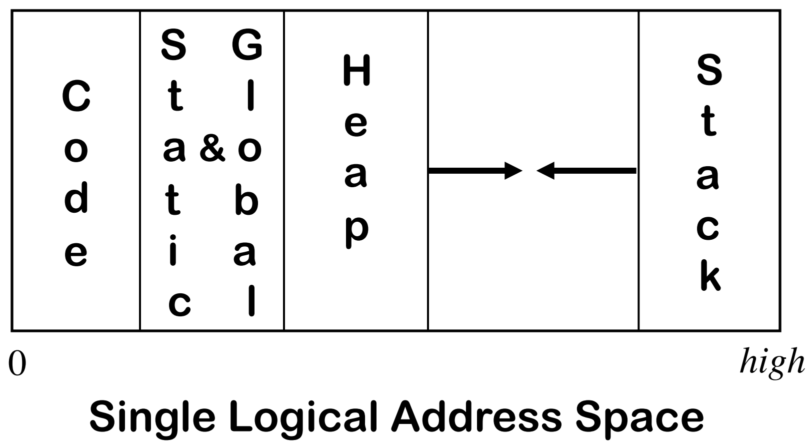
The Procedure Abstraction
Part III: Allocating Storage & Establishing Addressability

Placing Run-time Data Structures

Classic Organization

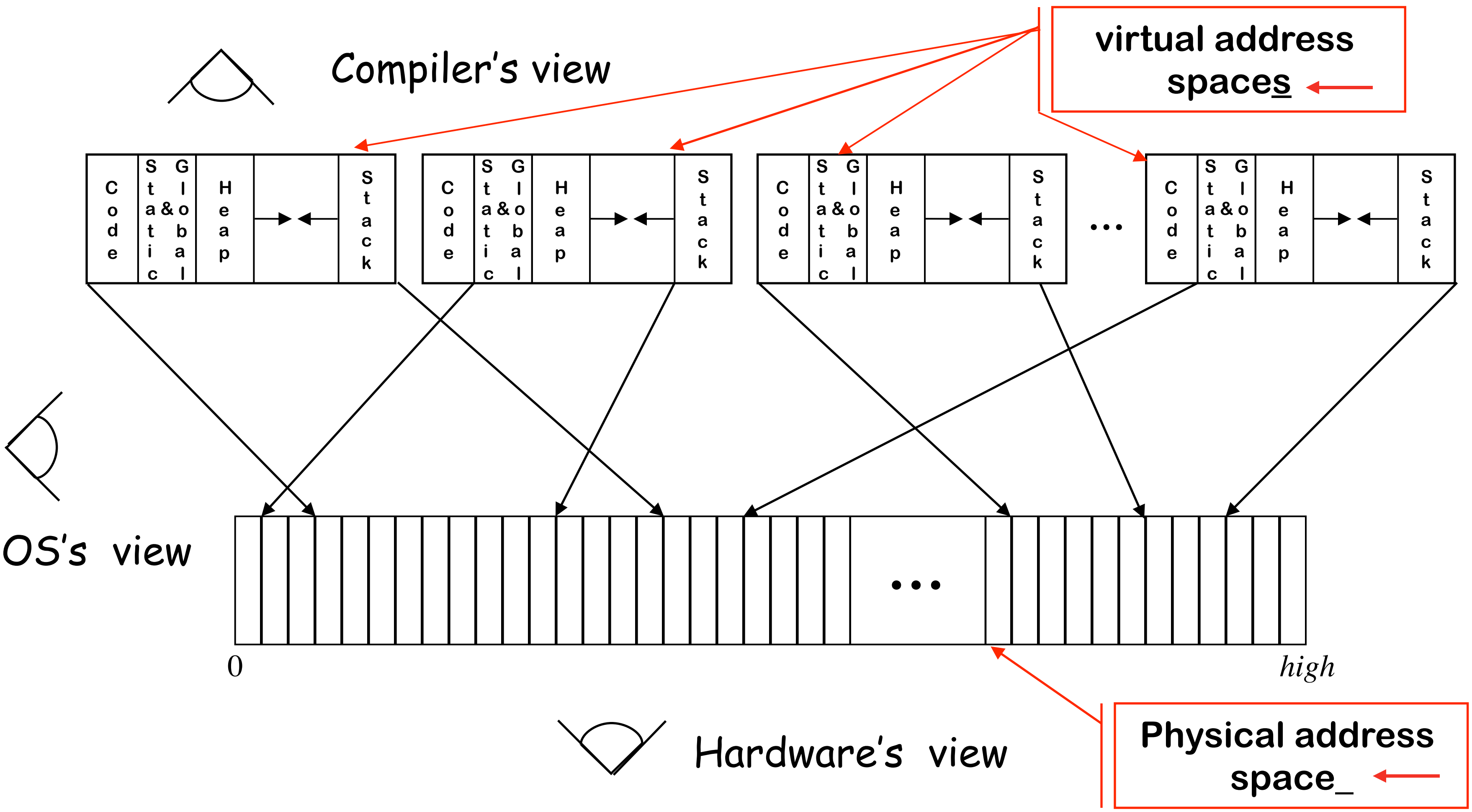


- Better utilization if stack & heap grow toward each other
- Very old result (Knuth)
- Code & data separate or interleaved
- Uses address space, not allocated memory

