# RUNTIME ENVIRONMENTS

Based on Chapter 7 of Aho, Lam, Sethi, Ullman:

*Compilers: Principles, Techniques, & Tools*

2nd Ed, Addison Wesley, 2007

# Table of Contents

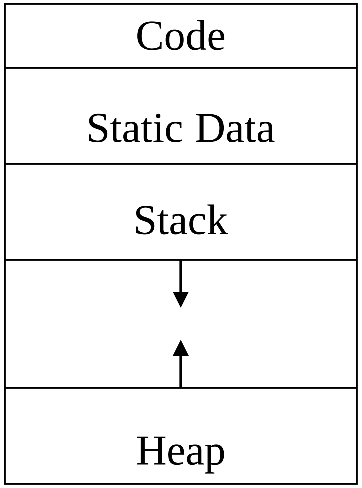

- Introduction
- Storage Organizations
- Storage Allocation Strategies
  - static allocation
  - stack allocation
  - dynamic allocation
- Access to Nonlocal Names
  - block
  - lexical scope without nested procedures
  - lexical scope with nested procedures
  - dynamic scope
- Parameter Passing

# Introduction

- Runtime environment
  - a set of data structures maintained at runtime to implement high-level structures of programming languages (such as procedures, variables, etc.) using low level machine structures (memory, register, etc.)
- Issues
  - organization of storage for data objects
  - allocation and deallocation of data objects
  - linkage between procedures
  - parameter passing mechanisms
  - runtime support packages
- Three kinds of runtime environment
  - static environment (FORTRAN 77)
  - stack-based environment (C, Pascal, C++, etc.)
  - dynamic environment (LISP)

# Storage Organization

- Subdivision of Runtime Memory
  - the generated target code
  - data objects
  - control stack to keep track of procedure activations



# Storage Organization

- Compile-time Layout of Local Data
  - the amount of storage needed for a name is determined from its type
  - storage for aggregates is allocated in one contiguous block
  - the field for local data is laid out as the declaration in a procedure are examined at compile time
  - A relative address for a local data is determined with respect to some position  (e.g. the beginning of activation record)
  - Storage layout for data objects is strongly influenced by the addressing constraints of the target machine
    - alignment
    - space padding

# Storage Allocation Strategies

- Three strategies
  - static allocation layout storage for all data objects at compile time
  - stack allocation manages the runtime storage as a stack
  - heap allocation allocates and deallocates storage as needed at run time from a data area know as a heap

# Static Allocation

- Static allocation
  - names are bound to storage at compile time
  - no need for a run time support package
  - every time a procedure is activated, its names are bound to the same storage
  - names have offsets within the activation record
  - activation records placed relatively from target code

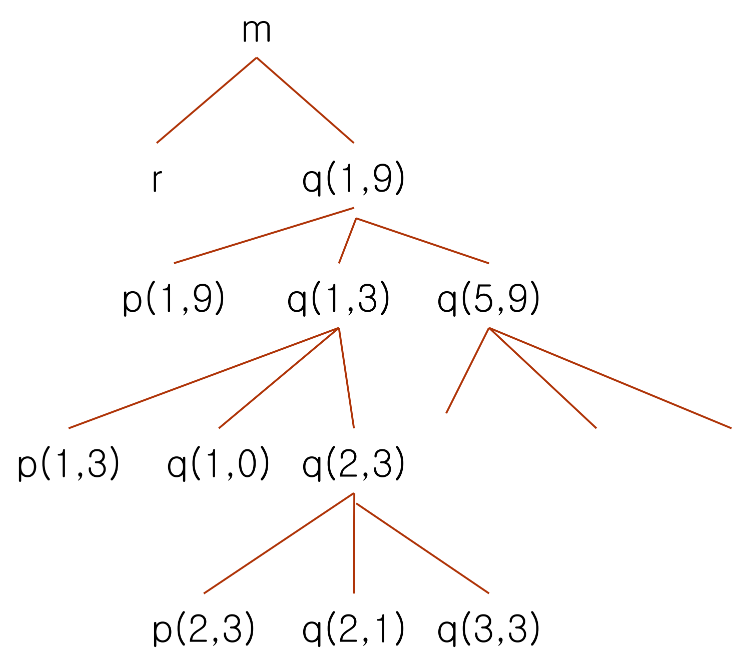
- Some Limitations
  - the size of a data object must be known at compile time
  - recursive procedures are restricted
  - data structure cannot be created dynamically

