Run Time Environments

ALSU Textbook Chapter 7.1–7.3

Tsan-sheng Hsu

tshsu@iis.sinica.edu.tw

`http://www.iis.sinica.edu.tw/~tshsu`

Preliminaries

- **During the execution of a program, the same name in the source can denote different data objects.**
- **The allocation and deallocation of data objects is managed by the run-time support package.**
- **Terminologies:**
  - **environment: the mapping of names to storage spaces.**
    name $\rightarrow$ storage space
  - **state: the current value of a storage space.**
    storage space $\rightarrow$ value
  - **binding: the association of a name to a storage location.**
- **Each execution of a procedure is called an activation.**
  - **Several activations of a recursive procedure may exist at the same time.**
    - ▷ ***A recursive procedure needs not to call itself directly.***
  - **Life time: the time between the first and last steps in a procedure.**

Activation record

returned value
actual parameters
optional control link
optional access link
saved machine status
local data
temporaries

- **Activation record (A.R.): data about an execution of a procedure.**

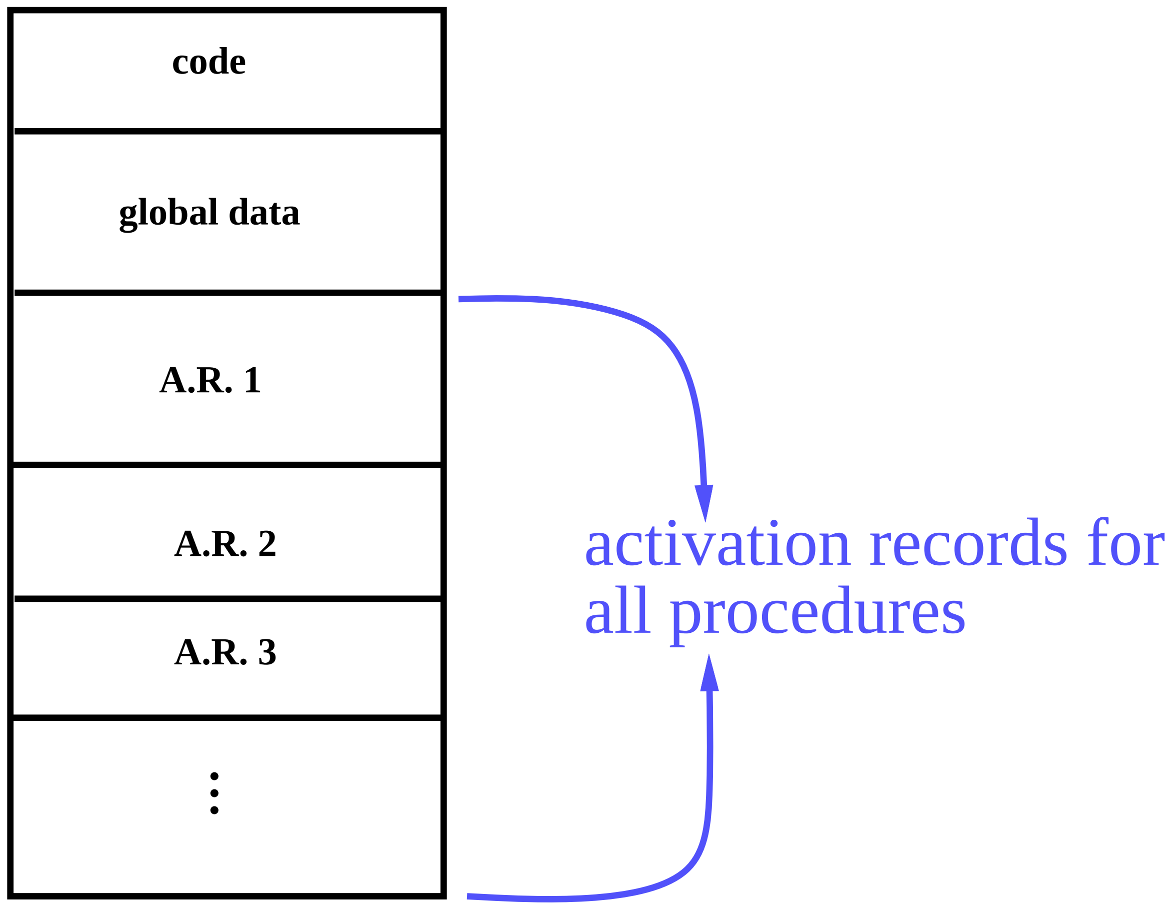
Contents of A.R.

- **Returned value for a function.**
- **Parameters:**
  - **Formal parameters:** the declaration of parameters.
  - **Actual parameters:** the values of parameters for this activation.
- **Links: where variables can be found.**
  - **Control (or dynamic) link:** a pointer to the activation record of the caller.
  - **Access (or static) link:** a pointer to places of non-local data,
- **Saved machine status.**
- **Local variables.**
- **Temporary variables.**
  - Evaluation of expressions.
  - Evaluation of arguments.
  - Evaluation of array indexes.
  - · · ·

Issues in storage allocation

- There are two different approaches for run time storage allocation.
  - Static allocation.
    - *Allocate all needed space when program starts.*
    - *Deallocate all space when program terminates.*
  - Dynamic allocation.
    - *Allocate space when it is needed.*
    - *Deallocate space when it is no longer needed.*
- Need to worry about how variables are stored.
  - That is the management of activation records.
- Need to worry about how variables are accessed.
  - Global variables.
  - Locally declared variables, that is the ones allocated within the current activation record.
  - Non-local variables, that is the ones declared and allocated in other activation records and still can be accessed.
    - *Non-local variables are different from global variables.*

Static storage allocation



Static storage allocation (1/3)

- **Static allocation:** **uses no stack and heap.**
  - **Strategies:**
    - ▷ *For each procedure in the program, allocate a space for its activation record.*
    - ▷ *A.R.'s can be allocated in the static data area.*
    - ▷ *Names bound to locations at compiler time.*
    - ▷ *Every time a procedure is called, a name always refer to the same pre-assigned location.*
  - **Used by simple or early programming languages.**
- **Disadvantages:**
  - **No recursion.**
  - **Waste lots of space when procedures are inactive.**
  - **No dynamic allocation.**
- **Advantages:**
  - **No stack manipulation or indirect access to names, i.e., faster in accessing variables.**
  - **Values are retained from one procedure call to the next if block structure is not allowed.**
    - ▷ *For example: static variables in C.*

Static storage allocation (2/3)