- Code, static, & global data have known size
  - Use symbolic labels in the code
- Heap & stack both grow & shrink over time
- This is a <u>virtual</u> address space

How Does This Really Work?

The Big Picture



Where Do Local Variables Live?

A Simplistic model

- Allocate a data area for each distinct scope
- One data area per "sheaf" in scoped table

What about recursion?

- Need a data area per invocation (or activation) of a scope
- We call this the scope's activation record
- The compiler can also store control information there !

More complex scheme

- One activation record (AR) per procedure instance
- All the procedure's scopes share a single AR *(may share space)*
- Static relationship between scopes in single procedure

Used this way, "static" means knowable at compile time (and, therefore, fixed).

Translating Local Names

How does the compiler represent a specific instance of *x* ?

- Name is translated into a *static coordinate*
  - → *< level,offset >* pair
  - → "*level*" is lexical nesting level of the procedure
  - → "*offset*" is *unique* within that scope
- Subsequent code will use the static coordinate to generate addresses and references
- "*level*" is a function of the table in which *x* is found
  - → Stored in the entry for each *x*
- "*offset*" must be assigned and stored in the symbol table
  - → Assigned at compile time
  - → Known at compile time
  - → Used to generate code that executes at run-time

Storage for Blocks within a Single Procedure

```
B0: {
        int a, b, c
B1:         {
         int v, b, x, w
B2:            {
               int x, y, z
               ….
         }
B3:            {
               int x, a, v
               …
         }
         …
       }
    …
}
```

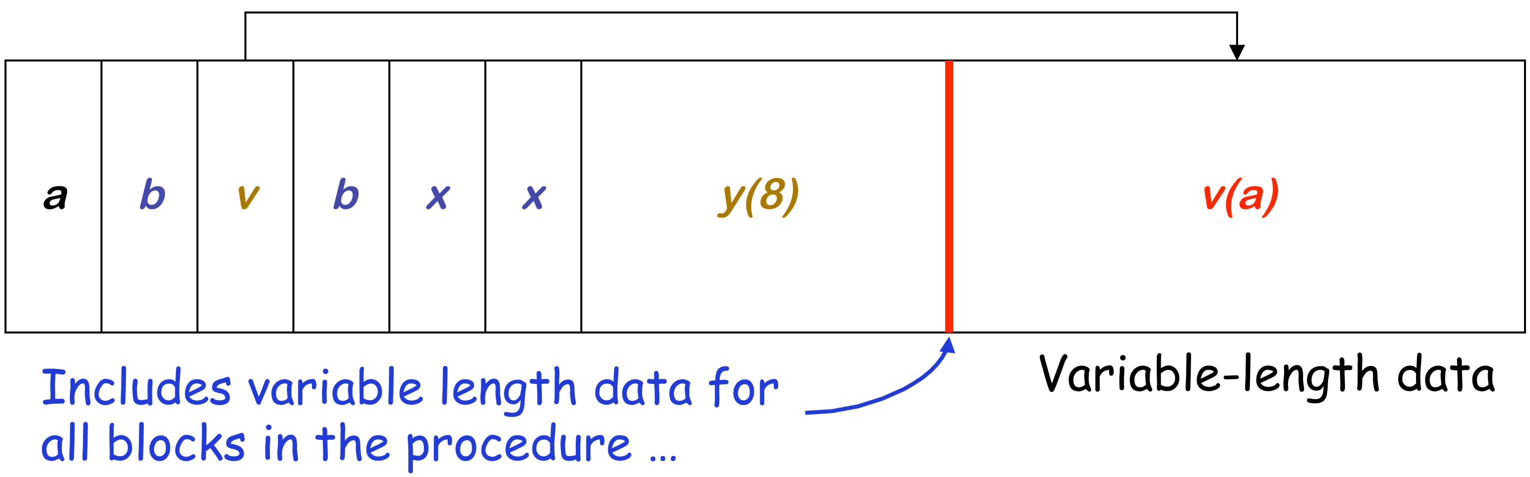
- Fixed length data can always be at a constant offset from the beginning of a procedure
  - In our example, the *a* declared at level 0 will always be the first data element, stored at byte 0 in the fixed-length data area
  - The *x* declared at level 1 will always be the sixth data item, stored at byte 20 in the fixed data area
  - The *x* declared at level 2 will always be the eighth data item, stored at byte 28 in the fixed data area
  - But what about the a declared in the second block at level 2?