# Stack Allocation

- Stack Allocation
  - storage is organized as a stack (runtime stack)
  - activation records are pushed and popped as activations begin and end, respectively
- Activation Trees

```
int a[11];
void readArray() {
  int i;
}
int partition(int m, int n) {
 ….
}
void quicksort(intm, 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);
}
```



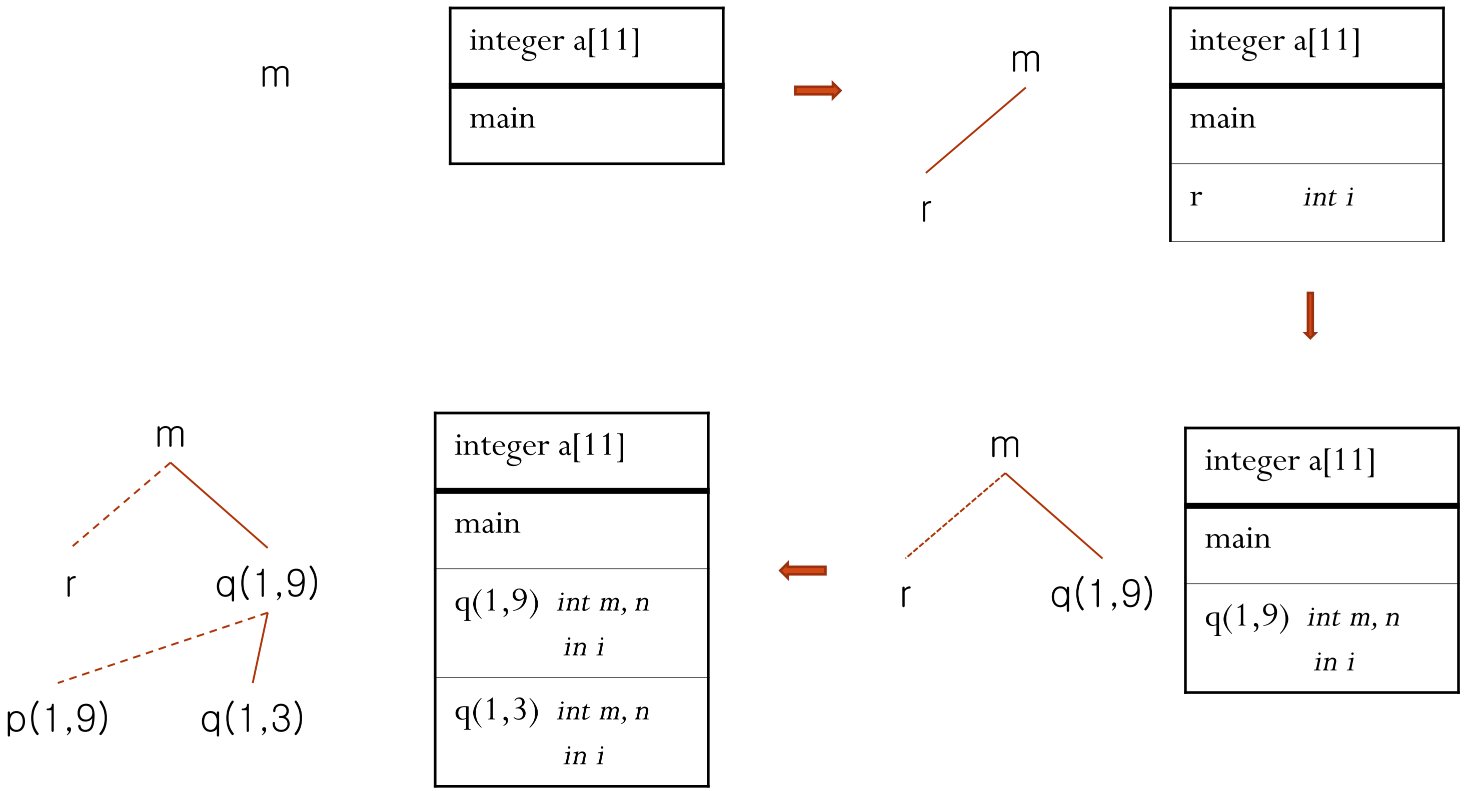
# Stack Allocation

- Activation Records
  - a contiguous block of storage containing information needed by a single execution of a procedure
  - in a stack-based runtime environment, activation records are managed on the runtime-stack
- Fields of an activation record
  - temporary data
  - local data
  - saved machine status
    - (program counter, machine registers etc.)
  - optional access link
  - control link
  - actual parameters
  - return value

| |
|---|
