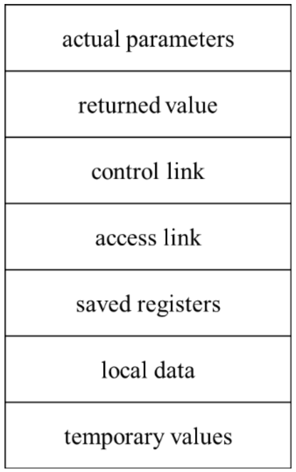
Runtime Environment



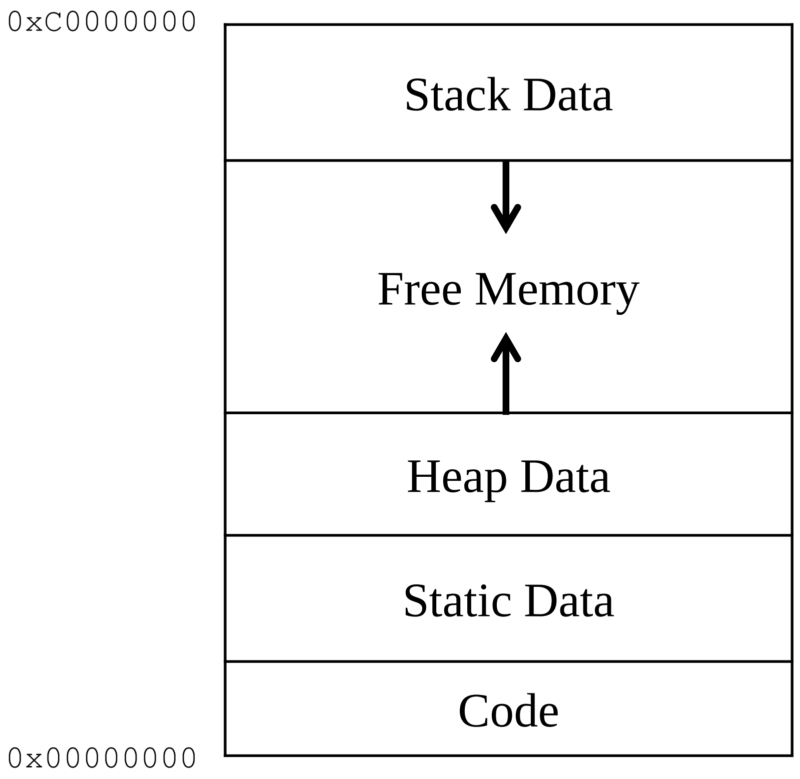
4190.409 Compilers, Spring 2016

Interaction with the Runtime Environment

- Application Binary Interface (ABI)
  - interface between the binary and the operating system (or another binary, such as libraries)
  - coverage
    - data types' size, layout, and alignment rules
    - calling convention
    - system call interface
    - binary format of object files

