CODE OPTIMIZATION

- Program transformation technique to improve the intermediate code by consuming fewer resources.

Characteristics

- Correct-it must not change the meaning of the program.
- increase the speed and performance of the program.
- The compilation time must be reasonable.
- The optimization process should not delay the overall compiling process.

Types of Code Optimization

- **Machine Independent Optimization –**

✔ improves the **intermediate code** to get a better target code.

✔ The part of the intermediate code which is transformed does not involve any CPU registers or absolute memory locations.

- **Machine Dependent Optimization –**

✔ Done after the **target code** has been generated and when the code is transformed according to the target machine architecture.

✔ It involves CPU registers, may have absolute memory references .

PRINCIPLE SOURCES OF CODE OPTIMIZATIONS

- A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.
- Two types

1. Function preserving transformations
2. Loop optimizations

Function Preserving Transformations

- Common sub expression elimination
- Copy propagation,
- Dead-code elimination
- Constant folding

1. Common Sub expression Elimination

An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation.

2. Copy Propagation

- Assignments of the form f : = g called copy statements
- Copy propagation means use of one variable instead of another.

Example-

x=Pi;

A=x*r*r;

After optimization

A=Pi*r*r;

3. Dead Code Elimination

- A variable is live at a point in a program if its value can be used subsequently
- Dead Code are statements that compute values that never get used.

Example

```
i=0;
if(i=1)  //dead code
{
a=b+5;
}
```

4. Constant Folding

- Computing the value of an expression which is a constant and using the constant instead is known as constant folding.

Example

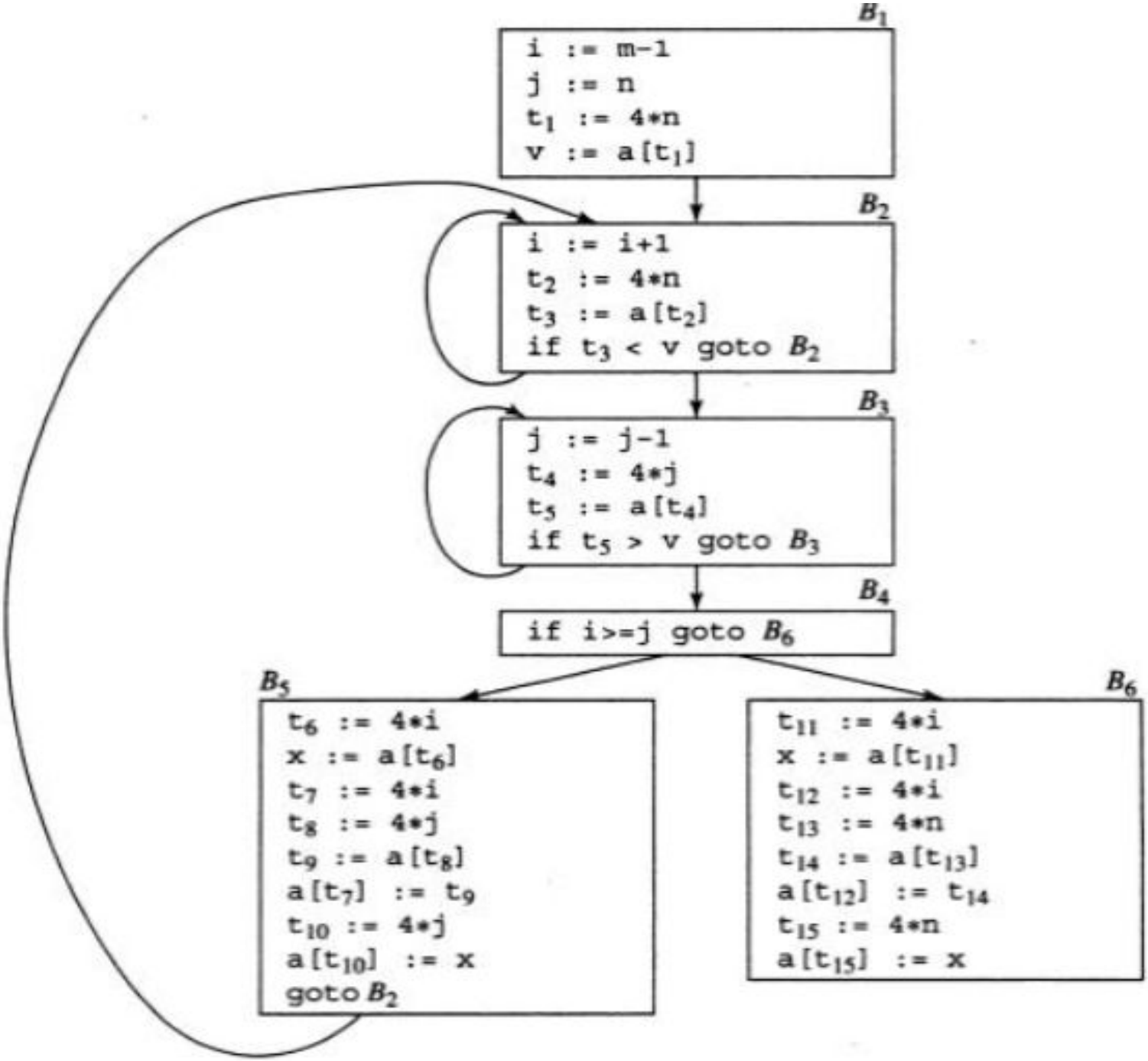
a=3.14157/2 can be replaced by

a=1.570

LOOP OPTIMIZATIONS

- The running time of a program may be improved if the number of instructions in an inner loop is decreased.
- **Types of loop optimizations**

1. Code motion-moves code outside a loop
2. Induction-variable elimination-replace variables from inner loop.
3. Reduction in strength- replaces expensive operation by a cheaper one.


B1
i := m-1
j := n
t1 := 4*n
v := a[t1]
B2
i := i+1
t2 := 4*n
t3 := a[t2]
if t3 < v goto B2
B3
j := j-1
t4 := 4*j
t5 := a[t4]
if t5 > v goto B3
B4
if i>=j goto B6
B5
t6 := 4*i
x := a[t6]
t7 := 4*i
t8 := 4*j
t9 := a[t8]
a[t7] := t9
t10 := 4*j
a[t10] := x
goto B2
B6
t11 := 4*i
x := a[t11]
t12 := 4*i
t13 := 4*n
t14 := a[t13]
a[t12] := t14
t15 := 4*n
a[t15] := x

1. Code Motion

- Reduces amount of code inside loop
- Takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop.

Example

```
while(i<u-9)
{ // code }
```

Can be replaced by

```
j=u-9
while(i<j)
{// }
```

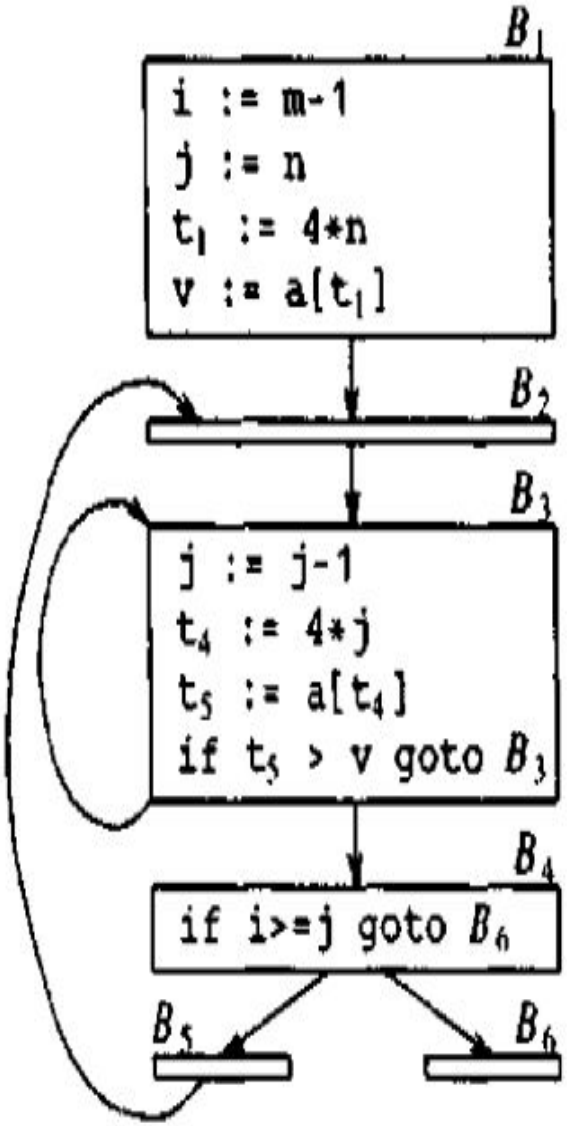
2. Induction Variables

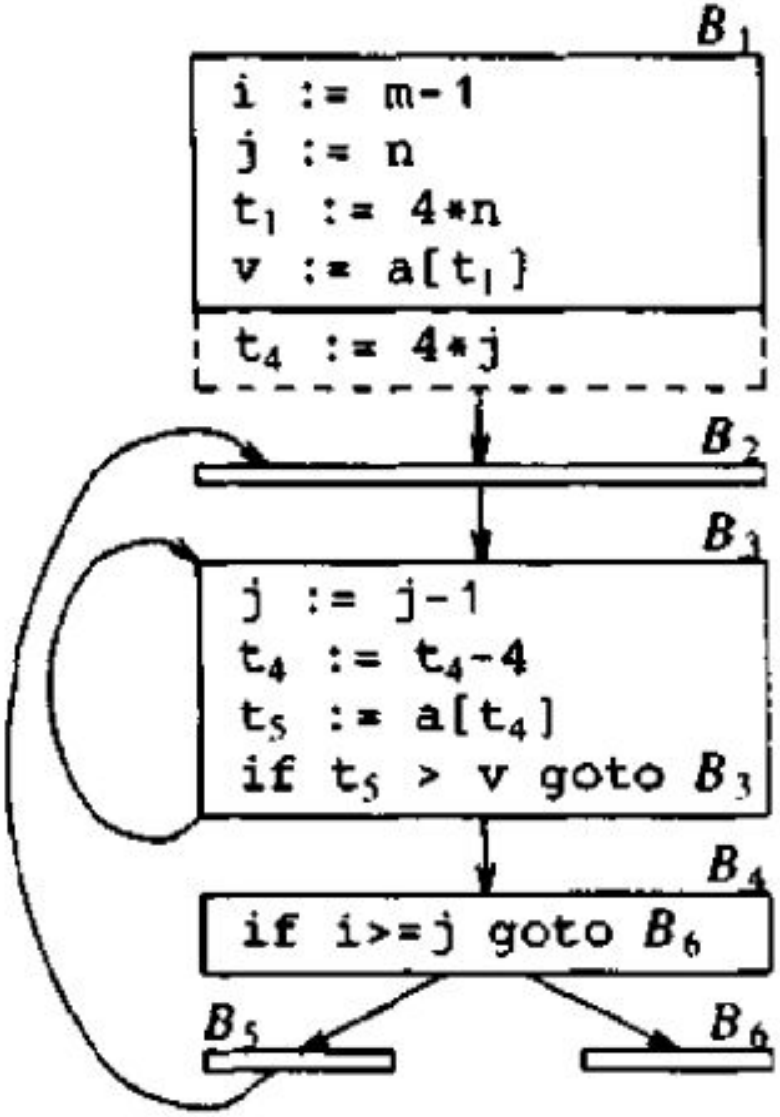
- It gets increased or decreased by a fixed amount on every iteration of a loop .
- Variables can get eliminated or strength reduction can be done.