| actual parameters |
| return value |
| control link |
| access link |
| machine status |
| local data |
| temporaries |

# Stack Allocation

- Snapshots of Runtime Stack



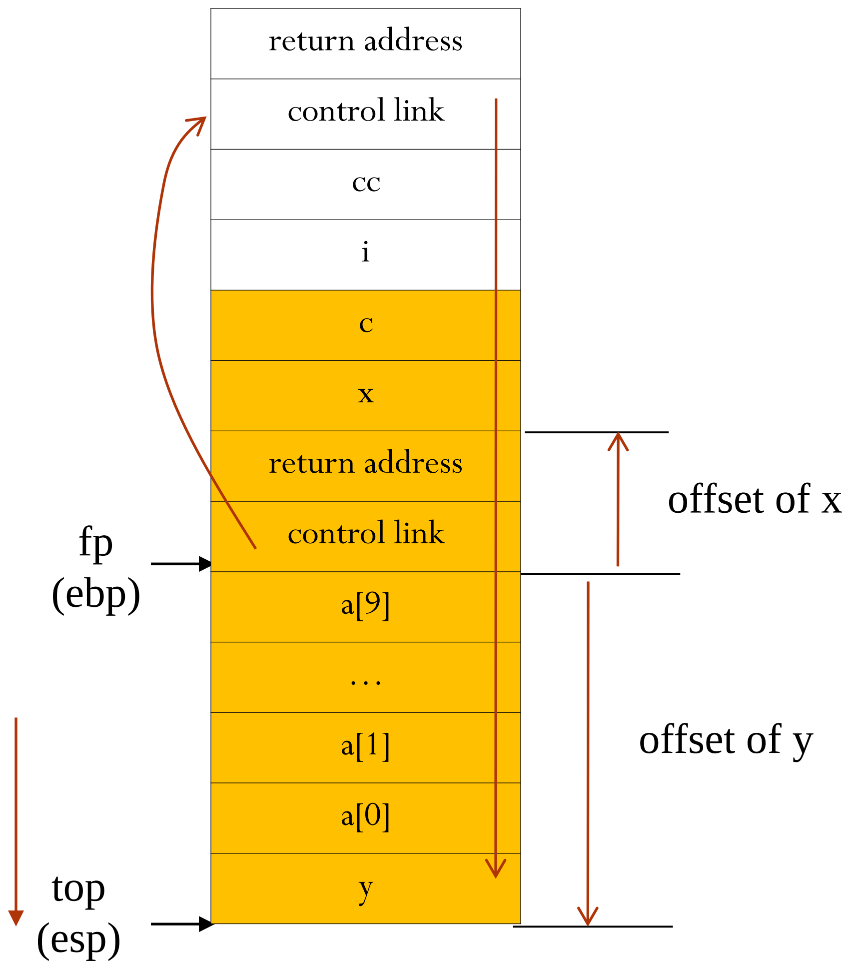
# Stack Allocation

- Calling Sequences
  - call sequence
    - the caller evaluates and store actual parameters
    - the caller stores a return address
    - the caller stores the old value of frame pointer(fp) into the callee's activation record.
    - the caller increments frame pointer(fp) to the callee's activation record
    - the callee saves register values and other status
    - the callee initializes its local data and begins execution
  - return sequence
    - the callee places a return value (possibly via a register)
    - the callee restores fp and other registers
    - the callee branches to the return address
    - the caller pops parameters from stack