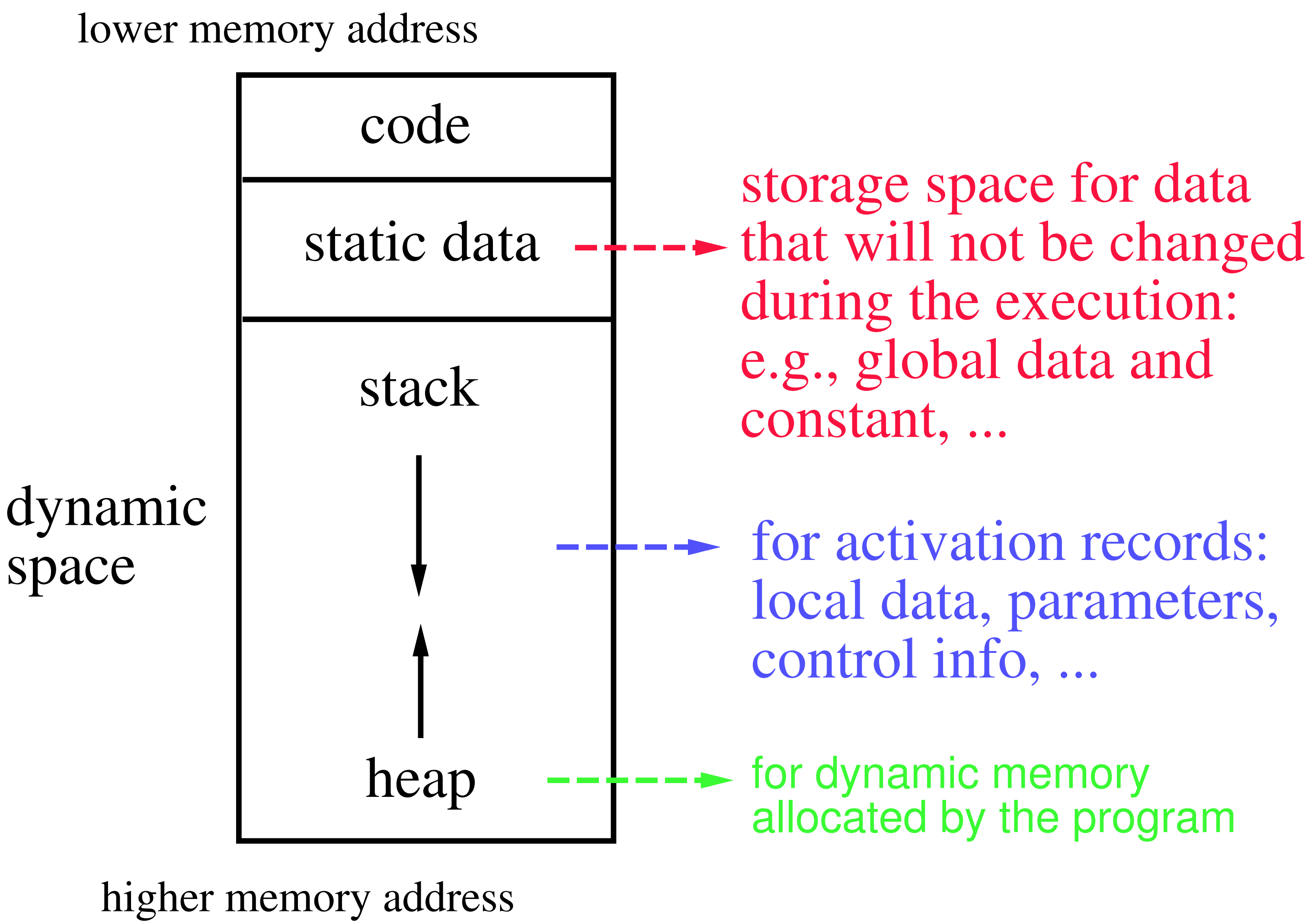
- **On procedure calls,**
  - **the calling procedure:**
    - ▷ *First evaluate arguments.*
    - ▷ *Copy arguments into parameter space in the A.R. of called procedure.*

      Conventions: call that which are passed to a procedure arguments from the calling side, and parameters from the called side.
    - ▷ *May need to save some registers in its own A.R.*
    - ▷ *Jump and link: jump to the first instruction of called procedure and put address of next instruction (return address) into register RA (the return address register).*
  - **the called procedure:**
    - ▷ *Copy return address from RA into its A.R.'s return address field.*
    - ▷ *control link := address of the previous A.R.*
    - ▷ *May need to save some registers.*
    - ▷ *May need to initialize local data.*

Static storage allocation (3/3)

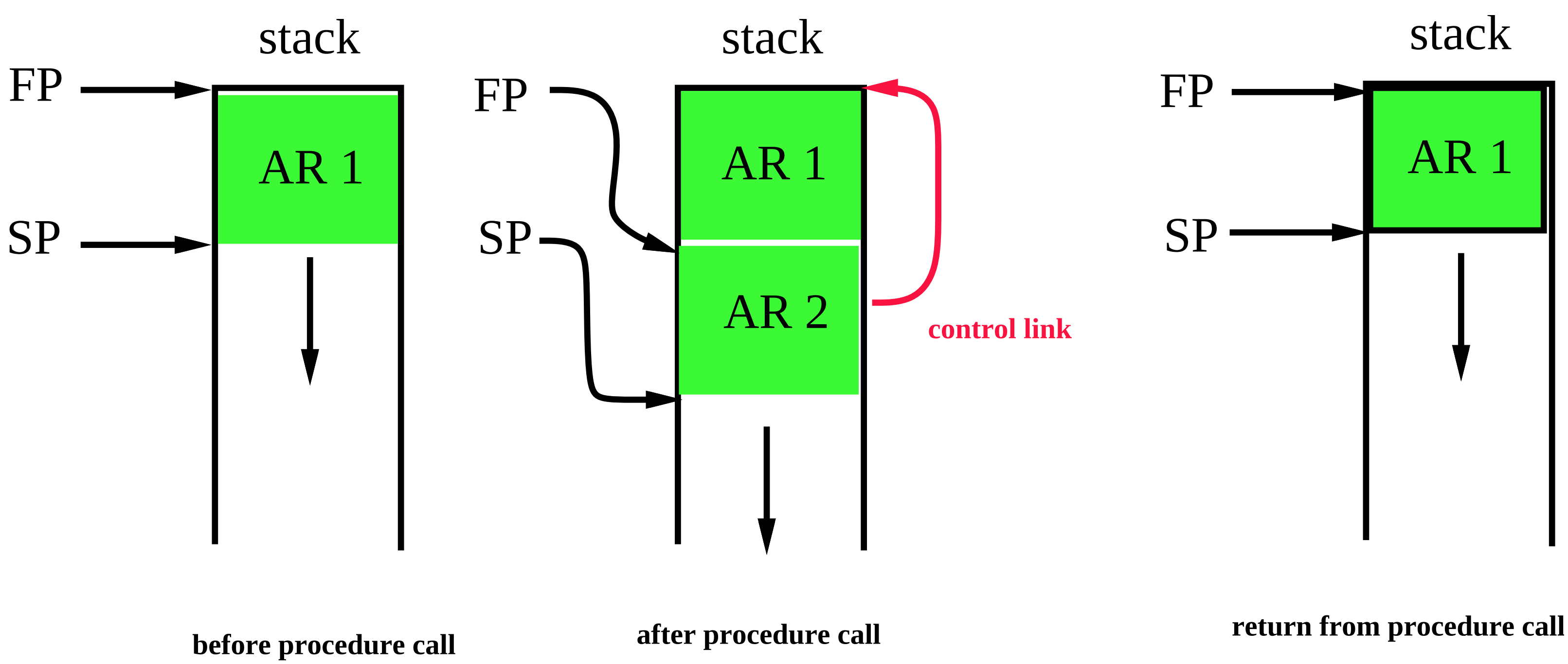
- **On procedure returns,**
  - **the called procedure:**
    - ▷ ***Restore values of saved registers.***
    - ▷ ***Jump to address in the return address field.***
  - **the calling procedure:**
    - ▷ ***May need to restore some registers.***
    - ▷ ***If the called procedure is actually a function, that is the one that returns values, put the return value in the appropriate place.***