3. Reduction in strength

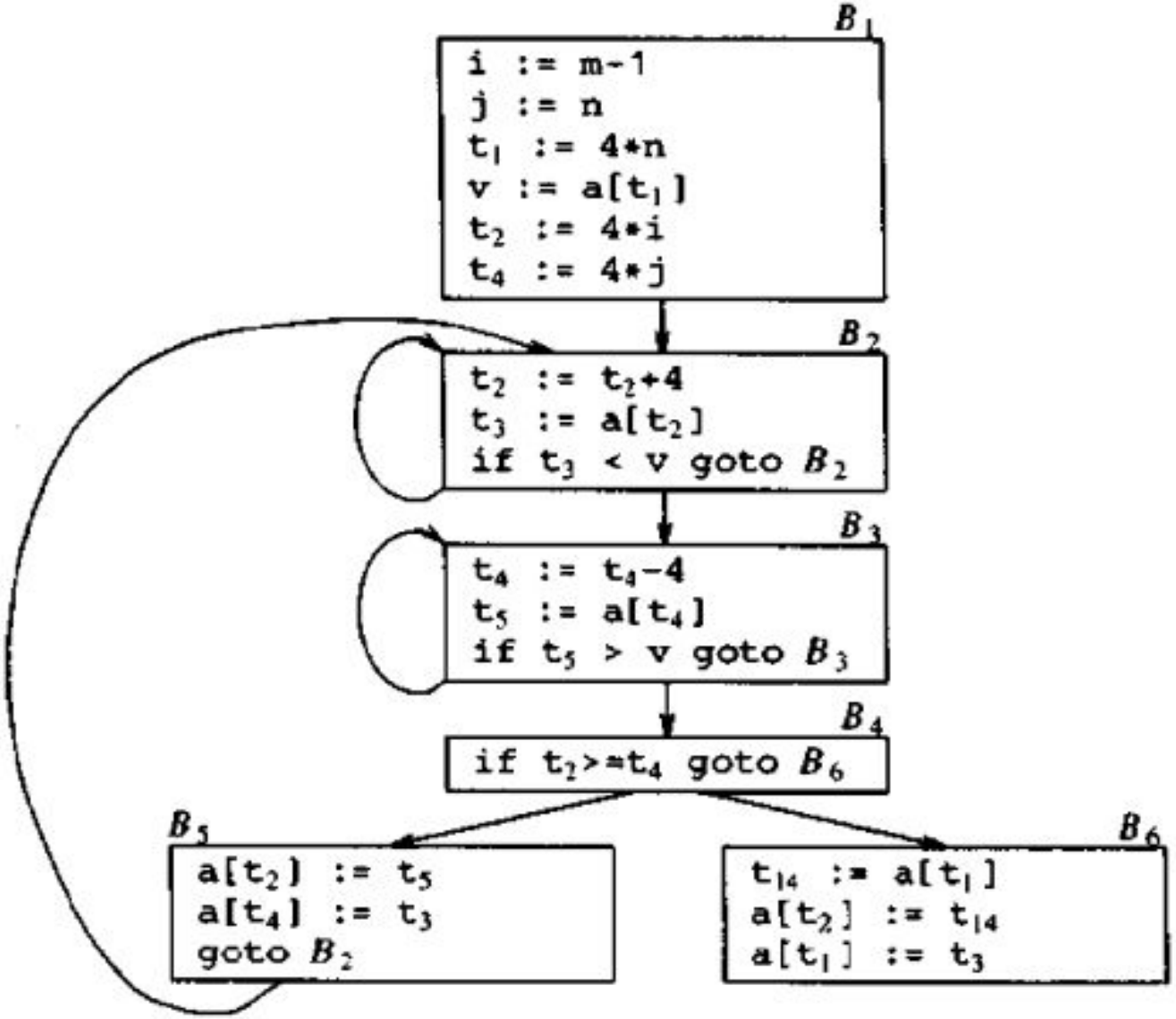
- Replace expensive operations by cheaper ones

Example


B1
i := m-1
j := n
t1 := 4*n
v := a[t1]
B2
B3
j := j-1
t4 := 4*j
t5 := a[t4]
if t5 > v goto B3
B4
if i>=j goto B6
B5
B6


B1
i := m-1
j := n
t1 := 4*n
v := a[t1]
t4 := 4*j
B2
B3
j := j-1
t4 := t4-4
t5 := a[t4]
if t5 > v goto B3
B4
if i>=j goto B6
B5
B6

After Induction variable elimination



INTRODUCTION TO GLOBAL DATA FLOW ANALYSIS

- Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program
- Based on optimization techniques used so far.
- Data flow equation

  out [S] = gen [S] U (in [S] - kill [S])
- **T**he information at the end of a statement is either generated within the statement , or enters at the beginning and is not killed as control flows through the statement.

Factors affecting data flow equations (out)

1.Values generated –gen

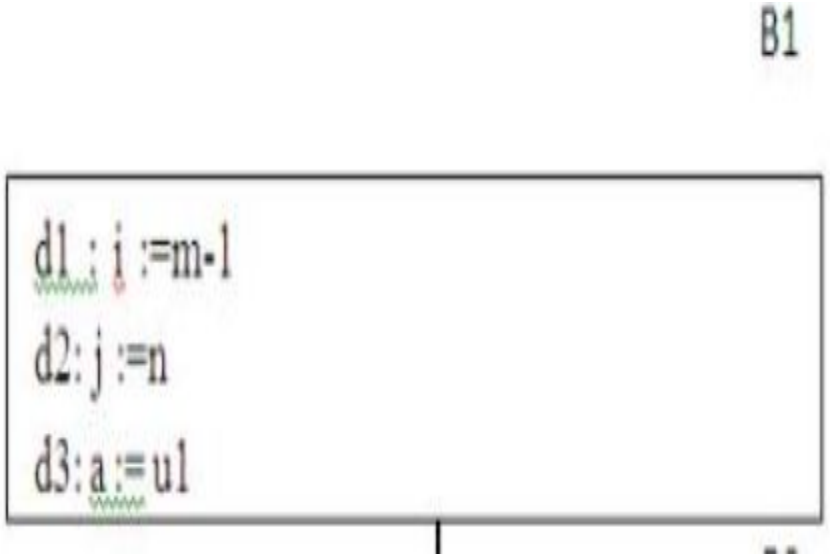
2.Values got as input-in

3. Values removed-kill

- Sometimes ‘in’ will also be computed.
- Equations are framed based on basic blocks where there is single and unique entry and exit point for the blocks.

Points and Paths:

- **Points** –entry and exit points
- B1 has four points: one before any of the assignments and one after each of the three assignments.



Paths

A path from p1 to pn is a sequence of points p1, p2,….,pn such that for each i between 1 and n-1

- Pi is the point immediately preceding a statement and pi+1 is the point immediately following that statement in the same block
- Pi is the end of some block and pi+1 is the beginning of a successor block.

Reaching definitions

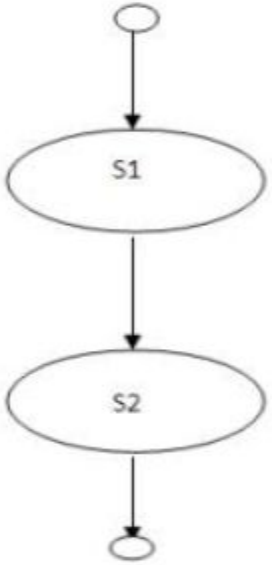
- Defining a variable means giving value to it.
- The values to a variable may come from input device or any other storage operation (**unambiguous definition**).

The most usual forms of **ambiguous definitions** of x are:

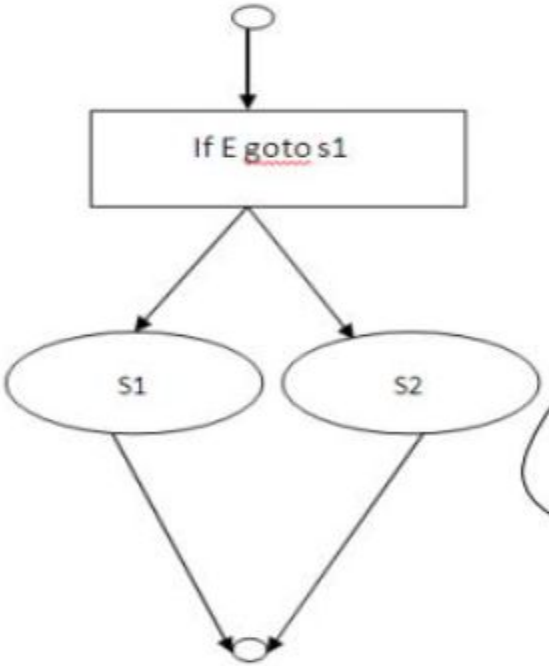
1. A call of a procedure with x as a parameter or a procedure that can access x.

2. An assignment through a pointer.

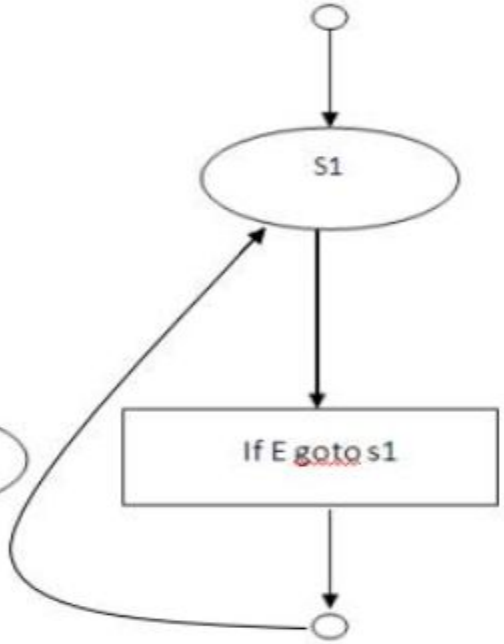
- A definition d reaches a point p if there is a path from the point immediately following d to p and also d is not “killed”.