# Stack Allocation

- Calling Sequences

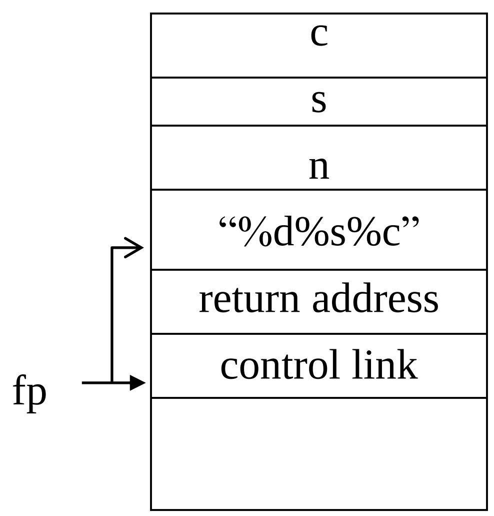
```
double f(int x, char c)
{ int a[10];
  double y;
  …
  return y;
}
int main()
{  int i;
   char cc;
   i  = f(i,cc);
   …
}
```



# Stack Allocation

- Dealing with Variable-Length Data
  - the number of arguments in a call may vary
  - the size of an array parameter or a local array variable may vary
- Variable number of arguments
  - e.g. printf in C
  - pushing the arguments to a call in reverse order
    - first argument can always be located at a fixed offset from *fp*

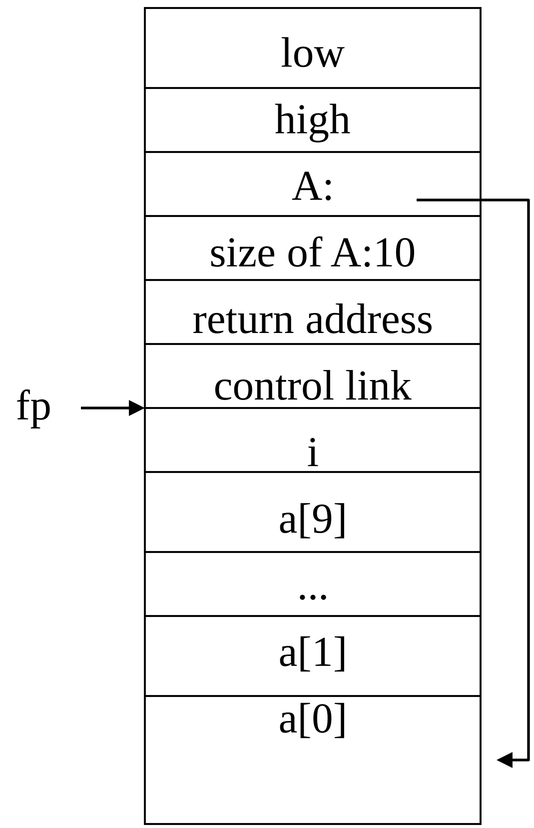
printf("%d%s%c",n,s,c);



# Stack Allocation

- Variable-length array
  - using an extra indirection
    - storing actual data at the top of the runtime stack(below sp)
    - storing a pointer to the actual data
  - example (Ada)

```
type Int_Vector is array(INTEGER range<>)
    of INTEGER;
procedure Sum (low,high: INTEGER;
        A: Int_Vector) return INTEGER
is
    i: INTEGER
begin
    …
end Sum;
```



# Access to Nonlocal Data

- Two types of scope rule
  - Lexical or static-scope rule : C-based languages, Pascal, ADA,, PHP, etc.
  - Dynamic scope rule: LISP, APL, Snobol, Perl etc.
- Lexical scope rule
  - Block structure and most closely nested rule
  - With nested procedures or without nested procedures
    - Not supporting nested procedures : C, C++, Java, etc
    - Supporting nested procedures : Pascal, Ada, JavaScrip, PHP, etc.
- Access to nonlocal data in nested procedures
  - Access links
  - Displays