Dynamic storage allocation



Dynamic storage allocation for stack (1/3)

- **Stack allocation:**
  - **Each time a procedure is called, a new A.R. is pushed onto the stack.**
  - **A.R. is popped when procedure returns.**
  - **A register (stack pointer or SP) points to top of stack.**
  - **A register (frame pointer or FP) points to start of current A.R.**



Dynamic storage allocation for stack (2/3)

- **On procedure calls,**
  - **the calling procedure:**
    - ▷ ***May need to save some registers in its own A.R..***
    - ▷ ***May need to set an optional access link.***
    - ▷ ***Push parameters onto stack.***
    - ▷ ***Jump and Link: jump to the first instruction of called procedure and put address of next instruction into register RA.***
  - **the called procedure:**
    - ▷ ***Save return address in RA.***
    - ▷ ***Save old FP (in the control link space).***
    - ▷ ***Set new FP (FP := SP).***
    - ▷ ***Set new SP (SP := SP +(size of parameters) + (size of RA) + (size of FP). (These sizes can be computed at compile time.)***
    - ▷ ***May need to save some registers.***
    - ▷ ***Push local data (produce actual data if initialized or just allocate spaces if not)***

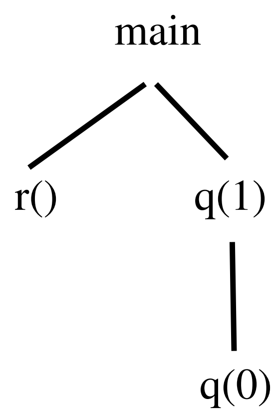
Dynamic storage allocation for stack (3/3)

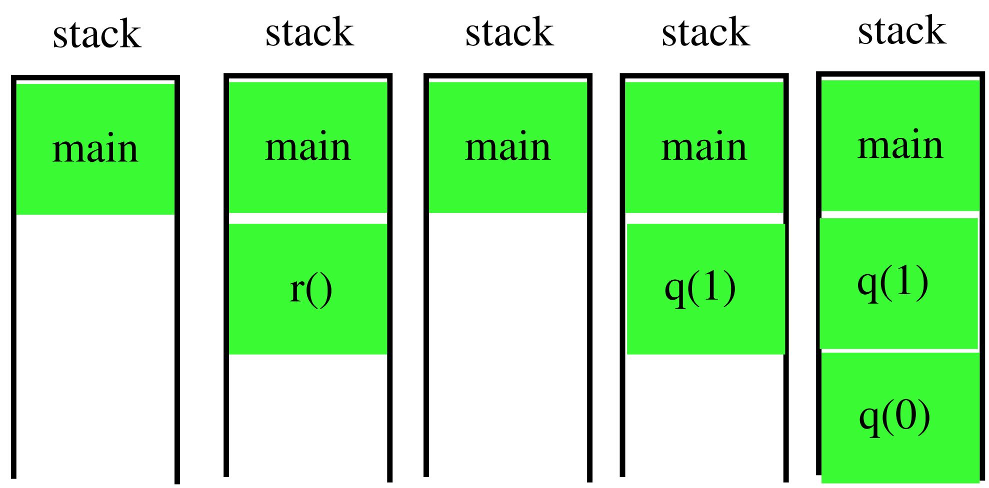
- **On procedure returns,**
  - **the called procedure:**
    - *Restore values of saved registers if needed.*
    - *Load return address into special register RA.*
    - *Restore SP (SP := FP).*
    - *Restore FP (FP := control link).*
    - *Return.*
  - **the calling procedure:**
    - *May need to restore some registers.*
    - *If a function that was called, put the return value into the appropriate place.*

Activation tree

- **Use a tree structure to record the changing of the activation records.**
- **Example:**

```
main{
    r();
    q(1);
}

r{
...
}

q(int i)
{
if(i>0) then q(i-1);
}
```