S1 ; S2



IF E then S1 else S2



do S1 while E

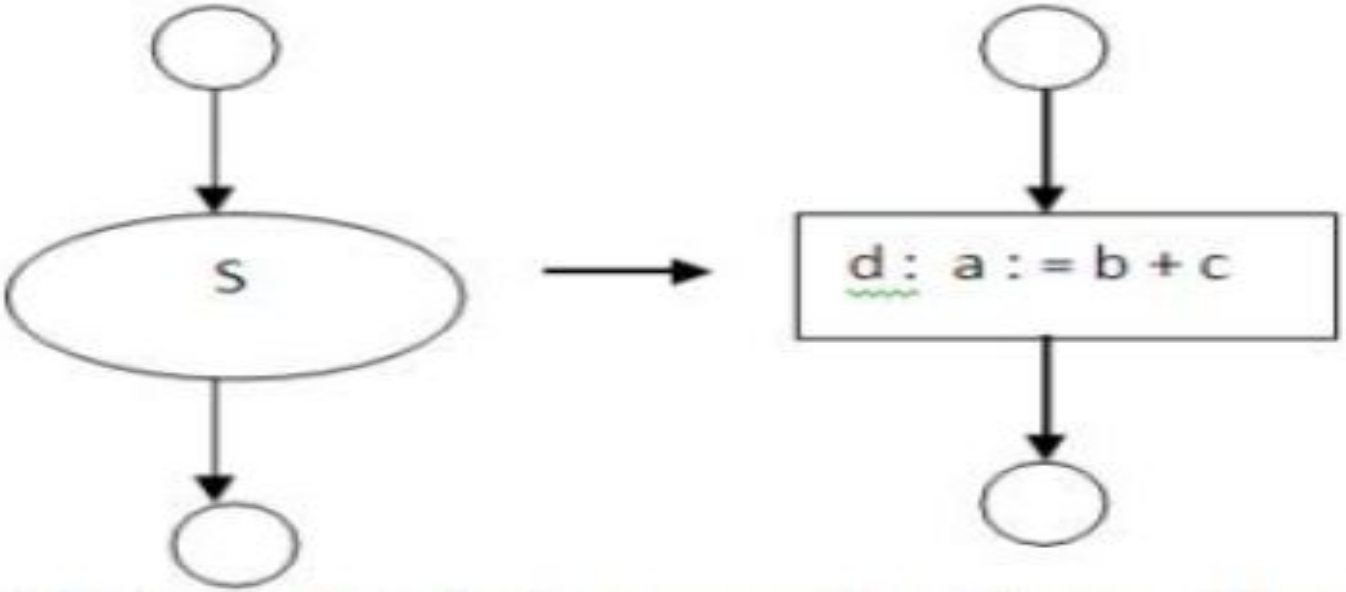
gen[S]- The set of definitions generated by S. It is the set of definitions that reach the end of S without following paths outside S.

kill[S]- is the set of definitions that never reach the end of S.

In[S]-Input of S

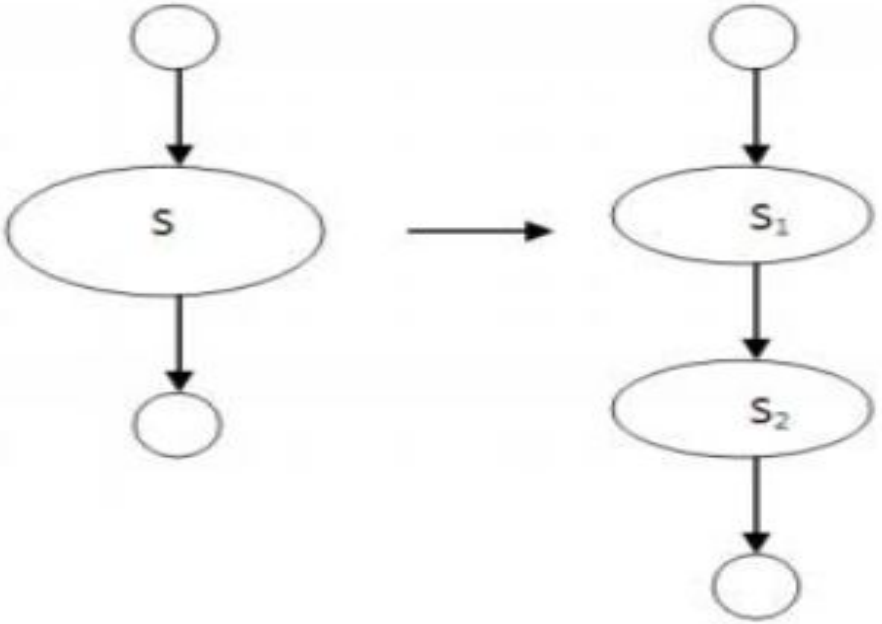
Out[S]-output of S. This takes into account the paths outside S also

Data-flow analysis of structured programs



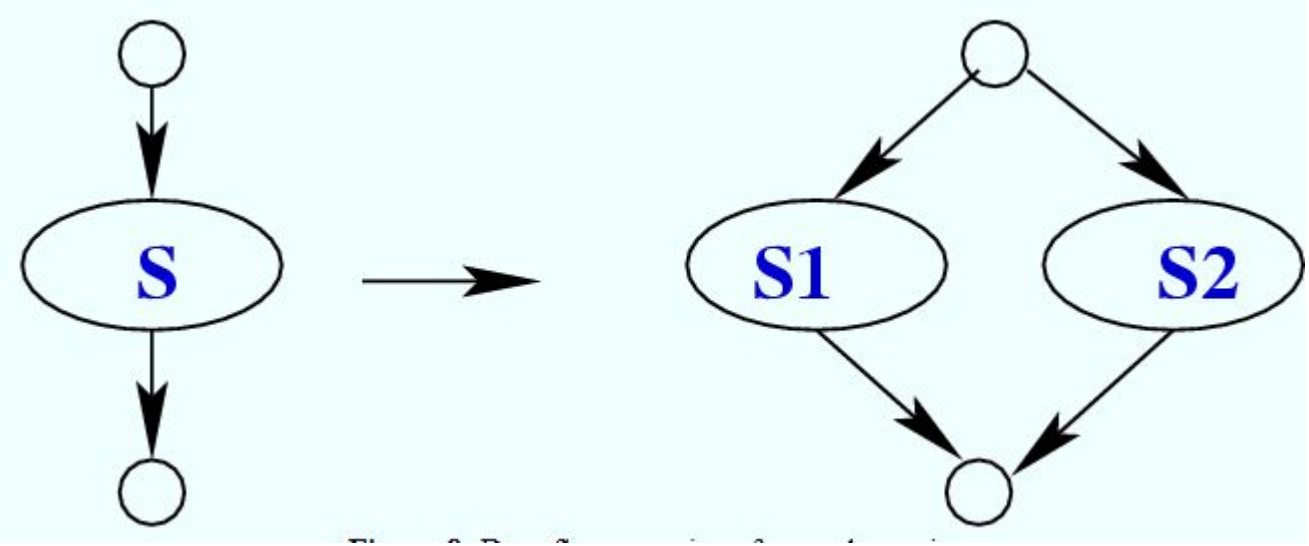
gen [S] = { d }
kill [S] = Da - { d }
out [S] = gen [S] U (in[S] - kill[S])

Da is the set of all definitions in the program for variable a.

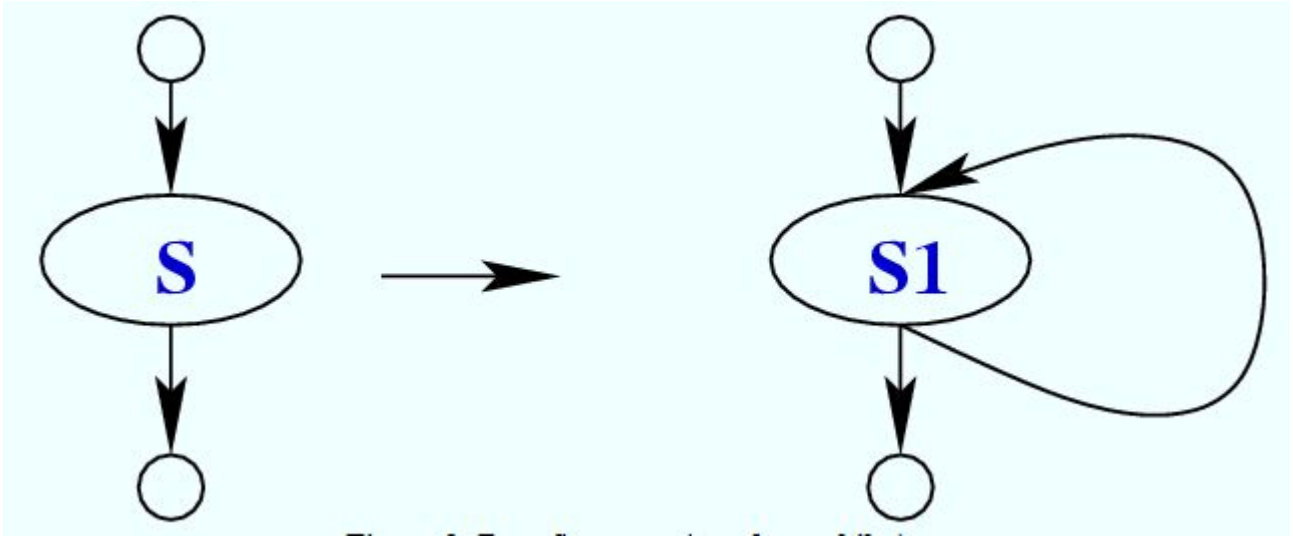


gen[S]=gen[S2] U (gen[S1]-kill[S2])
Kill[S] = kill[S2] U (kill[S1] - gen[S2])

in [S1] = in [S]
in [S2] = out [S1]
out [S] = out [S2]