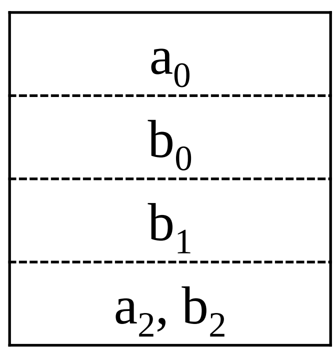
# Block Structure

- Scoping rule - Most closely nested rule
  - the scope of a declaration in a block B includes B
  - If a name x is not declared in a block B, an occurrence of x in B is in the scope of a declaration of x in an enclosing block B’ such that
    - B’ has a declaration of x
    - B’ is more closely nested around B than any other block with a declaration of x

```
      main()
      {
          int a = 0;                            //B0-B2
          int b = 0;                            //B0-B1
          {
              int b = 1;                        //B1-B3
          B2  {
                  int a = 2;                    //B2
B0                printf("%d%d",a,b);
              }
      B1      {
                  int b = 3;                    //B3
          B3      printf("d%d",a,b);
              }
              printf("%d%d",a,b);
          }
          printf("%d%d",a,b);
      }
```

# Implementation of Block Structure

- View a block as a parameterless procedure
  - the space for names can be allocated when the block is entered
  - the space is deallocated when control leaves the block
  - nonlocal environment for a block can be maintained the same as in nested procedures (to be explained later)
- Views a block as a part of the enclosing procedure
  - the space for block is allocated together with the space of the enclosing procedure



# Lexical Scope Without Nested Procedures

- Storage for all names declared outside any procedures can be allocated statically (e.g. C language)
- Important benefit of static allocation for nonlocals
  - procedure as a parameter with no substantial change in data-access strategy
  - any nonlocal to one procedure is nonlocal to all procedures

```
program pass (input, output);
   var m: integer;
   function f(n: integer):integer;
      begin f:= m + n end;
   function g(n: integer): integer;
      begin g: m * n end;
   procedure b(function h(n:integer):integer);
      begin write(h(2)) end;
   begin
            m:=0;
            b(f); b(g);
   end
```

# Lexical Scope with Nested Procedures

- Issues with Nested Procedures
  - in languages using static scoping rule, a procedure can access variables that are declared in its enclosing procedures
  - example

```
procedure q
    x: integer
    ...
       procedure p
        ...x...
```

    - finding the declaration that applies to a non-local name x in p is a static decision
    - finding the relevant activation record of q from an activation of p is a dynamic decision
    - needs some run-time information (e.g. access links)