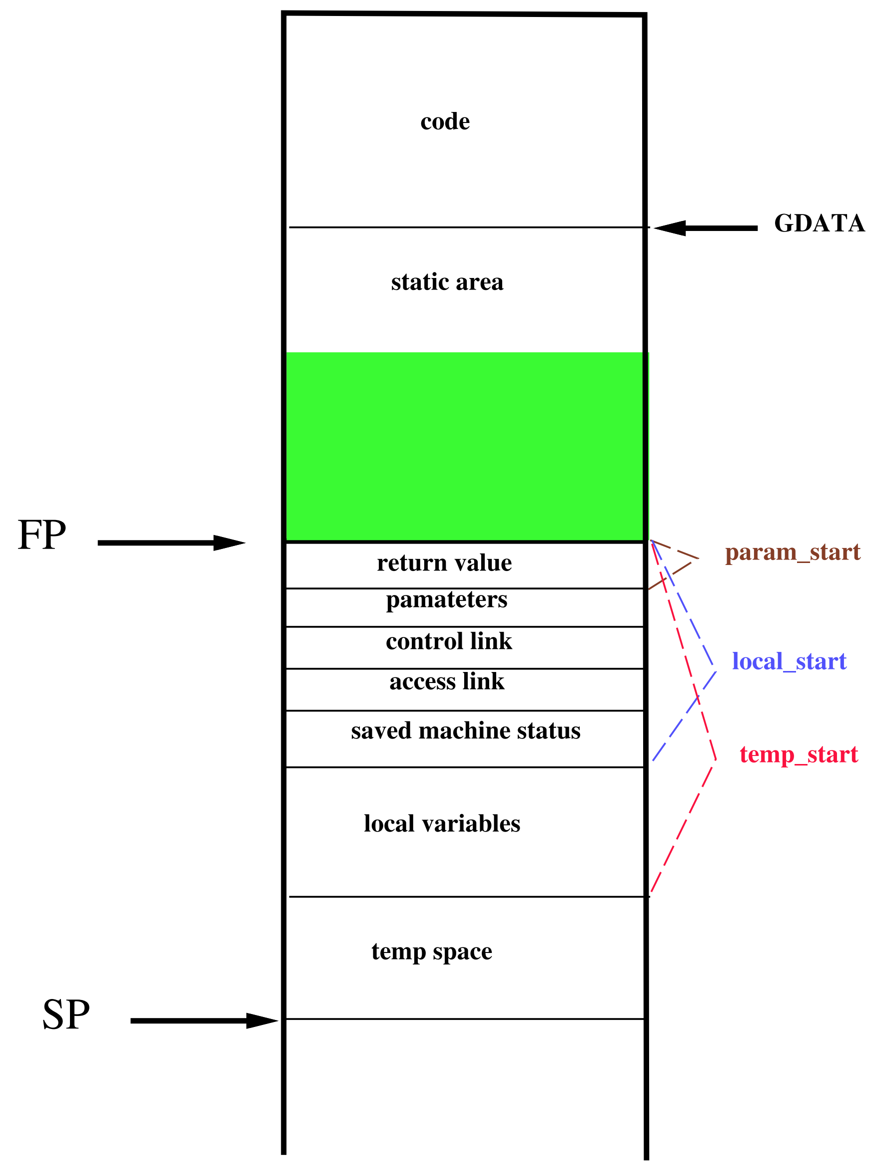
Dynamic storage allocation for heap

- **Storages requested from programmers during execution:**
  - **Example:**
    - ***PASCAL:*** new ***and*** $free$***.***
    - ***C:*** $malloc$ ***and*** $free$***.***
  - **Issues:**
    - ***Garbage collection.***
    - ***Dangling reference.***
    - ***Segmentation and fragmentation.***
- **More or less O.S. issues.**

Accessing global variables

- **Global variables:**
  - **Access by using names.**
  - **Addresses known at compile time.**
  - **Access a global variable u by $offset(u)$ from the global variable area.**
    - ***Let $GDATA$ be the starting address of the global data area.***
    - ***The value $offset(u)$ is the amount of spaces allocated to global variables declared before u.***
    - ***The address of u is $GDATA + offset(u)$.***
    - ***The actual address is only known at run time, depending on the value of $GDATA$.***

Example for memory management

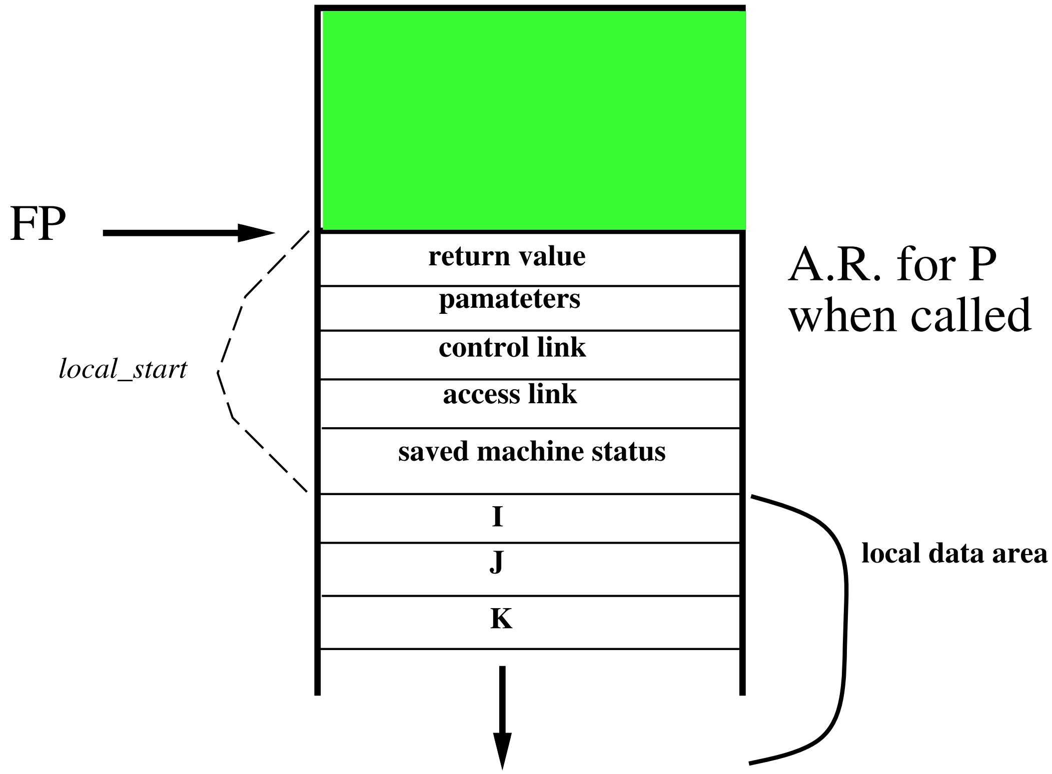


Accessing local variables

- **Local variables:**
  - **Stored in the activation record of declaring procedure.**
  - **Access a local variable v in a procedure P by $offset(v)$ from the frame pointer (FP).**
    - ▷ ***Let $local\_start(P)$ be the amount of spaces used by data in the activation record of procedure P that are allocated before the local data area.***
    - ▷ ***The value $local\_start(P)$ can be computed at compile time.***
    - ▷ ***The value $offset(v)$ is the amount of spaces allocated to local variables declared before v.***
    - ▷ ***The address of v is FP $+ local\_start(P) + offset(v)$.***
    - ▷ ***The actual address is only known at run time, depending on the value of FP.***