Memory Organization

- Private, logical address space
- Typical layout on x86 Linux machines



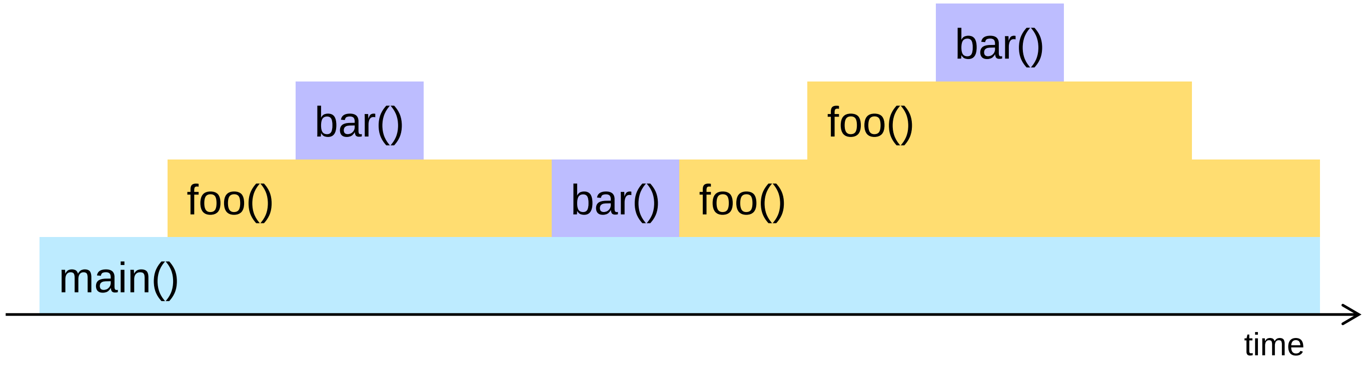
Memory Organization

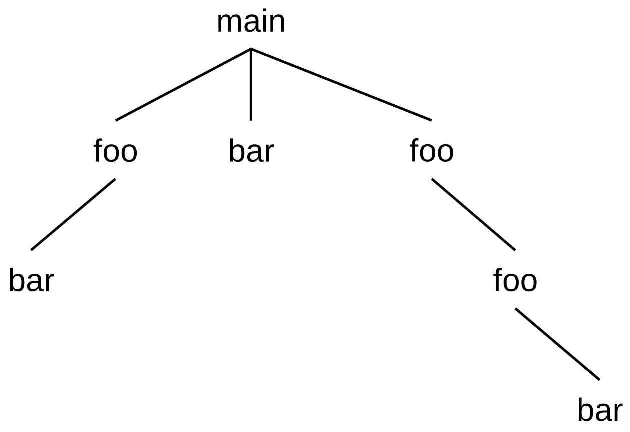
- Code, static data
  - allocated at compile time
  - known addresses

- Dynamic data, stack
  - allocated at run time
  - unknown addresses (and depend on machine/system)

Activation Trees

- Procedure activations are perfectly nested in time





Stack Allocation and Activation Trees

- When a procedure is called, the corresponding data is stored on a run-time stack
  - feasible because procedure activations are nested in time

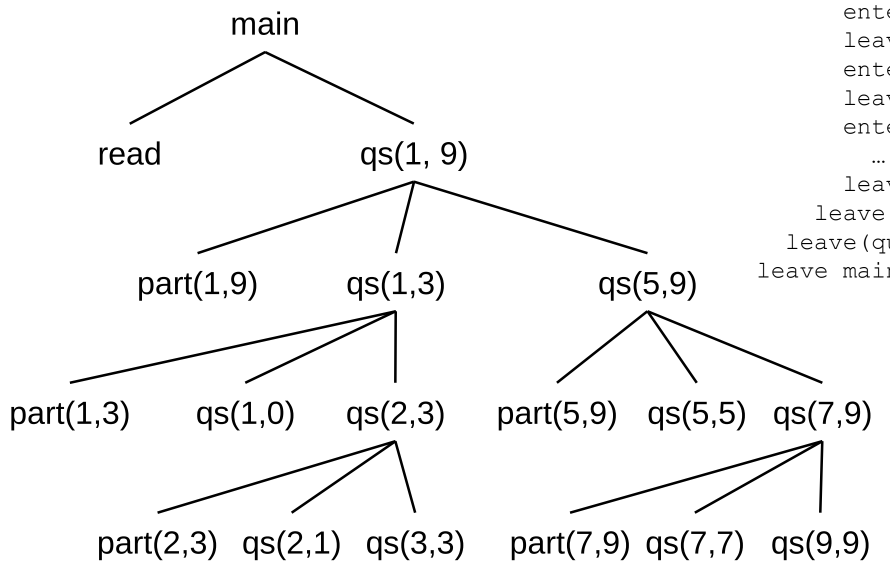
- Quicksort example (see textbook 7.2)

```
int a[11];

void read(void) { int i; … }

int partition(int m, int n) { … }

void quicksort(int m, int n) {
  int i;
  if (n > m) {
    i = partition(m,n);
    quicksort(m, i-1);
    quicksort(i+1, n);
  }
}

void main(void) {
  read();
  quicksort(1, 9);
}
```