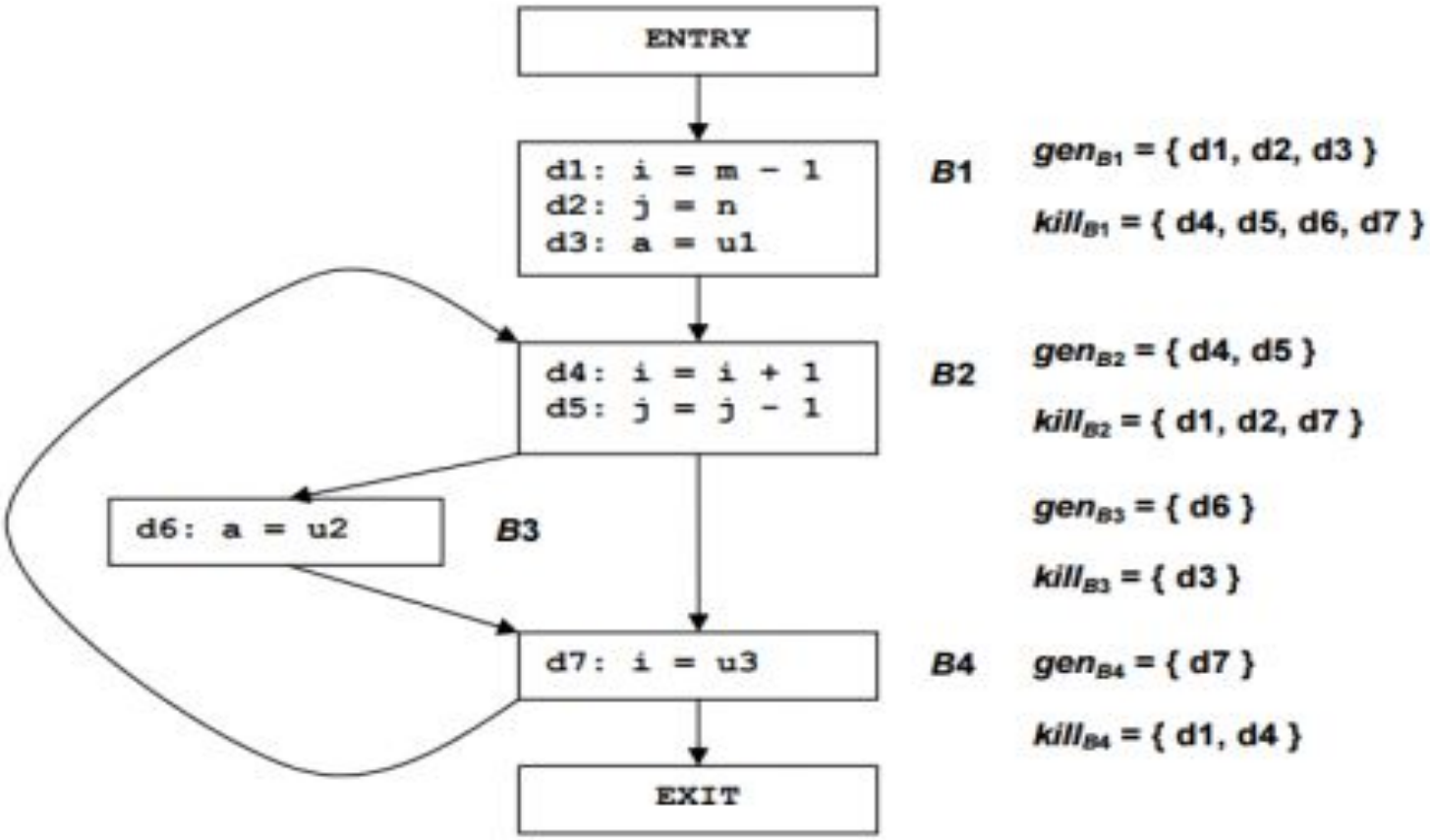


Gen (S)= gen (S1) U Gen(s2)
Kill (S)=Kill (S1) U Kill (S2)
In (S1)=in (S)
In(S2)=in(S)
Out(S)= out (S1) U out (S2)



Gen (S)= gen (S1)
Kill (S)=Kill (S1)
In (S1)=in (S) U gen (S1)
Out(S)= out (S1)

Example for reaching definitions



Example for reaching definitions

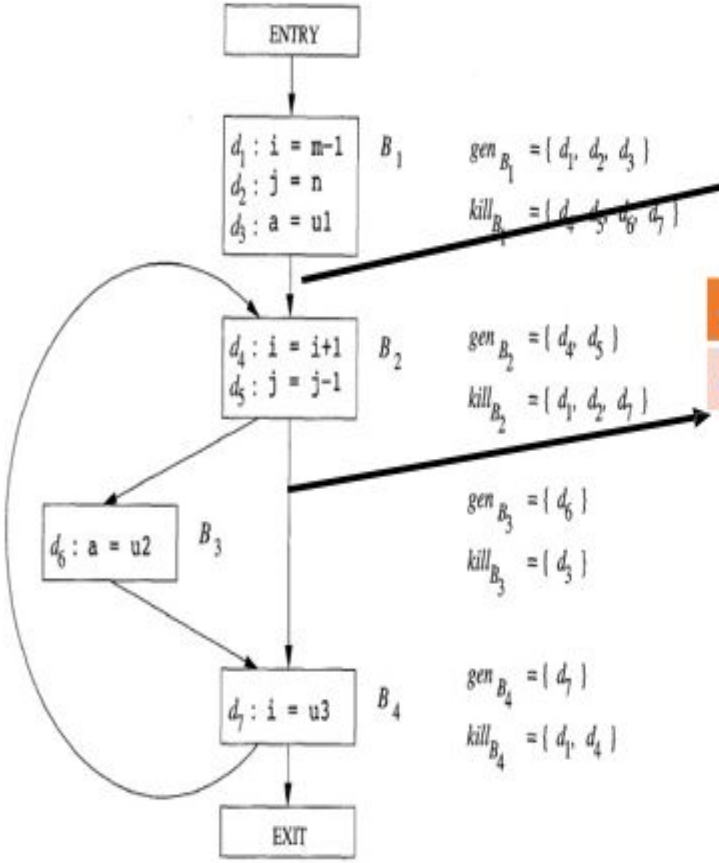


Figure 9.13: Flow graph for illustrating reaching definitions

d1	d2	d3	d4	d5	d6	d7
1	1	1	0	0	0	0

d1	d2	d3	d4	d5	d6	d7
0	0	1	1	1	0	0

Block B	OUT[B]0	IN[B]1	OUT[B]1
B1	000 0000	000 0000	111 0000
B2	000 0000	111 0000	001 1100
B3	000 0000	001 1100	000 1110
B4	000 0000	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111

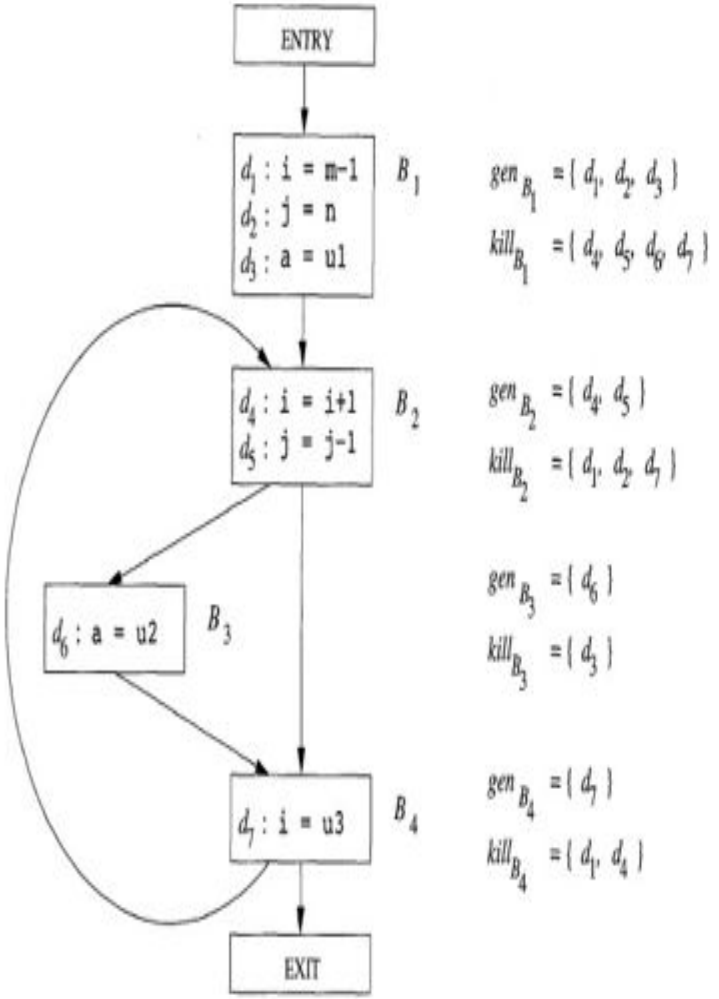


Figure 9.13: Flow graph for illustrating reaching definitions

Block B	OUT$[B]^0$	IN$[B]^1$	OUT$[B]^1$	IN$[B]^2$	OUT$[B]^2$
B_1	000 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	111 0000	001 1100	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	000 1110
B_4	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111	001 0111	001 0111

Figure 9.15: Computation of IN and OUT

$$\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$$

$$\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P].$$

Then we consider $B = B_2$ and compute

$$\begin{aligned}\text{IN}[B_2]^1 &= \text{OUT}[B_1]^1 \cup \text{OUT}[B_4]^0 \\ &= 111\ 0000 + 000\ 0000 = 111\ 0000 \\ \text{OUT}[B_2]^1 &= gen[B_2] \cup (\text{IN}[B_2]^1 - kill[B_2]) \\ &= 000\ 1100 + (111\ 0000 - 110\ 0001) = 001\ 1100\end{aligned}$$

- At the end of the first pass, 0UT[B2]1 = 001 1100, reflecting the fact that *d4* and *d5* are generated in B2, while *d3* reaches the beginning of B2 and is not killed in B2.
- Notice that after the second round, 0UT[B2] has changed to reflect the fact that *d6* also reaches the beginning of B2 and is not killed by B2.
- We did not learn that fact on the first pass, because the path from *d6* to the end of B2, which is *B3* B4 B2, is not traversed in that order by a single pass.
- That is, by the time we learn that *d6* reaches the end of B4, we have already computed IN[B2] and OUT[B2] on the first pass.

Local reaching definitions

- Save information only at certain points.
- Instead of saving in points , it could be done for basic blocks.

Use Definition Chains (ud-chains)

- Storing reaching definitions
- It is a list which stores for each use of a variable, of all the definitions that reaches that use.

Evaluation order

- This will be a problem in gen, kill, in and out sets for statements that has dependencies.
- After choosing an evaluation order, we can release the space for a set after all uses of it have occurred.

General Control Flow

- Data-flow analysis must take all control paths into account.
- When programs can contain goto, break, continue statements then there will be modification in the actual control paths.

RUN TIME ENVIRONMENTS

SOURCE LANGUAGE ISSUES

- Procedures
- Activation trees
- Control stack
- Scope of declaration
- Binding of names

Procedure activations

- 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:

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
The activation of q, or some procedure q called, either directly or indirectly, aborts	p ends simultaneously with q.
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

Activation Trees

- The activations of procedures during the running of an entire program can be represented as tree structure called as activation tree.
- In the tree each node corresponds to one activation.
- The root is the activation of the "main" procedure that initiates execution of the program.
- At a node for an activation of procedure p, the children correspond to activations of the procedures called by this activation of p.
- We show these activations in the order that they are called, from left to right.
- Notice that one child must finish before the activation to its right can begin.

