the returned value into its own activation record and use it to evaluate an expression.

Heap allocation

- Heap allocation is used when **stack allocation** (LIFO) **cannot** be used due to:
  1. **Local variables' values** need to be kept **even after a function ends**.
  2. A **called function (callee)** may **outlive** the caller (e.g., if a function returns a reference to its local data).
- In these cases, activation records (function call data) can't be deallocated in order.
- Stack works on Last-In-First-Out (LIFO), but here random deallocation is needed, so heap is used.

- Heap provides contiguous memory blocks when needed for:
  - ✓Activation records
  - ✓Other objects
- Memory pieces can be freed in any order, creating mixed free and used spaces over time.
- In stack, activation records are strictly ordered, deallocated when the function ends.
- In heap, activation records can stay even after the function ends if needed.
- Example: If procedure r is called and retained, it won't be deallocated like stack.
- If another procedure q(1,9) is called, its record cannot be placed next to r automatically as in a stack--heap manager handles this placement.
- **The heap will manage free and used spaces.**
- **Efficiently allocate and deallocate memory blocks as needed**.

- Heap management is complex and costly in terms of time and space.
- Efficient heap management is important for performance, especially when using activation records or data that require dynamic allocation
- **To make heap management faster for small and predictable-size data**, we can use special handling:
  - ❑Linked List of Free Blocks:
    - ✓Maintain a linked list for each size of block (free blocks).
  - ❑Fulfilling Requests:
    - ✓When a block of size s is needed, try to give a block of size s' (smallest available size $\geq$ s).
    - ✓When block s' is freed, return it to the same linked list it was taken from.
  - ❑Large Blocks:
    - ✓Use the general heap manager for large storage requests.

- **Benefits of this Approach:**
  - ✓**Fast allocation and deallocation** of small-sized blocks.
  - ✓**Linked lists** allow quick take/return of memory blocks.
  - ✓For **large blocks**, although more time-consuming, the **computation time dominates**, so allocation time is **less critical**.