Stack Allocation and Activation Trees

- Relationship activation tree and program behavior
  - sequence of calls = preorder traversal
  - sequence of returns = postorder traversal
  - calling sequence = path from node to root
  - currently live activations = nodes on path to root

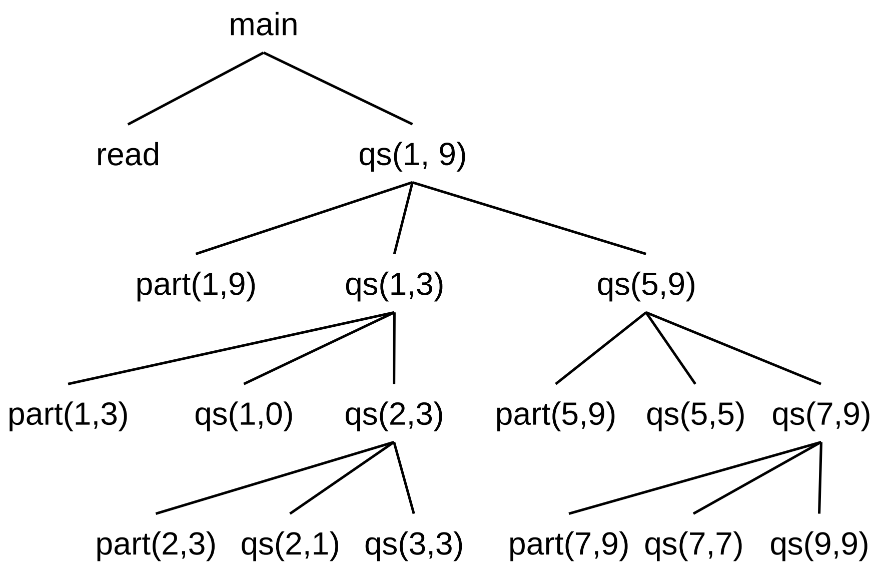
```
enter main
  enter read
  leave read
  enter quicksort(1,9)
    enter partition(1,9)
    leave partition(1,9)
    enter quicksort(1,3)
      enter partition(1,3)
      enter quicksort(1,0)
      leave quicksort(1,0)
      enter quicksort(2,3)
        enter partition(2,3)
        leave partition(2,3)
        enter partition(2,1)
        leave partition(2,1)
        enter partition(3,3)
        leave partition(3,3)
      leave quicksort(2,3)
    leave quicksort(1,3)
    enter quicksort(5,9)
      enter partition(5,9)
      leave partition(5,9)
      enter quicksort(5,5)
      leave quicksort(5,5)
      enter quicksort(7,9)
        …
      leave quicksort(7,9)
    leave quicksort(5,9)
  leave(quicksort(1,9)
leave main
```