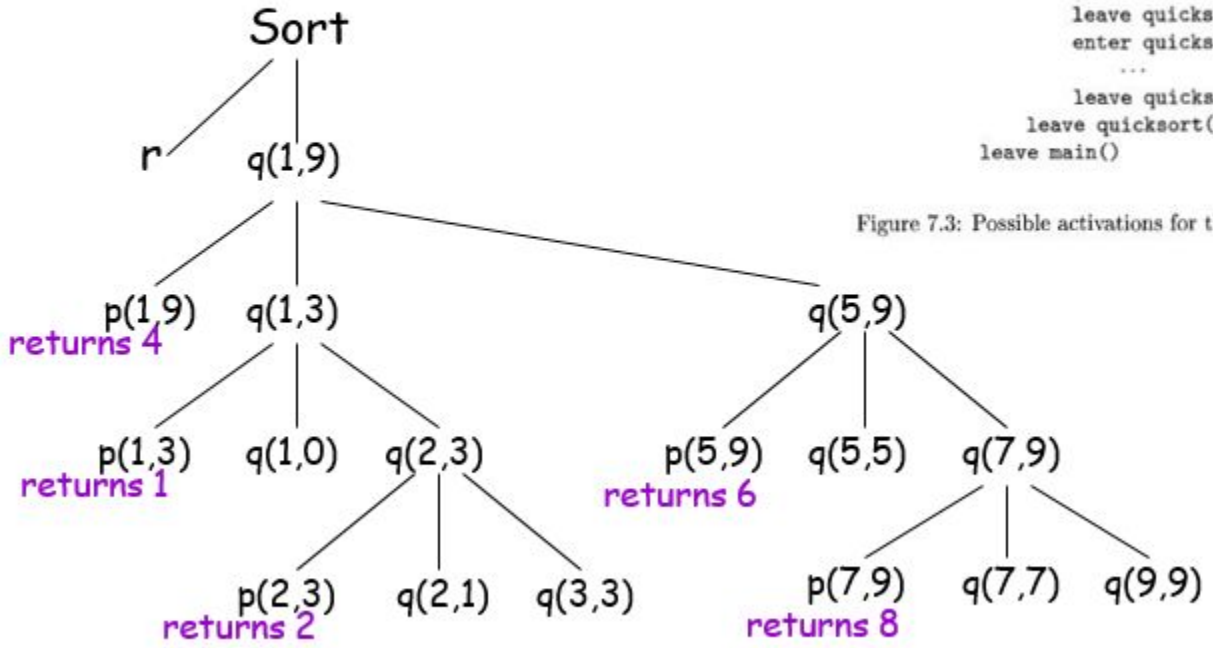
Activation Trees

- The activations of procedures during the running of an entire program can be represented as tree structure called as activation tree.
- In the tree each node corresponds to one activation.
- The root is the activation of the "main" procedure that initiates execution of the program.
- At a node for an activation of procedure p, the children correspond to activations of the procedures called by this activation of p.
- We show these activations in the order that they are called, from left to right.
- Notice that one child must finish before the activation to its right can begin.

Activation Tree

```
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()
```

Figure 7.3: Possible activations for the program of Fig. 7.2

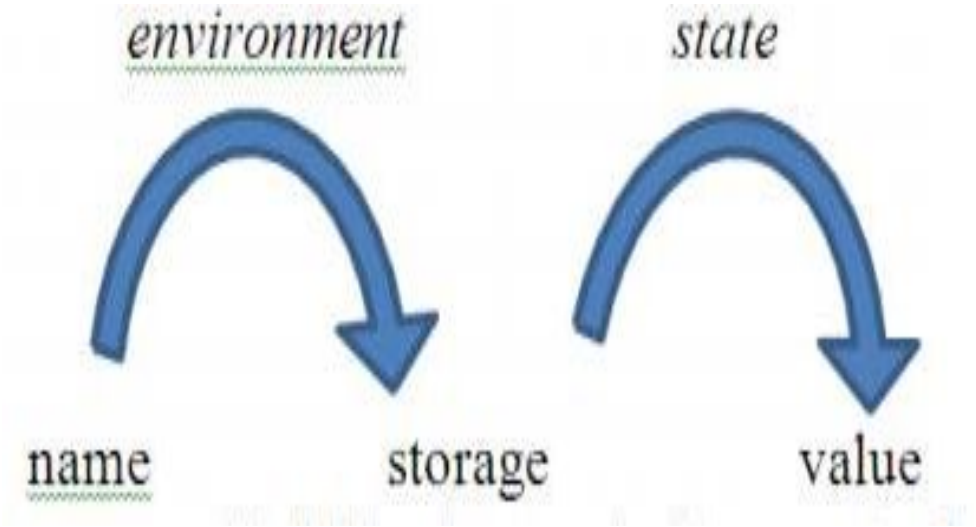


Control stack

- Flow of control in program corresponds to depth first traversal of activation tree
- Use a stack called control stack to keep track of live procedure activations
- Push the node when activation begins and pop the node when activation ends
- When the node n is at the top of the stack the stack contains the nodes along the path from n to the root

- **Binding of names**
- Same name may denote different data objects at run time.
- **Data object** is a storage location that holds values.
- **Environment** is a function that maps a name to a storage location.
- **State** refers to a function that maps a storage location to the value held there.
- When an environment associates storage location s with a name x, we say that x is bound to s. This association is referred to as a **binding** of x.

Binding of names



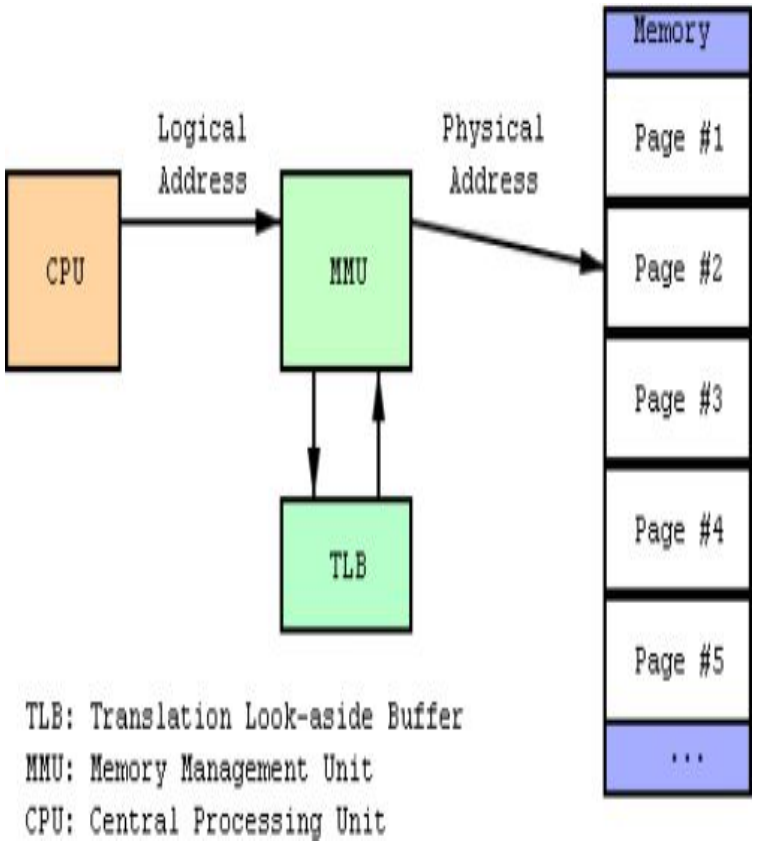
STORAGE ORGANISATION

- The operating system maps the logical address into physical addresses.
- The storage layout is influenced by the addressing constraints of the target machine.
- A character array of length 10 needs only enough bytes to hold 10 characters, a compiler may allocate 12 bytes to get alignment, leaving 2 bytes unused.
- This unused space due to alignment considerations is referred to as **padding.**
- Activation records maintain the variable information.

STORAGE ORGANISATION

From the perspective of the compiler writer:

- The executing target program runs in its own logical address space in which each program value has a location.
- The management and organization of this logical address space is shared between the ***compiler, operating system, and target machine***.
- The *operating system maps the logical addresses into physical addresses*, which are usually spread throughout memory.



BASIS FOR COMPARISON	LOGICAL ADDRESS	PHYSICAL ADDRESS
Basic	It is the virtual address generated by CPU	The physical address is a location in a memory unit.
Address Space	Set of all logical addresses generated by CPU in reference to a program is referred as Logical Address Space.	Set of all physical addresses mapped to the corresponding logical addresses is referred as Physical Address.
Visibility	The user can view the logical address of a program.	The user can never view physical address of program
Access	The user uses the logical address to access the physical address.	The user can not directly access physical address.

STORAGE ALLOCATION STRATEGIES

1. A **statically** determined **area Code** that holds the executable target code. The size of the target code can be determined at compile time.
2. A **statically** determined **data area** Static for holding global constants and other data generated by the compiler. The size of the global constants and compiler data can also be determined at compile time.
3. A **dynamically** managed **area Heap** for holding data objects that are allocated and freed during program execution. The size of the Heap cannot be determined at compile time.
4. A **dynamically** managed **area Stack** for holding activation records as they are created and destroyed during procedure calls and returns. Like the Heap, the size of the Stack cannot be determined at compile time.

STORAGE ALLOCATION STRATEGIES

Three strategies

- Static allocation
- Stack allocation
- Heap allocation

HEAP ALLOCATION

- Stack allocation strategy cannot be used when

1. The values of local names must be retained when an activation ends.
2. A called activation outlives the caller.

- Heap allocation uses pieces of contiguous storage, as demanded by activation records or other objects.
- The memory may be deallocated in any order, so over the time the heap will consist of alternate areas that are free and in use.
- The heap is the portion of the store that is used for data that lives indefinitely, or until the program explicitly deletes
- Memory Manager: The memory manager keeps track of all the free space in heap storage at all times. It performs two basic functions

Heap (…)

- **Allocation. When a program requests memory for a variable or object, the memory manager produces a chunk of contiguous heap memory of the requested size. If space is exhausted, the memory manager passes that information back to the application program**
- ***Deallocation*. The memory manager returns deallocated space to the pool of free space, so it can reuse the space to satisfy other allocation requests.**

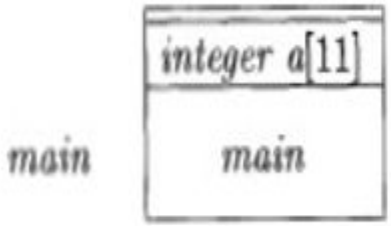
Required properties of memory management in heap

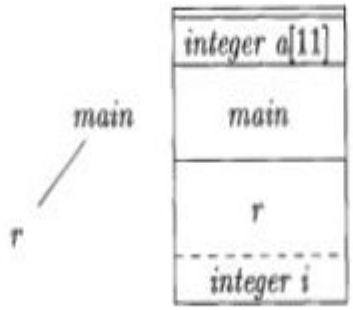
- ***Space Efficiency***. A memory manager should minimize the total heap space needed by a program
- ***Program Efficiency***. *A* memory manager should make good use of the memory subsystem to allow programs to run faster. By attention to the placement of objects in memory, the memory manager can make better use of space and, hopefully, make the program run faster
- ***Low Overhead***. Because memory allocations and deallocations are frequent operations in many programs, it is important that these operations be as efficient as possible