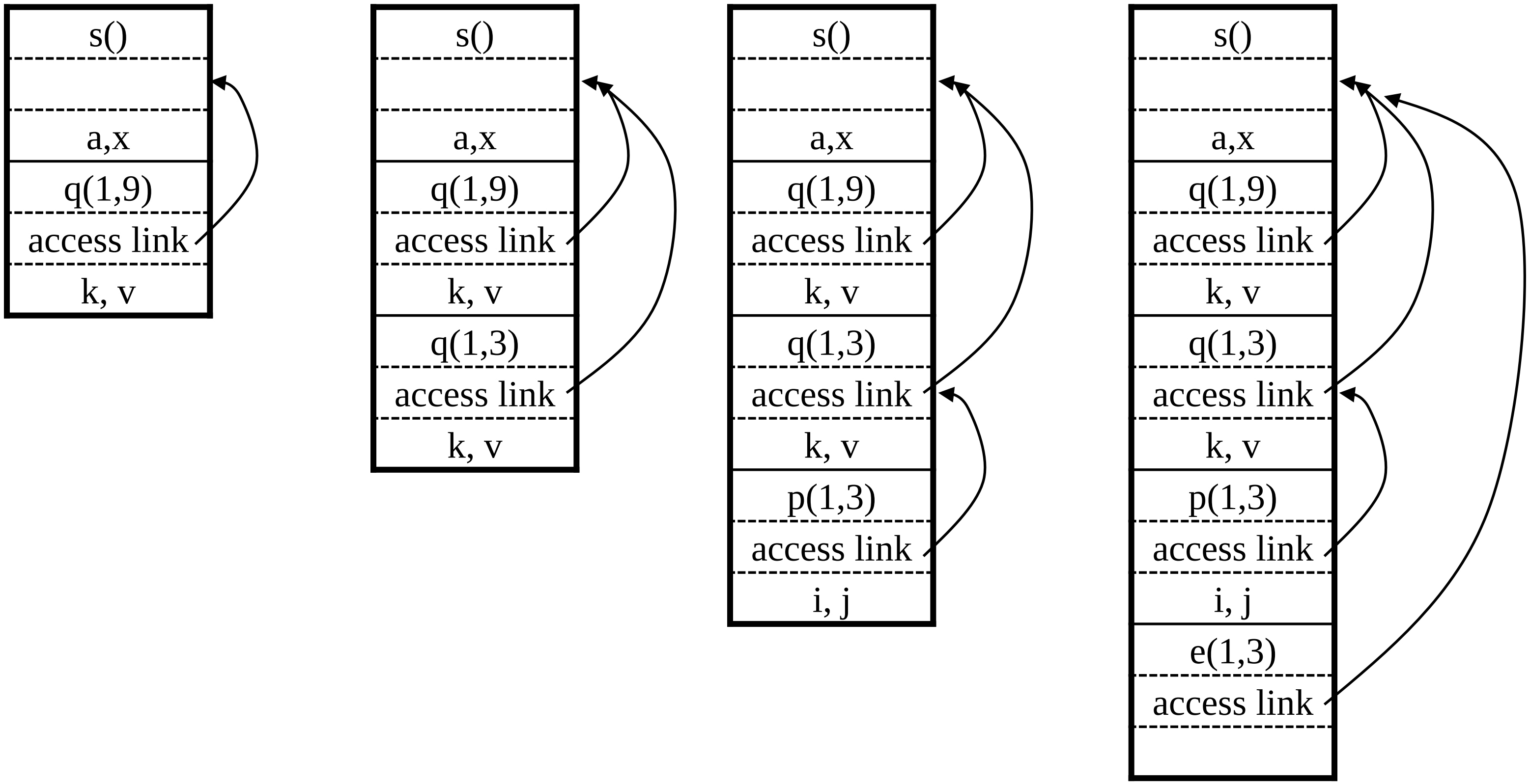
# Access Links

- Access links
  - Links to find the closest enclosing procedures
  - If procedure p is nested immediately within q, the access link in an activation record for p points to the activation record for the most recent activation of q

- Example : sort program

```
program sort(input,output);
  var a : array [0..10] of integer;
      x : integer;
  procedure readarray;
    var i : integer;
    begin … a…. end;
  procedure exchange(i, j: integer);
    begin  x := a[i]; a[i] := a[j];  a[j] := x; end;
  procedure quicksort(m, n: integer)
    var k, v : integer;
    function partition(y, z : integer): integer;
      var i, j : integer;
      begin … a…
             … v…
             … exchange(i, j);
      end
    begin …a…v…partition…quicksort… end
begin … end // sort
```

# Access Links



# Access Links

- *Nesting depth(ND)* of a procedure
  - *ND* of the main program : 1
  - *ND* of an enclosed procedure : *ND* of enclosing proc. + 1
  - a name *x* is associated with ND(*Nx*) of the declaring procedure

*in partition*

| | |
|---|---|
| begin … a… | *Na* : 1 |
| … v… | *Nv* : 2 |
| … exchange(i, j); | *Ni* : 3 |
| end | |