Accessing local variables – example

```
int P()
{
int I,J,K;
...
}
```

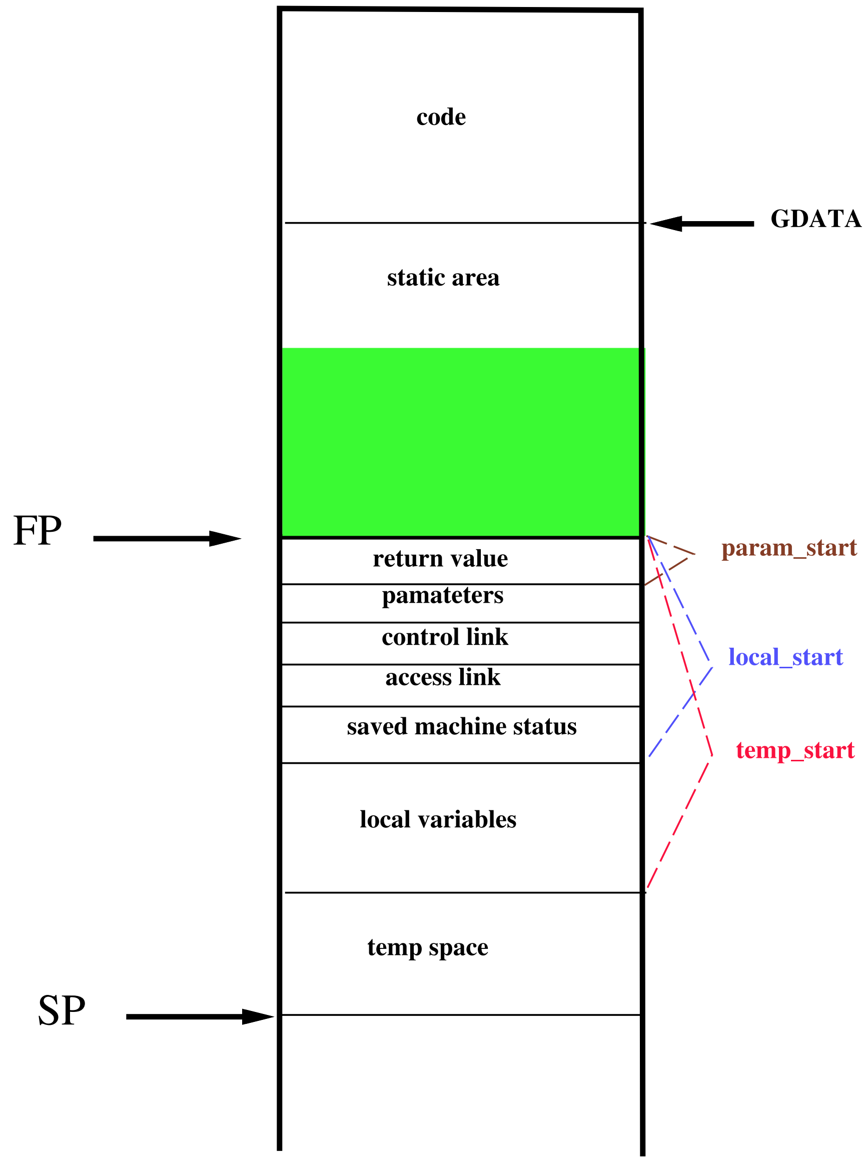


- **Address of J is FP $+local\_start(P) + offset(J)$.**
  - ▷ $offset(J)$ ***is*** $1 * sizeof(int)$ ***and is known at compile time.***
  - ▷ $local\_start(P)$ ***is known at compile time.***
  - ▷ ***Actual address is only known at run time, i.e., depends on the value of FP.***

Code generation routine

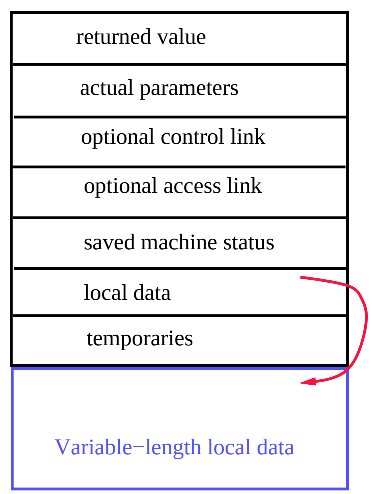
- **Code generation:**
  - **gen([address #1], [assignment], [address #2], operator, address #3);**
    - *Use switch statement to actually print out the target code;*
    - *Can have different gen() for different target codes;*
- **Variable accessing: depend on type of [address #i], generate different codes.**
  - **Watch out the differences between l-address and r-address.**
  - **Parameter: FP+param_start+offset.**
  - **Local variable: FP+local_start+offset.**
  - **Local temp space: FP+temp_start+offset.**
  - **Global variable: GDATA+offset.**
  - **Registers, constants, $\ldots$**
  - **Non-local variable: to be discussed if nested function/procedure declaration is allowed.**
  - **Special cares needed for arrays: need to add $base$ and compute the proper offset given an index.**

Example for memory management



Variable-length local data

- **Allocation of space for objects the sizes of which are not known at compile time.**
  - **Example: Arrays whose size depends on the value of one or more parameters of the called procedure.**
  - **Cannot calculate proper offsets if they are allocated on the A.R.**
- **Strategy:**
  - **Allocate these objects at the bottom of A.R.**
    - ▷ *Automatically de-allocated when the procedure is returned.*
  - **Keep a pointer to such an object inside the local data area.**
  - **Need to de-reference this pointer whenever it it used.**