Activation Records

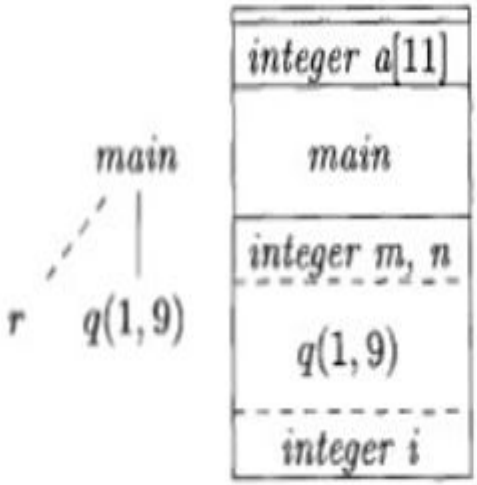
- Procedure calls and returns are usually managed by a run-time stack called the control stack. Each live activation has an activation record.
- The root of the activation tree will be at the bottom of the stack
- The entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides.
- The latter activation has its record at the top of the stack

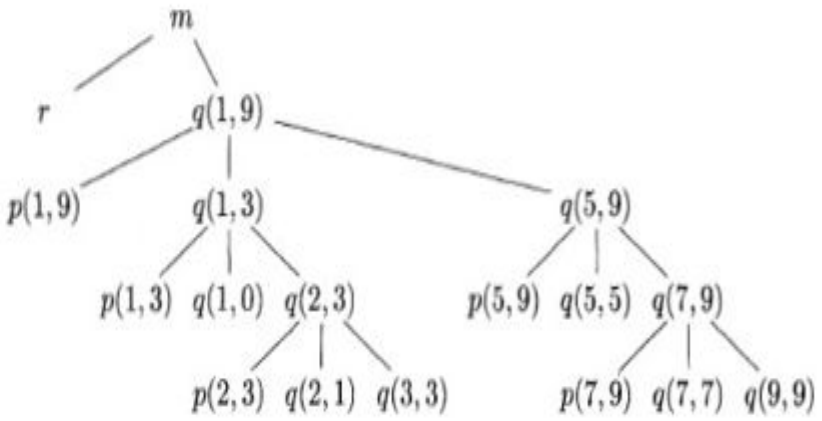
Actual parameters	The actual parameters used by the calling procedure
Returned values	Space for the return value of the called function
Control link	A control link, pointing to the activation record of the caller
Access link	An "access link" may be needed to locate data needed by the called procedure
Saved machine status	information about the state of the machine just before the call to the procedure
Local data	Local data belonging to the procedure whose activation record this is
Temporaries	Temporary values, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers



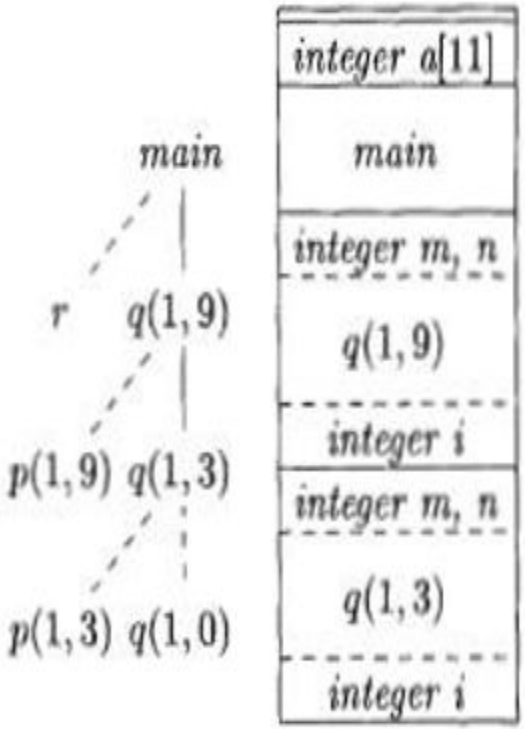


- Procedure *r* is activated When control reaches the first call in the body of main and its activation record is pushed onto the stack.
- The activation record for *r* contains space for local variable i.

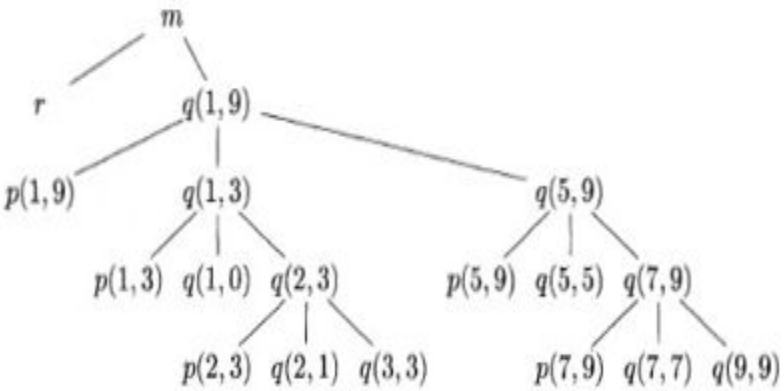




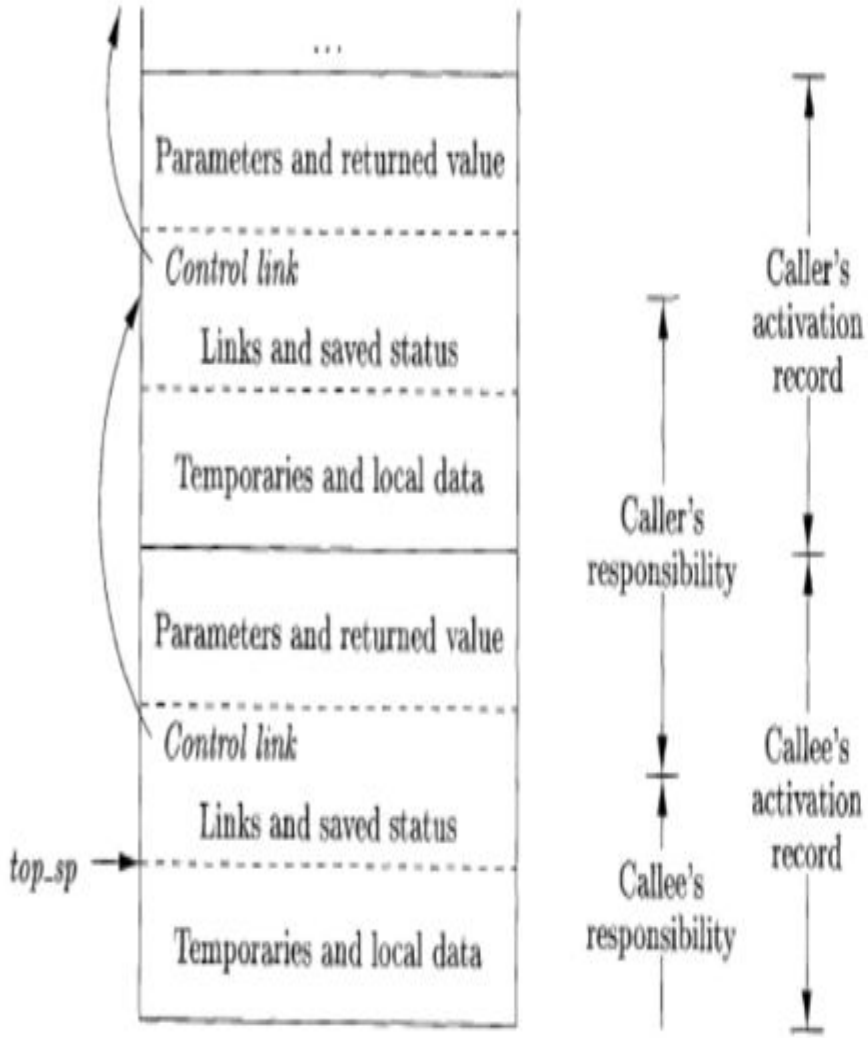
- When control returns from this activation, its record is popped, leaving just the record for main on the stack.
- Control then reaches the call to q (quicksort) with actual parameters 1 and 9, and an activation record for this call is placed on the top of the stack.



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



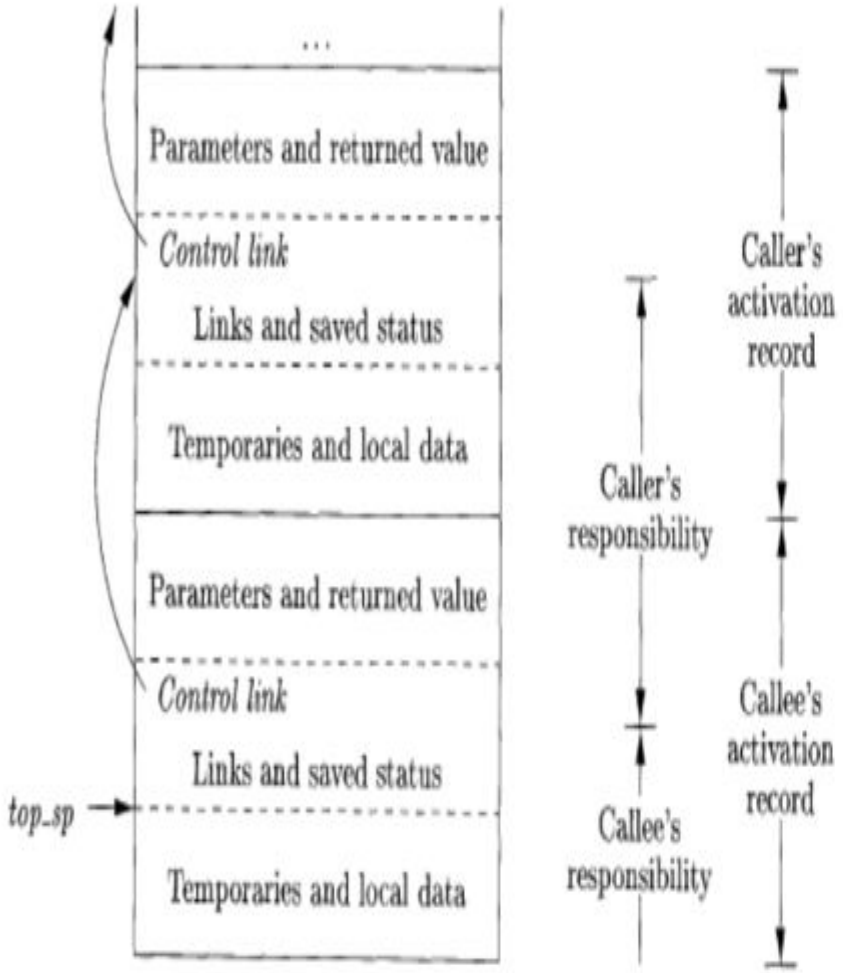
- The activation record for q contains space for the parameters m and n and the local variable ***i***.
- Notice that space once used by the call of *r* is reused on the stack.
- No trace of data local to *r* will be available to q(1,9).
- When q(1,9) returns, the stack again has only the activation record for main.
- A recursive call to q(1,3) was made.
- Activations p(1,3) and q(1,0) have begun and ended during the lifetime of q(I,3), leaving the activation record for q(I,3) on top