# Access Links

- The address of nonlocal *a* in procedure *p*
  - (*Np* – *Na*, offset within activation record)
- How to access nonlocal variable *a*
  - Follow *Np* – *Na* access links from the top activation record
  - Access the variable at the fixed offset
- Examples
  - v in partition : Np = 3, Nv = 2 → one access link reaches quicksort
  - a in partition : Np = 3, Nv = 1 → two access links reaches sort

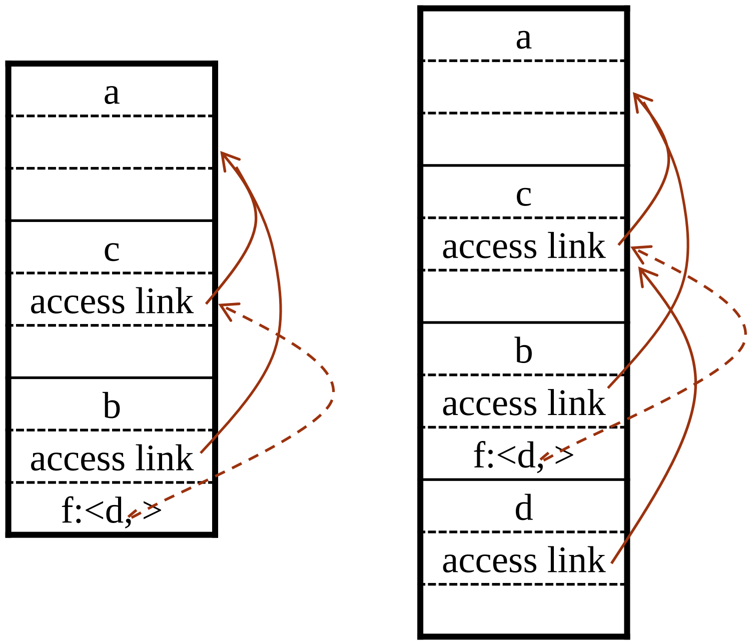
# Access Links

- How to set up access links
  - suppose procedure $p$ calls procedure $q$
  1. Case $Np < Nq$: $q$ is declared within $p$ ($Nq = Np + 1$)
     - Set the access link for $q$ to point to access link for $p$
  2. Case $Np \geq Nq$:
     - A procedure(let's call $r$) with nesting depth $Nq - 1$ encloses both the calling and called procedure most closely
     - Follow $Np$-($Nq$-1) access links from caller to reach $r$
     - Set the access link for q to point to the access link for $r$
- Examples
  - quicksort calls partition : case 1, thus partition -> quicksort
  - partition calls exchange : case 2, Np-Ne+1=2, thus exchage -> sort

# Access Links

- When a procedure *p* is passed to another procedure *q* as a parameter, and *q* calls its parameter (i.e. *p*)
- How does *q* set up the access link for *p*
  - the caller *r* (if *r* calls *q*) can always determine the access link for *p*, as if *r* calls *p* directly
  - therefore the caller *r* set up the parameter *p* with its access link on the stack