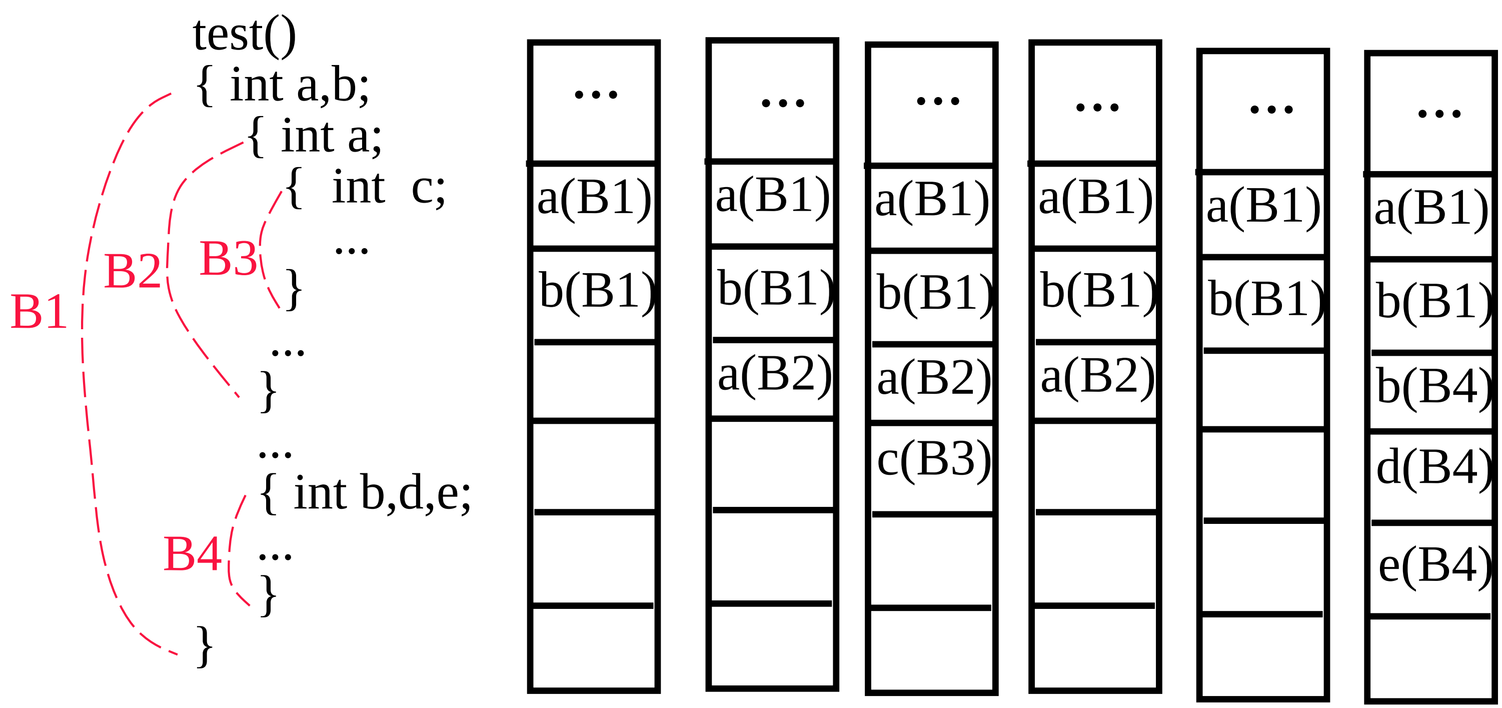
Accessing non-local variables

- **Two scoping rules for accessing non-local data.**
  - **Lexical or static scoping.**
    - ▷ *PASCAL, C and FORTRAN.*
    - ▷ *The correct address of a non-local name can be determined at compile time by checking the syntax.*
    - ▷ *Can be with or without block structures.*
    - ▷ *Can be with or without nested procedures.*
  - **Dynamic scoping.**
    - ▷ *LISP.*
    - ▷ *A use of a non-local variable corresponds to the declaration in the "most recently called, still active" procedure.*
    - ▷ *The question of which non-local variable to use cannot be determined at compile time. It can only be determined at run-time.*

Lexical scoping with block structures (1/2)

- **Block**: a statement containing its own local data declaration.
- Scoping is given by the following so called **most closely nested rule.**
  - The scope of a declaration in a block B includes B itself.
  - If x is used in B, but not declared in B, then we refer to x in a block B', where
    - *B' has a declaration x, and*
    - *B' is more closely nested around B than any other block with a declaration of x.*
- If a language does not allow nested procedures, then
  - a variable is either global, or is local to the procedure containing it;
  - at runtime, all the variables declared (including those in blocks) in a procedure are stored in its A.R., with possible overlapping;
  - during compiling, proper offset for each local data is calculated using information known from the block structure.

Lexical scoping with block structures (2/2)



- **Maintain the current offset in a procedure.**
- **Maintain the amount of spaces used in each block.**
  - **Initialize to 0 when a block is opened.**
  - **Substrate the total amount of spaces used in the block from the current offset when this block is closed.**

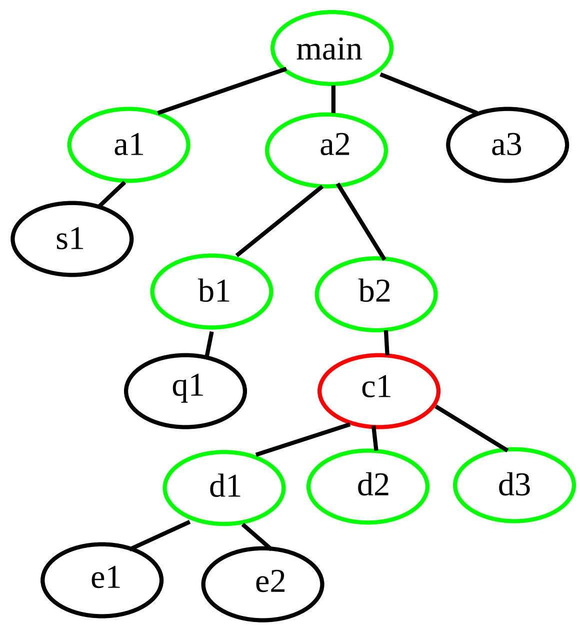
Lexical scoping with nested procedures

- **Nested procedure : a procedure that can be declared within another procedure.**
- **Issues:**
  - **What are the procedures that can be called at a given location?**
  - **What are the variables that can be accesses at a given location during compiler time?**
  - **How to access these variable during run time?**

Calling procedures

- **A procedure Q_i can call any procedure that is its child, older siblings, direct ancestors or the older siblings of its direct ancestor.**
  - ▷ ***The procedure Q_{i+1} that is declared in Q_i.***
  - ▷ ***The procedure Q_{i-1} who declares Q_i.***
  - ▷ ***The procedure Q_{i-j} who declares Q_{i-j+1}, $j > 1$.***
  - ▷ ***The procedure P_j whom is declared together with, and before, Q_j, $j \leq i$.***
- **Use the symbol table to find the procedures that can be called.**

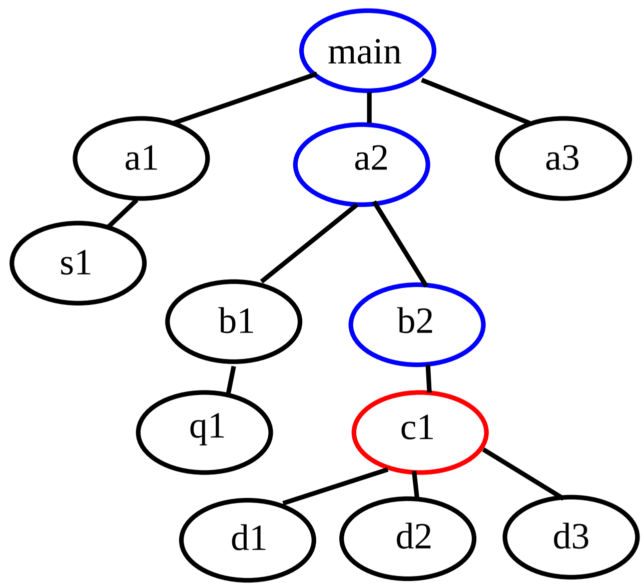
```
procedure main
  procedure a1
      procedure s1
  procedure a2
      procedure b1
          procedure q1
      procedure b2
          procedure c1
            procedure d1
                procedure e1
                procedure e2
            procedure d2
            procedure d3
           ***
   procedure a3
```



Accessing variables

- **A procedure can only access the variables that are either local to itself or global in a procedure that is its direct ancestor.**
  - ▷ ***When you call a procedure, a variable name follows the lexical scoping rule.***
  - ▷ ***Use the access link to link to the procedure that is lexically enclosing the called procedure.***
  - ▷ ***Need to set up the access link properly to access the right storage space.***

```
procedure main
  procedure a1
      procedure s1
  procedure a2
      procedure b1
          procedure q1
      procedure b2
          procedure c1
             procedure d1
             procedure d2
             procedure d3
            ***
   procedure a3
```