Actual parameters
Returned values
Control link
Access link
Saved machine status
Local data
Temporaries

- ✓ The caller evaluates the actual parameters.
- ✓ 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.
- ✓ That is, *top-sp* is moved past the caller's local data and temporaries and the callee's parameters and status fields.
- ✓ The callee saves the register values and other status information.
- ✓ The callee initializes its local data and begins execution

Caller and callee sequence



Actual parameters
Returned values
Control link
Access link
Saved machine status
Local data
Temporaries

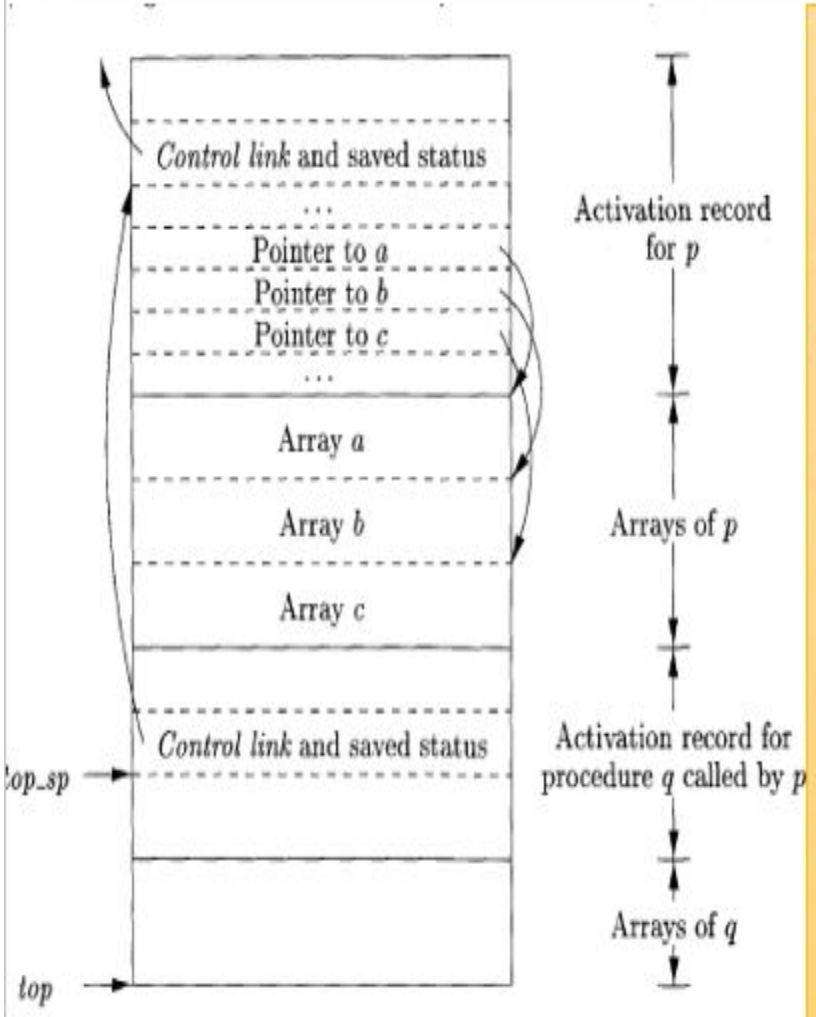
Return sequence

- The callee places the return value next to the parameters.
- Using information in the machine-status field, the callee restores *top-sp* and other registers, and then branches to the return address that the caller placed in the status field.
- Although *top-sp* has been decremented, the caller knows where the return value is, relative to the current value of *top-sp;* the caller therefore may use that value.

Variable-Length Data on the Stack

- The run-time memory-management system must deal frequently with the allocation of space for objects the sizes of which are not known at compile time
- But which are local to a procedure and thus may be allocated on the stack.
- ***In modern languages, objects whose size cannot be determined at compile time are allocated space in the heap***

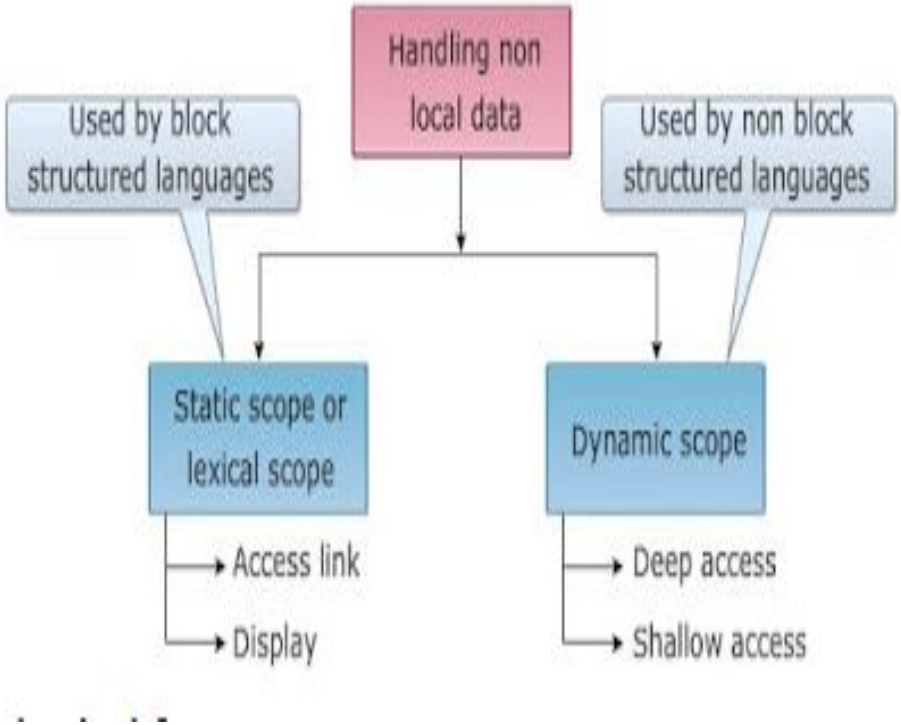
- Let consider procedure p has three local arrays, whose sizes be determined at compile time.
- The storage for these arrays is not part of the activation record for p.
- Only a pointer to the beginning of each array appears in the activation record itself.
- Thus, when p is executing, these pointers are at known offsets from the top-of-stack pointer, so the target code can access array elements through these pointers.



- The activation record for a procedure q, called by p.
- The activation record for q begins after the arrays of p, and any variable-length arrays of q are located beyond that.
- ✓ **Access to the data on the stack is through two pointers, *top* and *top-sp*.**
- ✓ **Here, *top* marks the actual top of stack; it points to the position at which the next activation record will begin.**
- ✓ **The second, *top-sp* is used to find local, fixed-length fields of the top activation record.**
  - **The code to reposition *top* and *top-sp* can be generated at compile time in terms of sizes that will become known at run time.**
  - **When q returns, *top-sp* can be restored from the saved control link in the activation record for q.**
  - **The new value of *top* is *top-sp* minus the length of the machine-status, control and access link, return-value, and parameter fields in q's activation record.**
  - **This length is known at compile time**

ACCESS TO NON LOCAL NAMES

- Now let us see how a procedure access their data that not belong to its own
- That is the mechanism for finding data used within a procedure p but that does not belong to p.
- Access becomes more complicated in languages where procedures can be declared inside other procedures.
- There are two types of scope rules, for the non-local names- Static scope and Dynamic scope.



Static or Lexical Scope

- Here , scope is verified by examining the text of the program.
- Eg-PASCAL, C and ADA -block structured languages.
- **Block –**This defines a new scope with a sequence of statements that contains the local data declarations. It is enclosed within the delimiters.

Data Access Without Nested Procedures

The ***global variable*** v has a scope consisting of all the functions that follow the declaration of v, except where there is a local definition of the identifier v.

For languages that do not allow nested procedure declarations, allocation of storage for variables and access to those variables is simple

1. **Global variables are allocated static storage.**
   - The locations of these variables remain fixed and are known at compile time.
   - So to access any variable that is not local to the currently executing procedure, we simply use the statically determined address.
2. **Any other name must be local to the activation at the top of the stack.**
   - We may access these variables through the top-sp pointer of the stack.

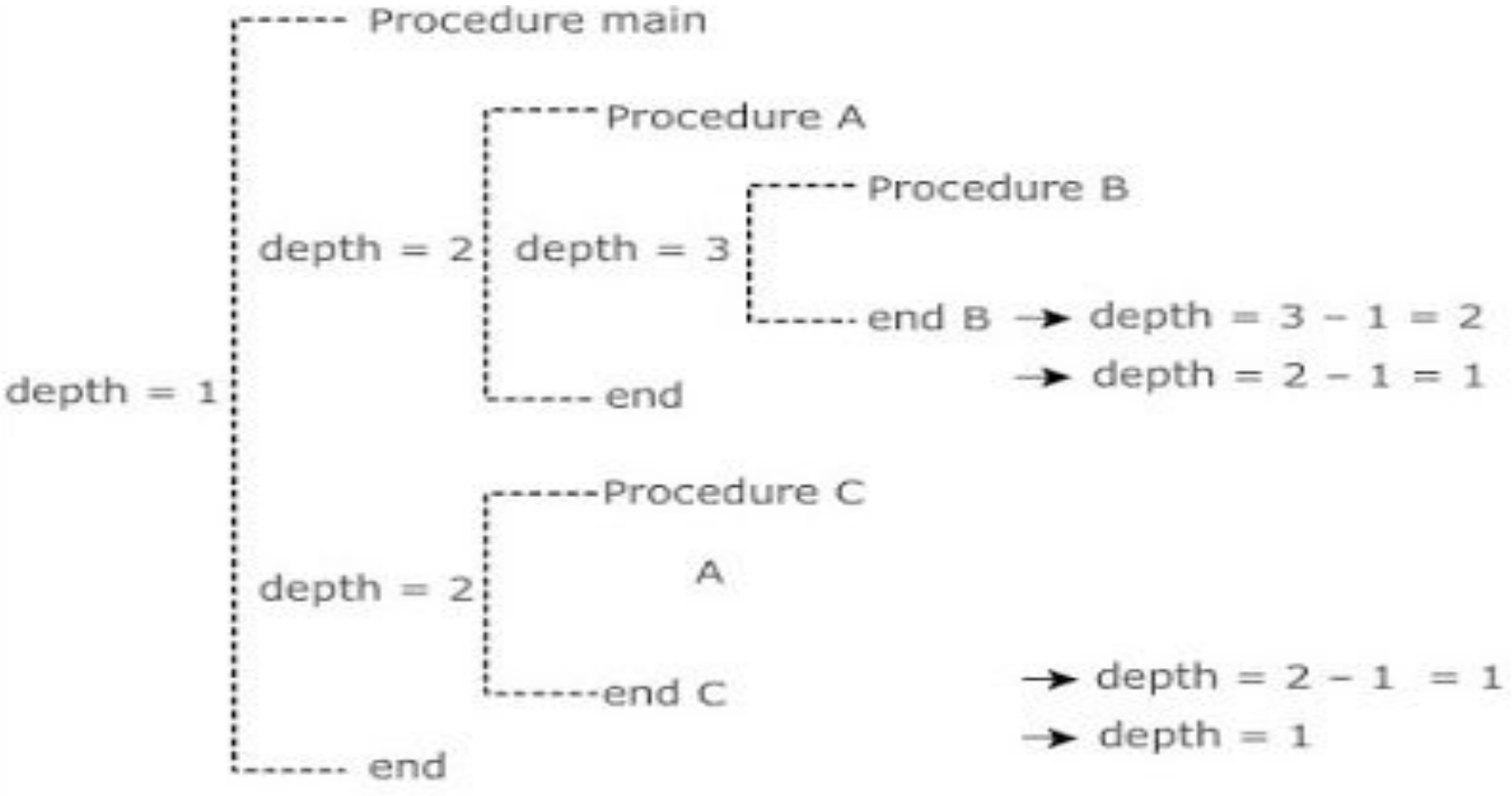
Data Access With Nested Procedures

- Access becomes far more complicated when a language allows procedure declarations to be nested and also uses the normal static scoping rule;
- Let us give **nesting depth** 1 to procedures that are not nested within any other procedure