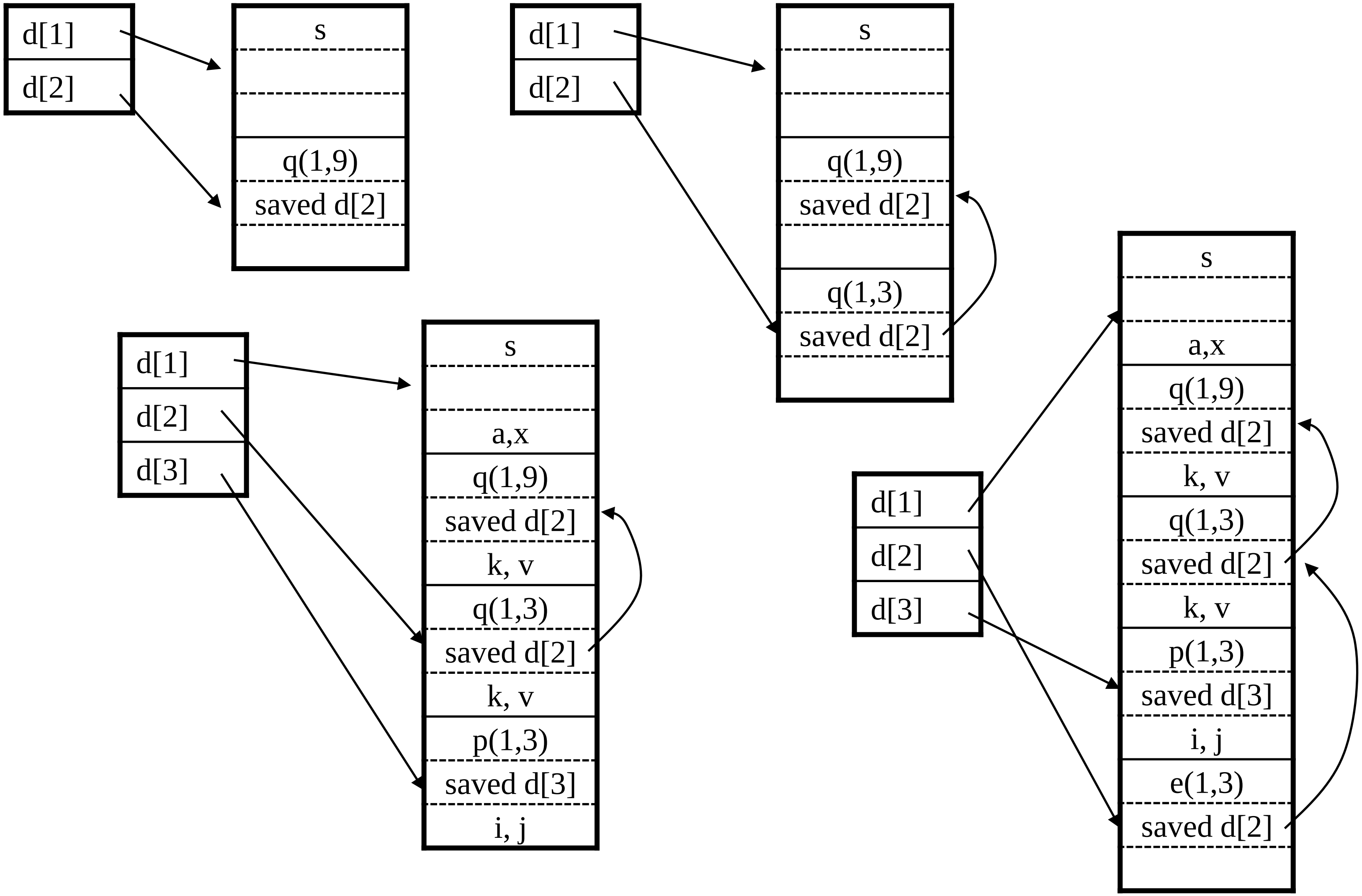
```
fun a(x) =
   let
      fun b(f) =
          ...f...
      fun c(y) =
          let
              fun d(z) = ...
          in
               ...b(d)...
          end
   in
        ...c(1)
   end;
```



# Displays

- Displays
  - an array of pointers to activation records
  - faster method to access nonlocals than with access links
  - a nonlocal *a* at nesting depth *i* is in the activation record pointed to by display element *d*[*i*]
- Maintaining the display
  - when a new activation record for a procedure at *ND i* is set up
    - save the value of *d*[*i*] in the new activation record
    - set *d*[*i*] to point to the new activation record
  - when control returns from the activation record, *d*[*i*] is reset

# Displays



d[1]
d[2]
s
q(1,9)
saved d[2]
d[1]
d[2]
s
q(1,9)
saved d[2]
q(1,3)
saved d[2]
d[1]
d[2]
d[3]
s
a,x
q(1,9)
saved d[2]
k, v
q(1,3)
saved d[2]
k, v
p(1,3)
saved d[3]
i, j
s
a,x
q(1,9)
saved d[2]
k, v
q(1,3)
saved d[2]
k, v
p(1,3)
saved d[3]
i, j
e(1,3)
saved d[2]
d[1]
d[2]
d[3]