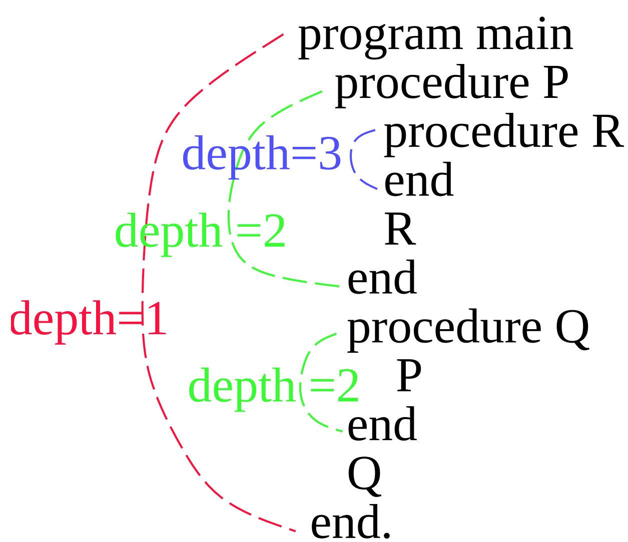


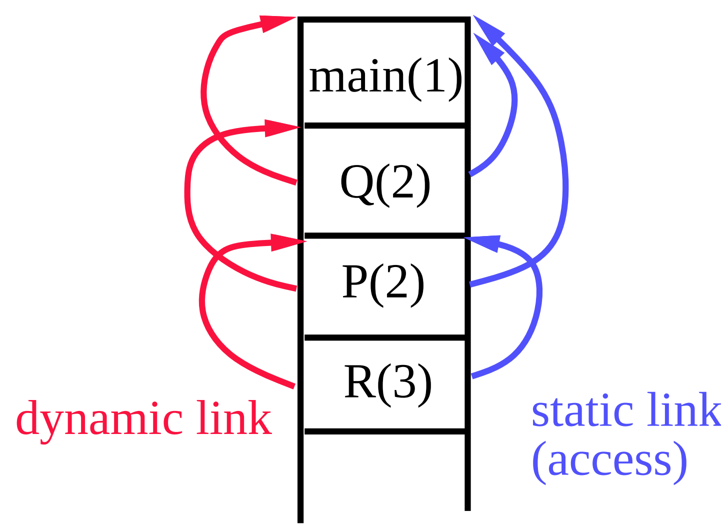
Pointers needed during procedure calls

- **According to the syntax, which is independent of procedure calls during the run time, the A.R.'s of the procedures that are my direct ancestors.**
  - **Access link or static link.**
- **According to the sequence of procedure calls during the run time, the A.R. of the procedure who calls me.**
  - **Control link or dynamic link.**

Accessing non-local variables

- **Nesting depth** :
  - Depth of main program = 1.
  - Add 1 to depth each time entering a nested procedure.
  - Substrate 1 from depth each time existing from a nested procedure.
  - Each variable is associated with a nesting depth.
  - Assume in a depth-h procedure, we access a variable at depth k, then
    - ▷ $h \geq k$.
    - ▷ *Follow the access (static) link $h - k$ times, and then use the offset information to find the address.*



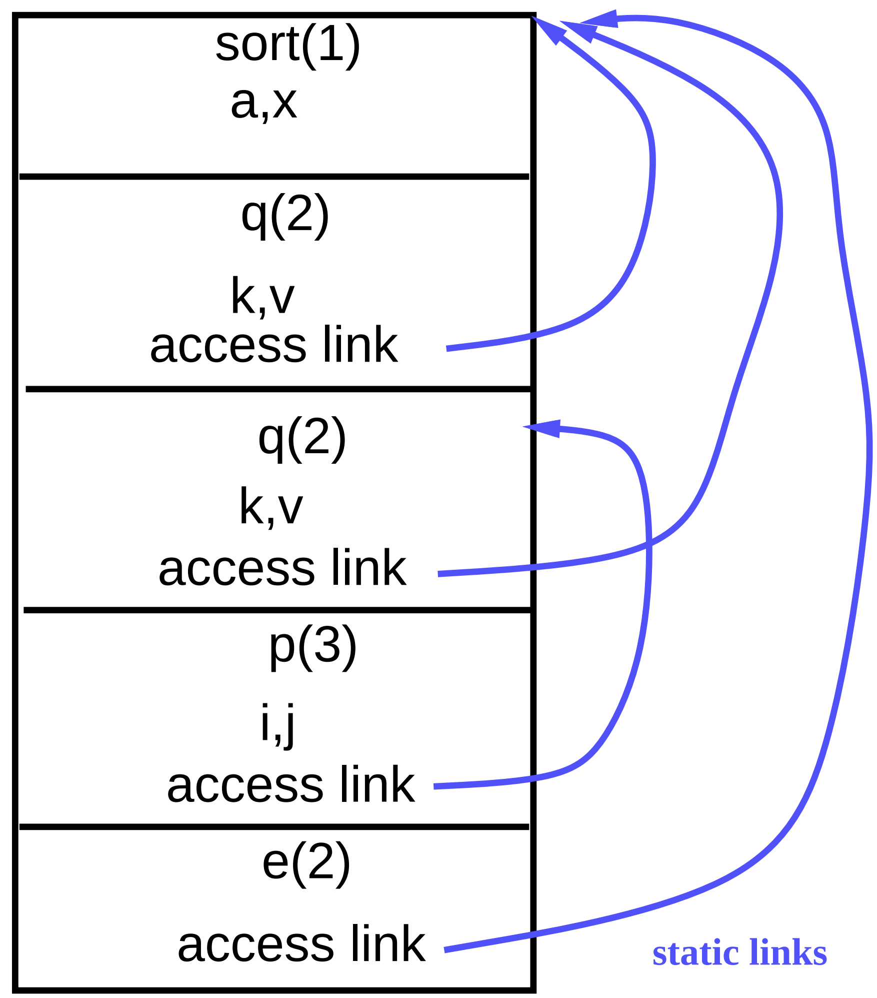


Algorithm for setting the links

- **The control link is set to point to the A.R. of the calling procedure.**
- **How to properly set the access link at compile time?**
  - **Procedure P at depth n_P calls procedure X at depth n_X:**
  - **If $n_P < n_X$, then X is enclosed in P and $n_P = n_X - 1$.**
    - ▷ ***Same with setting the control (dynamic) link.***
  - **If $n_P \geq n_X$, then it is either a recursive call or calling a previously declared procedure.**
    - ▷ ***Observation: go up the access (static) link once, then the depth is decreased by 1.***
    - ▷ ***Hence, the access (static) link of X is the access link of P going up $n_P - n_X + 1$ times.***
  - **Content of the access (static) link in the A.R. for procedure P:**
    - ▷ ***Points to the A.R. of the procedure Q who encloses P lexically.***
    - ▷ ***An A.R. of Q must be active at this time.***
    - ▷ ***Several A.R.'s of Q (recursive calls) may exist at the same time, it points to the latest activated one.***