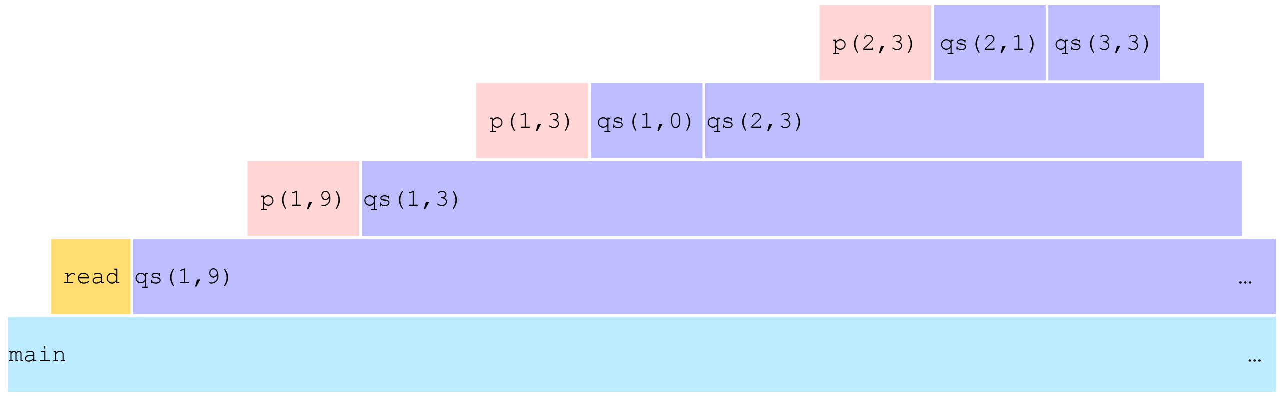


Stack Allocation and Activation Trees

- Quicksort example (cont'd): Possible activations and activation tree

```
enter main
  enter read
  leave read
  enter quicksort(1,9)
    enter partition(1,9)
    leave partition(1,9)
    enter quicksort(1,3)
      enter partition(1,3)
      enter quicksort(1,0)
      leave quicksort(1,0)
      enter quicksort(2,3)
        enter partition(2,3)
        leave partition(2,3)
        enter partition(2,1)
        leave partition(2,1)
        enter partition(3,3)
        leave partition(3,3)
      leave quicksort(2,3)
    leave quicksort(1,3)
    enter quicksort(5,9)
      enter partition(5,9)
      leave partition(5,9)
      enter quicksort(5,5)
      leave quicksort(5,5)
      enter quicksort(7,9)
        …
      leave quicksort(7,9)
    leave quicksort(5,9)
  leave(quicksort(1,9)
leave main
```





Procedure Activation Records/Frames

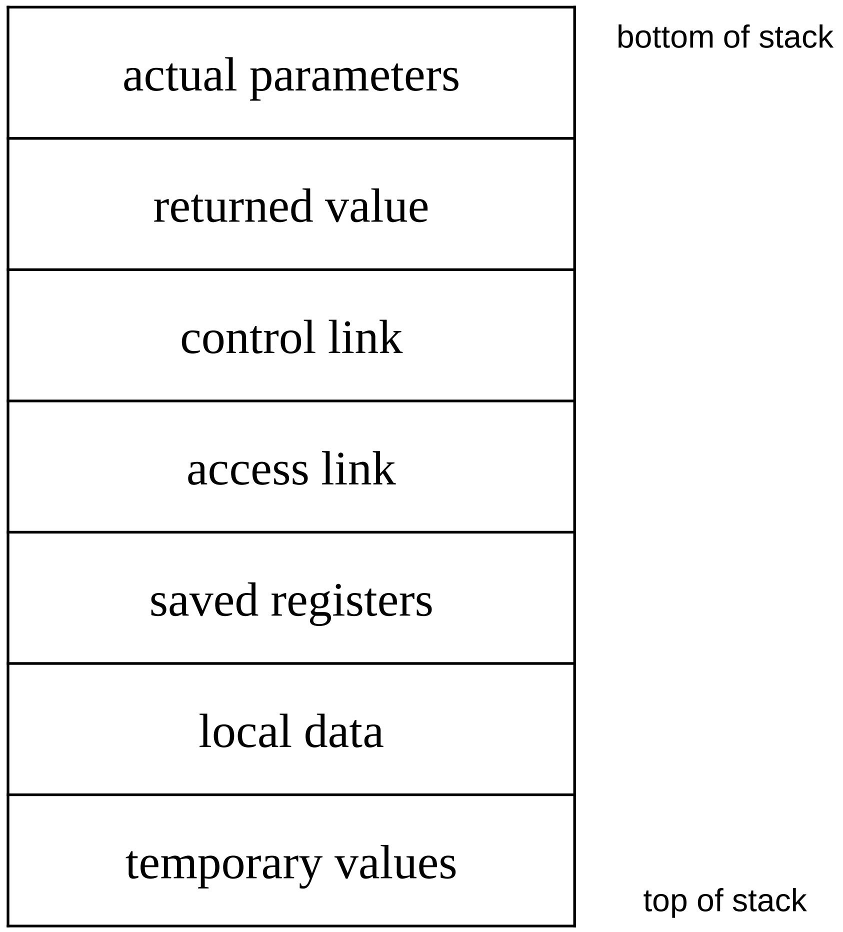
- Or: where do local variables go?