Lexical scope for nested procedures

Procedure main-> Procedure A -> Procedure B

Procedure main-> Procedure C



Blocks…

- The beginning and end of the block are specified by the delimiter.
- At a program point, declarations are visible

✔ made inside the procedure

✔ names of all enclosing procedures

✔ names declared made immediately within such procedures

- Block storage allocation is implemented by stack.
- Lexical scope is implemented by **nesting depth, access link and displays.**

Nesting Depth

- Lexical scope isimplemented by using nesting depth of a procedure.
- The procedure of calculating nesting depth is as follows:

1. The main programs nesting depth is ‘1.
2. When a new procedure begins, add ‘1’ to nesting depth each time.
3. When you exit from a nested procedure, subtract ‘1’ from depth each time

- The variable declared in specific procedure is associated with nesting depth.

Access Link:

- Access links are pointer to each activation record.
- If procedure p is nested within a procedure q then access link of p points to access link or most recent activation record of procedure q.
- A direct implementation of the normal static scope rule for nested functions is obtained by adding a pointer called the access link to each activation record.
- Note that the nesting depth of q must be exactly one less than the nesting depth of p.

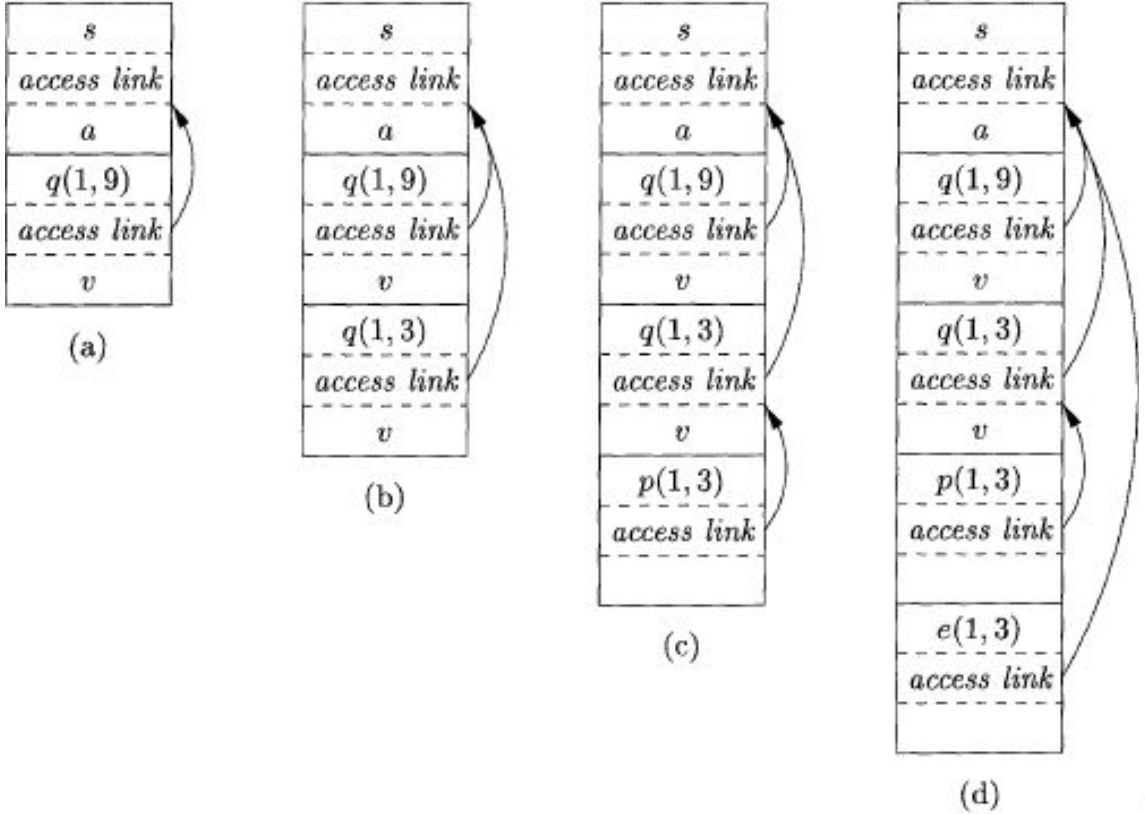


Figure 7.11: Access links for finding nonlocal data

```
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()
```

Figure 7.3: Possible activations for the program of Fig. 7.

- After sort has called readArray to load input into the array *a* and then called quicksort(1,9) to sort the array.
- The access link from quicksort(1,9) points to the activation record for sort, not because sort called quicksort but because sort is the most closely nested function surrounding quicksort in the program
- In successive steps of recursive call to quicksort(1,3), followed by a call to partition, which calls exchange.
- The quicksort(1,3)'s access link points to sort, for the same reason that quicksort(1,9)'s does.
- The access link for exchange bypasses the activation records for quicksort and partition, since exchange is nested immediately within sort.
- That arrangement is fine, since exchange needs to access only the array *a*, and the two elements it must swap are indicated by its own parameters ***i*** and *j*.

Parameter Passing

- Call by value
  - actual parameters are evaluated and their r-values are passed to the called procedure
  - caller evaluates the actual parameters and places r value in the storage for
  - formals
  - call has no effect on the activation record of caller
- Call by reference (call by address)
  - the caller passes a pointer to each location of actual parameters
  - if actual parameter is a name then l-value is passed
  - if actual parameter is an expression then it is evaluated in a new location and the address of that location is passed

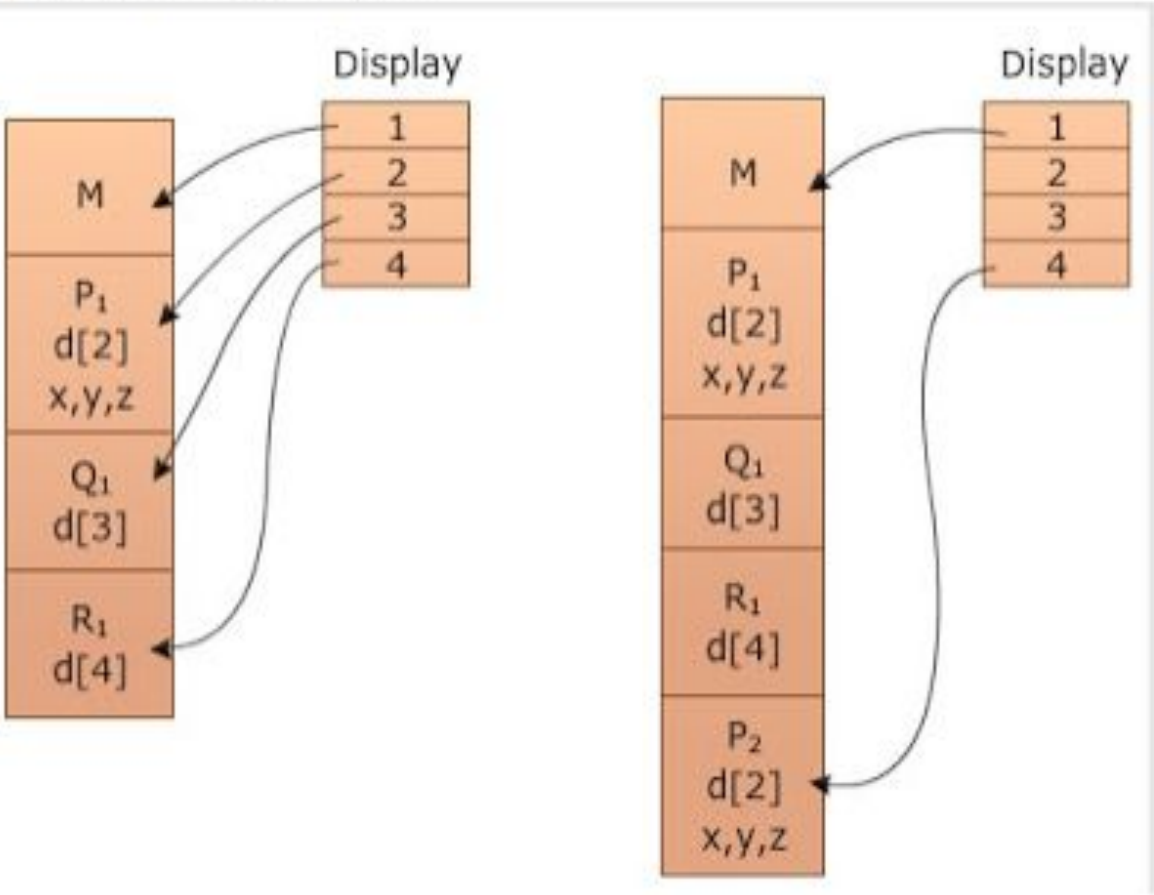
Displays

- Access links makes the computations slower.
- Activation record can be accessed from direct location instead of pointers.
- Display is a **global array (d) of pointers** to activation records, indexed by lexical nesting depth.
- d[i] is an array element which points to the most recent activation of the block.
- A nonlocal X is found in the following manner:
  - through one array access to find the activation record containing X.
  - Use relative address within the activation record

Displays

- Access links makes the computations slower.
- Activation record can be accessed from direct location instead of pointers.
- Display is a **global array (d) of pointers** to activation records, indexed by lexical nesting depth.
- d[i] is an array element which points to the most recent activation of the block.
- A nonlocal X is found in the following manner:
  - through one array access to find the activation record containing X.
  - Use relative address within the activation record

How to maintain display information?



› When a procedure is called, a procedure 'p' at nesting depth 'i' is setup:

Save value of d[i] in activation record for 'p'

'I' set d[i] to point to new activation record

› When a 'p' returns:

Reset d[i] to display value stored