Access (static) links – example

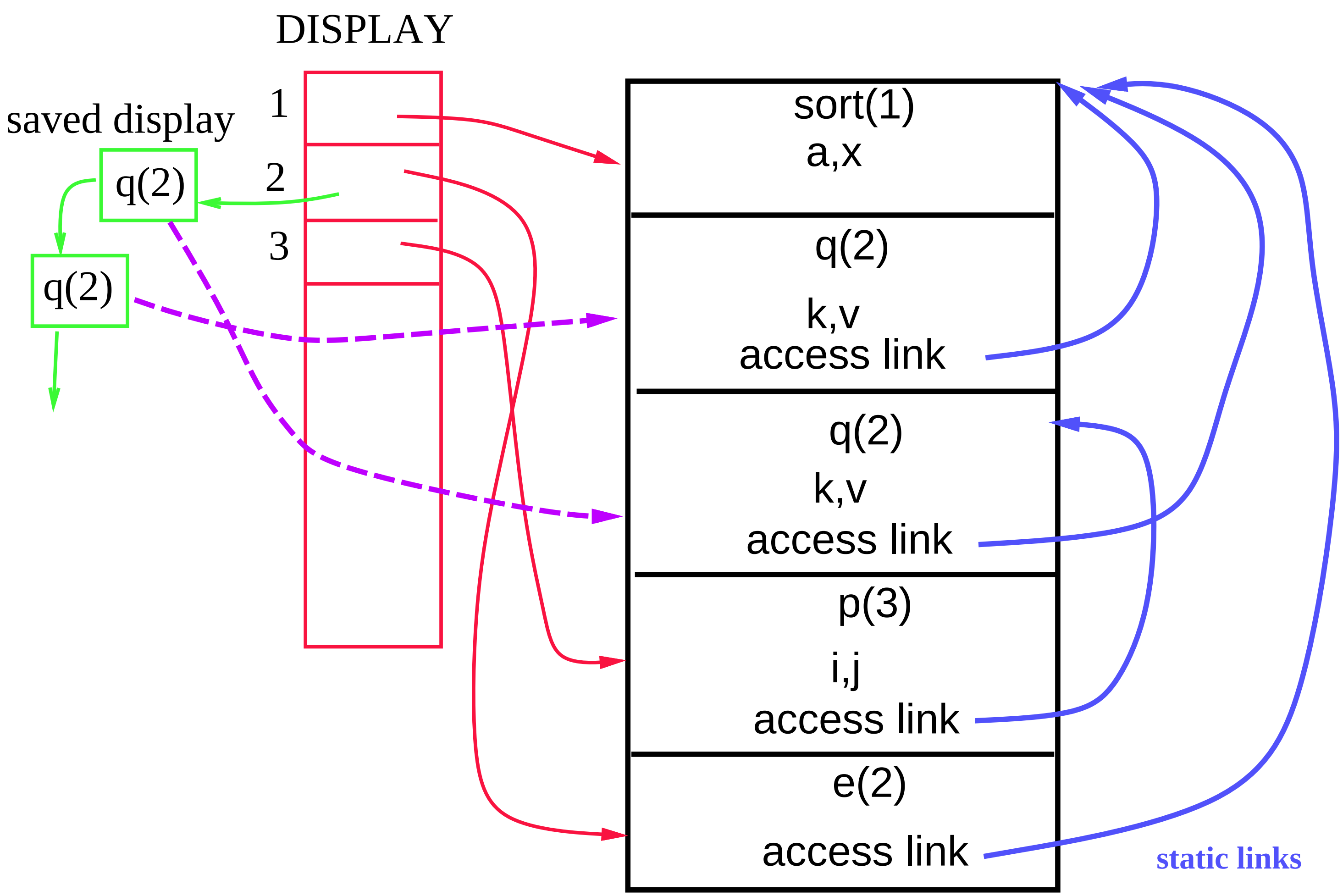
```
Program sort
   var a: array[0..10] of int;
       x: int;
   procedure r
   var i: int;
   begin ...  r
   end

   procedure e(i,j)
   begin ... e
      a[i] <-> a[j]
   end

   procedure q
      var k,v: int;
      procedure p
      var i,j;
      begin ... p
        call e
      end
   begin ... q
        call q or p
   end

begin ... sort
   call q
end
```



Accessing non-local data using DISPLAY

- **Idea:**
  - **Maintain a global array called DISPLAY.**
    - ▷ ***Using registers if available.***
    - ▷ ***Otherwise, stored in the static data area.***
  - **When procedure P at nesting depth k is called,**
    - ▷ ***DISPLAY[1], . . ., DISPLAY[k-1] hold pointers to the A.R.'s of the most recent activation of the $k-1$ procedures that lexically enclose P.***
    - ▷ ***DISPLAY[k] holds pointer to P's A.R.***
    - ▷ ***To access a variable with declaration at depth x, use DISPLAY[x] to get to the A.R. that holds x, then use the usual offset to get x itself.***
    - ▷ ***Size of DISPLAY equals maximum nesting depth of procedures.***
  - **Bad for languages allow recursions.**
- **To maintain the DISPLAY:**
  - **When a procedure at nesting depth k is called**
    - ▷ ***Save the current value of DISPLAY[k] in the save-display area of the new A.R.***
    - ▷ ***Set DISPLAY[k] to point to the new A.R., i.e., to its save-display area.***
  - **When the procedure returns, restore DISPLAY[k] using the value saved in the save-display area.**

DISPLAY: example



Access (static) links v.s. DISPLAY

- **Time and space trade-off.**
  - **Access (static) links require more time (at run time) to access non-local data, especially when non-local data are many nesting levels away.**
  - **DISPLAY probably require more space (at run time).**
  - **Code generated using DISPLAY is simpler.**

Dynamic scoping

- **Dynamic scoping: a use of a non-local variable refers to the one declared in the "most recently called, still active" procedure.**
- **The question of which non-local variable to use cannot be determined at compile time.**
- **It can only be determined at run time.**
- **May need symbol tables at run time.**
- **Two major methods for implement non-local accessing under dynamic scoping.**
  - **Deep access.**
  - **Shallow access.**

Dynamic scoping – Example

- **Code:**

```
program main
   procedure UsesX
   begin
     write(x);
   end
   procedure DeclaresX
      var x: int;
   begin
      x := 100;
      call UsesX;
   end
   procedure test
   var x : int;
   begin
       x := 30;
       call DeclaresX;
       call UsesX;
   end
begin
   call test;
end
```

- **Which x is it in the procedure UsesX?**
- **If we were to use static scoping, this is not a legal statement; No enclosing scope declares x.**

Deep access

- **Def: given a use of a non-local variable, use control links to search back in the stack for the most recent A.R. that contains space for that variable.**
- **Requirements:**
  - **Be able to locate the set of variables stored in each A.R. at run time.**
  - **Need to use the symbol table at run time.**

Shallow access

- **Idea:**
  - **Maintain a current list of variables.**
  - **Space is allocated (in registers or in the static data area) for every possible variable name that is in the program (i.e., one space for variable x even if there are several declarations of x in different procedures).**
  - **For every reference to x, the generated code refers to the same location.**
- **When a procedure is called,**
  - **it saves, in its own A.R., the current values of all of the variables that it declares (i.e., if it declares x and y, then it saves the values of x and y that are currently in the space for x and y);**
  - **it restores those values when the procedure returns.**

Comparisons of deep and shallow accesses

- **Shallow access allows fast access to non-locals variables, but there is an overhead on procedure entry and exit that is proportional to the number of local variables.**
- **Deep access needs to use a symbol table at run time.**