- At run time, a procedure's local variable and book-keeping data stored on the stack
  - called **activation frame** or **activation record**

- Nested activation records
  - **lifetime**: a procedure's activation frame is created when the procedure starts executing and destroyed when the procedure returns to its caller.
  - leads to a *nesting in time*
  - recursive functions lead to several activation frames of the same function on the stack

Procedure Activation Frame

- **General Layout**



Procedure Activation Frame

- Setting up/destroying a procedure activation frame
  - typically a “joint-venture”
    - caller prepares actual parameters and space for returned values
    - callee is responsible for the control+access links, saving and restoring the machine status, plus managing local data and temporaries
  - other division possible. Design issues to consider
    - if P is called from *n* different locations, the code that goes into the caller must be repeated *n* times → typically, we try to push as much work to the callee as possible

Procedure Activation Frame

- Design of activation frames and a calling convention
  - calling convention
    “contract” between caller and callee on how the activation frame is organized, how to pass parameters (and results), plus which registers can be overwritten (aka caller-saved) and which must be preserved (aka callee-saved)
  - if you design your own system, you can do whatever the H/W allows you to do
  - on a real system, we have to follow the ABI calling convention in order to invoke system calls and library functions

Procedure Activation Frame

- Design principles for activation frames and the calling convention
  - values that are shared between the caller and the callee (parameters, result values) should be placed at the beginning of the callee's activation record
    - caller can place parameters without knowing the callee frame layout
    - can support functions with a variable number of parameters
  - fixed-length items are placed in the middle. These typically include the control link, the access link, and saved registers
    - identical code for all procedures
    - easier for debuggers to decipher stack contents
  - variable-length items are placed towards the end.
    - compiler can compute addresses of fixed items
    - compiler can add new temp values easily during register allocation, etc.