Variable-length Data

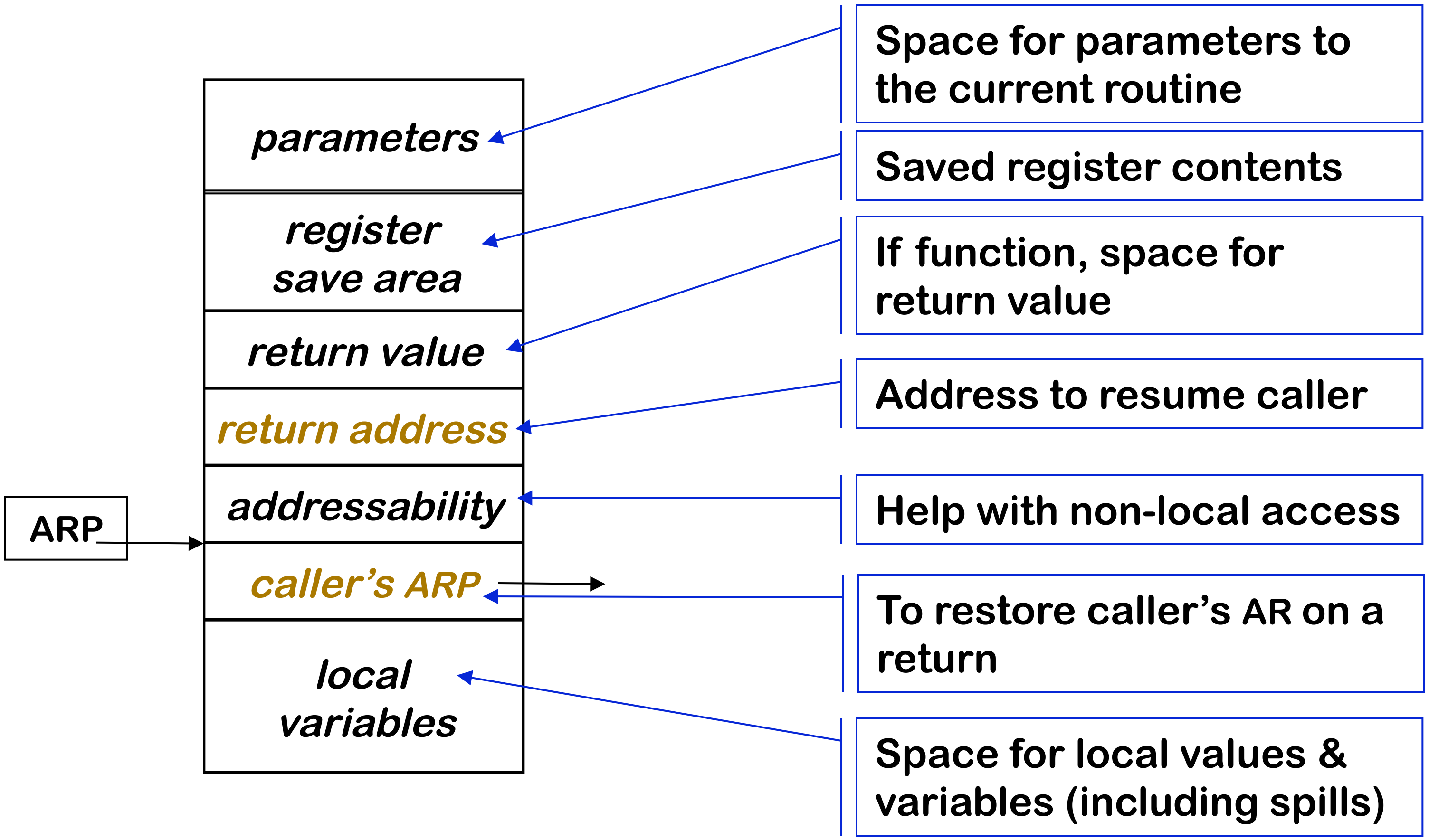
```
B0: {
        int a, b
        … assign value to
    a
B1:         {
          int v(a), b, x
B2:                 {
                    int x, y(8)
                    ….
          }
```

Arrays

- If size is fixed at compile time, store in fixed-length data area
- If size is variable, store descriptor in fixed length area, with pointer to variable length area
- Variable-length data area is assigned at the end of the fixed length area for block in which it is allocated



Activation Record Basics



One AR for each invocation of a procedure

Activation Record Details

How does the compiler find the variables?