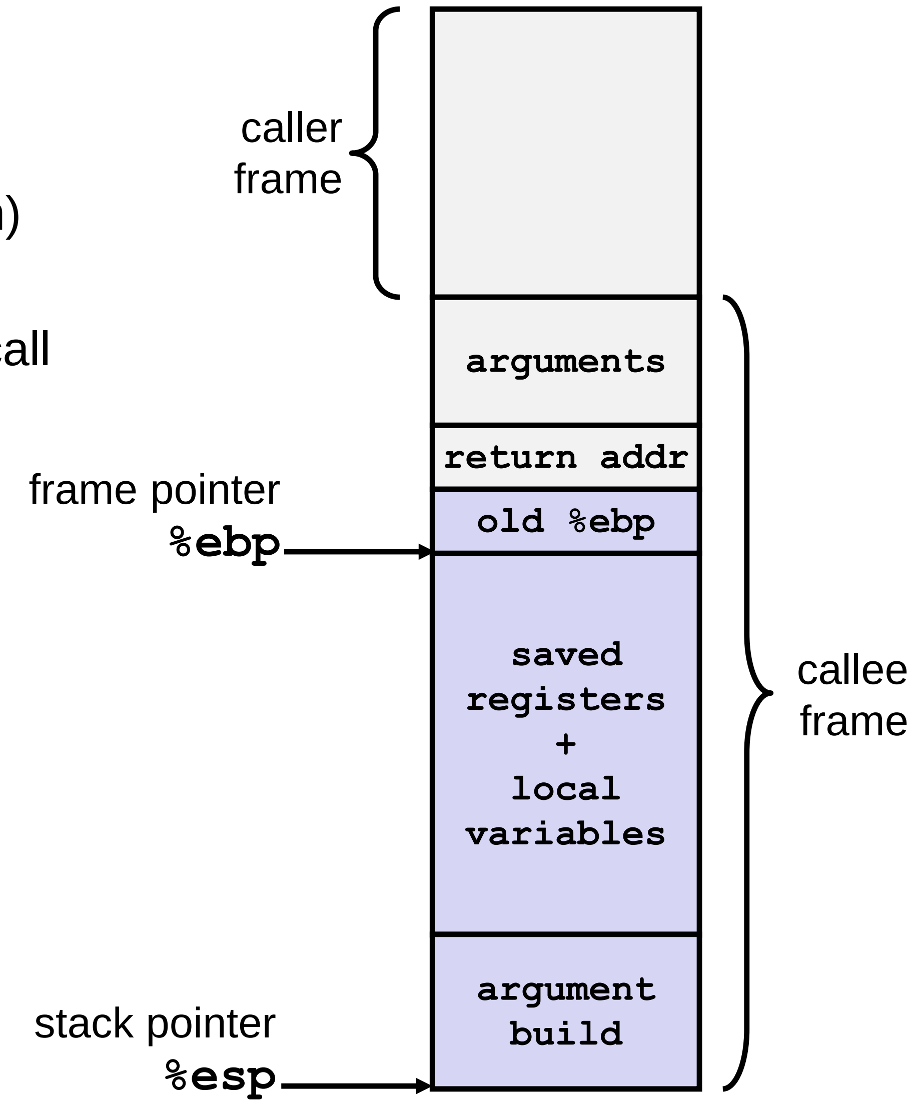
Procedure Activation Frame

- Design principles for activation frames and the calling convention
  - top of stack must be located judiciously
    Dragon book suggests to have it point to the end of the fixed-length fields

    + can access fixed-length fields at a constant offset
    - debugger doesn’t know where the stack “ends”
    - not a feasible approach if the hardware has no separate interrupt stack

Procedure Activation Frame

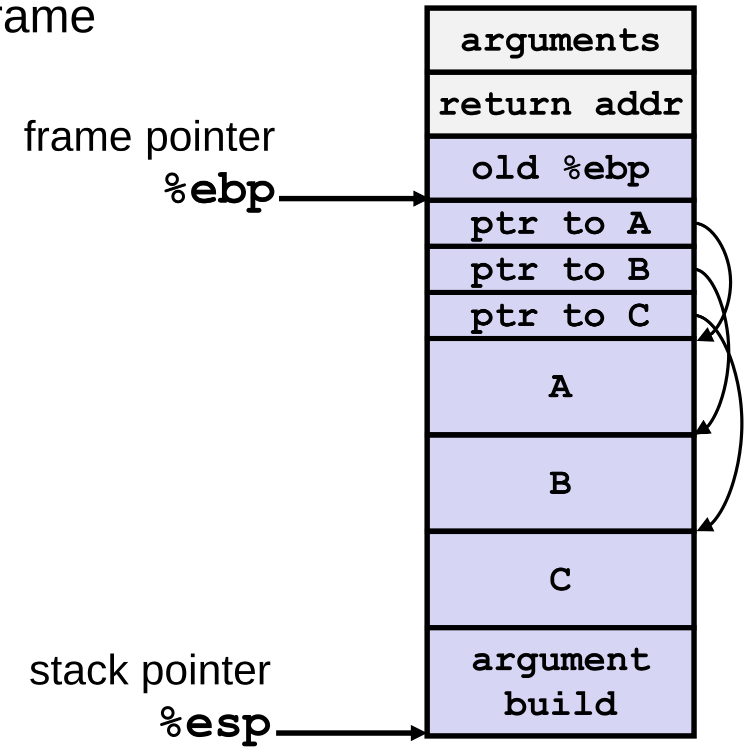
- IA32/Linux Stack Frame
  - Current Stack Frame (Top to Bottom)
    - static/dynamic "argument build" parameters for functions about to call
    - local variables if can't be kept in registers
    - saved register context
    - old frame pointer
  - Caller Stack Frame
    - return address
      - pushed by **`call`** instruction
    - arguments for this call



Procedure Activation Frame

- Variable-Length Data on the Stack
  - sometimes useful to avoid the overhead of dynamic memory allocation
  - must be careful that values do not escape
  - place variables at end of activation frame

```
void foo(int n) {
  int A[n];
  int B[n];
  int C[n];
  int i;

  …

}
```



Access to Non-Local Data

- Access to global variables
  - stored in static data section
  - static addresses
  - access easy
  - static local variables = global variables with a limited scope ("syntactic sugar")

- Access to local variables of enclosing procedures
  - two strategies
    - access links
    - displays (not discussed here, see textbook 7.3.8)

Accessing Data of Enclosing Procedures

- Necessary for languages with support for nested procedures
  - Algol, Pascal, Oberon, ML, …

```
procedure A(paramA: integer);
var localA: integer;

  procedure B(paramB: integer);
  var localB: integer;
  begin
    localA := paramA + paramB
  end B;

  procedure C(paramC: integer);
  var localC: integer;

    procedure D(paramD: integer);
    var localD: integer;
    begin
     …
    end D;

  begin
    B(paramA)
  end C;

begin
  C(5)
end A;
```

Accessing Data of Enclosing Procedures

- Necessary for languages with support for nested procedures
  - cannot determine the location of the data statically

```
procedure A();
var localA: integer;

  procedure B();
  begin
    … := localA;
  end B;

  procedure C(n: integer);
  begin
    if (n > 0) C(n-1);
    B()
  end C;

begin
  B();
  C(3)
end A;
```

Accessing Data of Enclosing Procedures

- Nesting depth of procedures

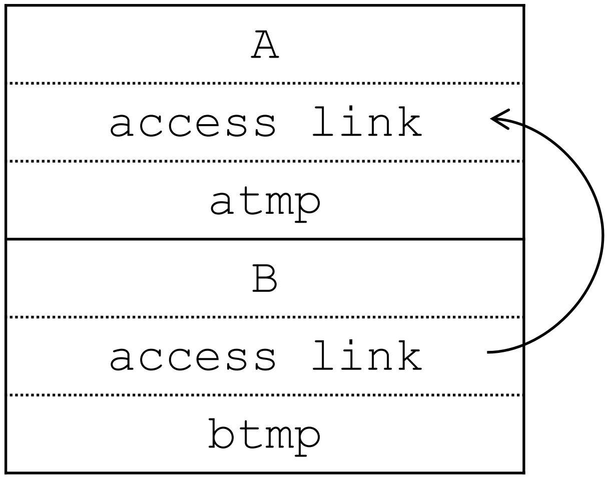
```
procedure A();                  ←  level 1

  procedure B();                ←  level 2
  begin
    …
  end B;

  procedure C();                ←  level 2

    procedure D();              ←  level 3
    begin
      …
    end D;

  begin
    …
  end C;

begin
  …
end A;
```

Accessing Data of Enclosing Procedures

- **Access Links**
  - point to the most recent activation of the immediately enclosing procedure

  consider:

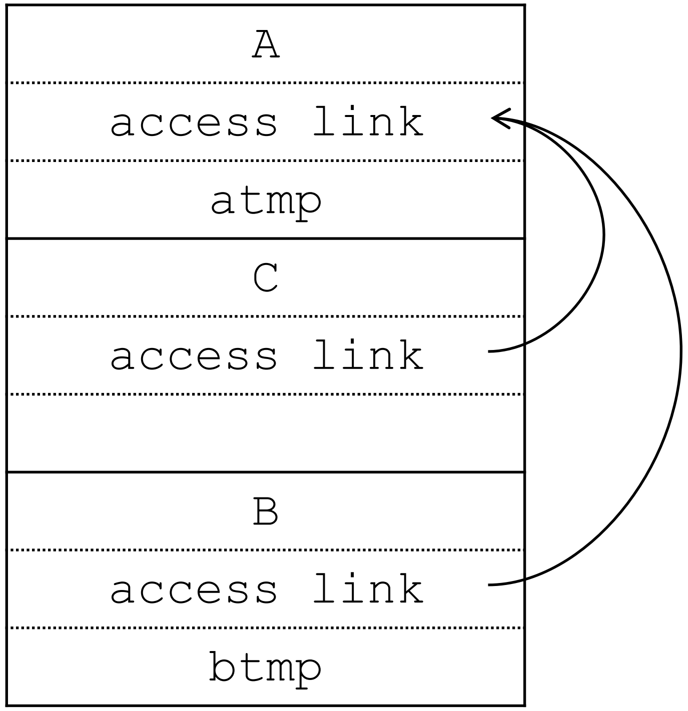
```
procedure A;
var atmp: integer;

  procedure B;
  var btmp: integer;
  begin
    … := atmp + btmp;
  end B;

begin
  B;
end A;
```



Access Links

- now consider:

```
procedure A;
var atmp: integer;

  procedure B;
  var btmp: integer;
  begin
    … := atmp + btmp;
  end B;

  procedure C;
  begin
    B;
  end C;

begin
  B;
  C;
end A;
```



Access Links in Procedure Activation Frames

- **Computing Access Links**
  - for calls by name the link depth can be computed statically

    q calls p

    - nesting depth of p > q: p must be defined in q, activation link points one up (call graph A → B as shown before)

    - same nesting depth: copy activation record of q over to p
      this includes recursive calls
      (call graph A → C → B as shown before)

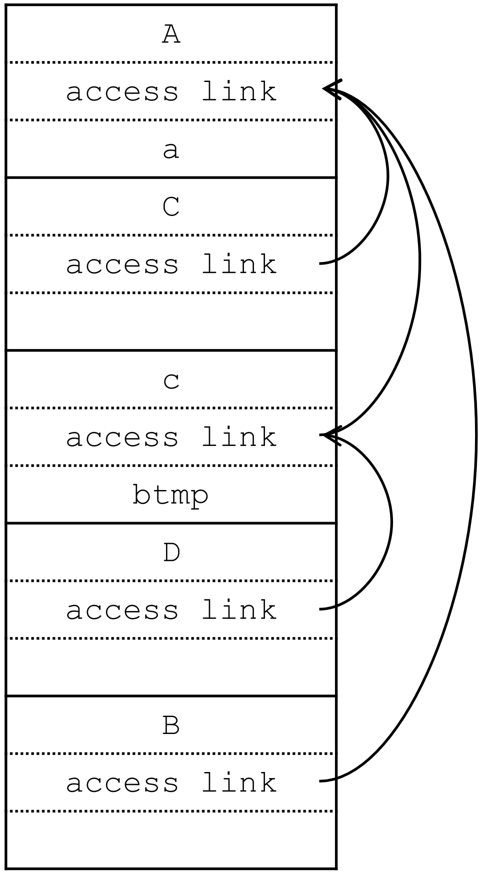
Access Links in Procedure Activation Frames

- **Computing Access Links**
  - for calls by name the link depth can be computed statically

    q calls p
    - nesting depth of p < q: p defined outside q, but in common ancestor. Activation link must point to that ancestor. To find the access link, the compiler inserts code that hops n_q-n_p+1 times starting at q.
    - example:
      dynamic call graph A → C → C → D → B

      D nesting depth 3
      B nesting depth 2
      → follow 3-2+1 hops starting at D to get B's access link

```
procedure A;
var a: integer;
  procedure B;
  begin
  end B;

  procedure C;
    procedure D;
    begin
      B;
    end D;
  begin
    if (…) C;
    D;
  end C;
begin
  C;
end A;
```



Access Links in Procedure Activation Frames

- **Computing Access Links**
  - the link depth cannot be computed statically for function pointers
    - solution: pass access link along with function pointer