- They are at known offsets from the AR pointer
- The static coordinate leads to a "loadAI" operation
  - → Level specifies an ARP, offset is the constant

Variable-length data

- If AR can be extended, put it after local variables
- Leave a pointer at a known offset from ARP
- Otherwise, put variable-length data on the heap

Initializing local variables

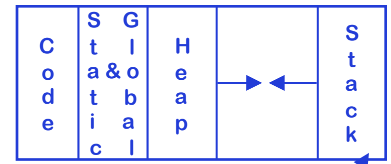
- Must generate explicit code to store the values
- Among the procedure's first actions

Activation Record Details

Where do activation records live?

- If lifetime of AR matches lifetime of invocation, *AND*
- If code normally executes a "return"

⇒ Keep ARs on a stack



Yes! This stack.

- If a procedure can outlive its caller, *OR*
- If it can return an object that can reference its execution state

⇒ ARs <u>must</u> be kept in the heap

- If a procedure makes no calls

⇒ AR can be allocated statically

Efficiency prefers static, stack, then heap

Communicating Between Procedures

Most languages provide a parameter passing mechanism

⇒ Expression used at "call site" becomes variable in callee

Two common binding mechanisms

- Call-by-reference passes a pointer to actual parameter
  - → Requires slot in the AR (for address of parameter)
  - → Multiple names with the same address?
- Call-by-value passes a copy of its value at time of call
  - → Requires slot in the AR
  - → Each name gets a unique location
  - → Arrays are mostly passed by reference, not value

call fee(x,x,x);

(may have same value)

- Can always use global variables …

Establishing Addressability

Must create base addresses

- Global & static variables
  - → Construct a label by mangling names (*i.e.,* &_fee)

- Local variables
  - → Convert to static data coordinate and use **ARP** + offset

- Local variables of other procedures
  - → Convert to static coordinates
  - → Find appropriate **ARP**
  - → Use that **ARP** + offset

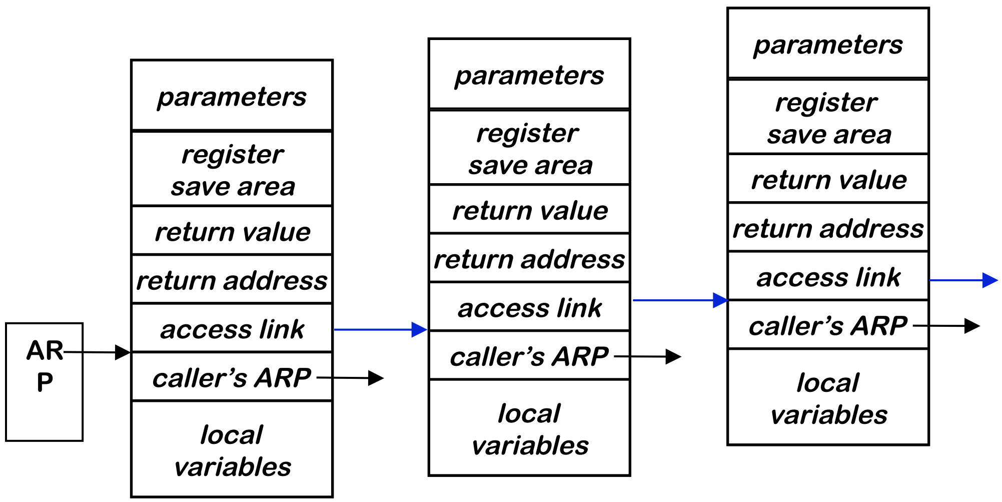
{ **Must find the right AR**

Need links to nameable ARs

Establishing Addressability

Using access links

- Each AR has a pointer to AR of lexical ancestor
- Lexical ancestor need not be the caller



- Reference to <*p*,16> runs up access link chain to *p*
- Cost of access is proportional to lexical distance

Establishing Addressability

Using access links

SC	**Generated Code**
<2,8>	loadAI r_0, 8 ⇒ r_2
<1,12>	loadAI r_0, -4 ⇒ r_1 loadAI r_1, 12 ⇒ r_2
<0,16>	loadAI r_0, -4 ⇒ r_1 loadAI r_1, -4 ⇒ r_1 loadAI r_1, 16 ⇒ r_2

Assume

- Current lexical level is 2
- Access link is at **ARP** - 4

Maintaining access link

- Calling level *k*+1
  → Use current **ARP** as link
- Calling level *j* < *k*
  → Find ARP for *j* –1
  → Use that ARP as link

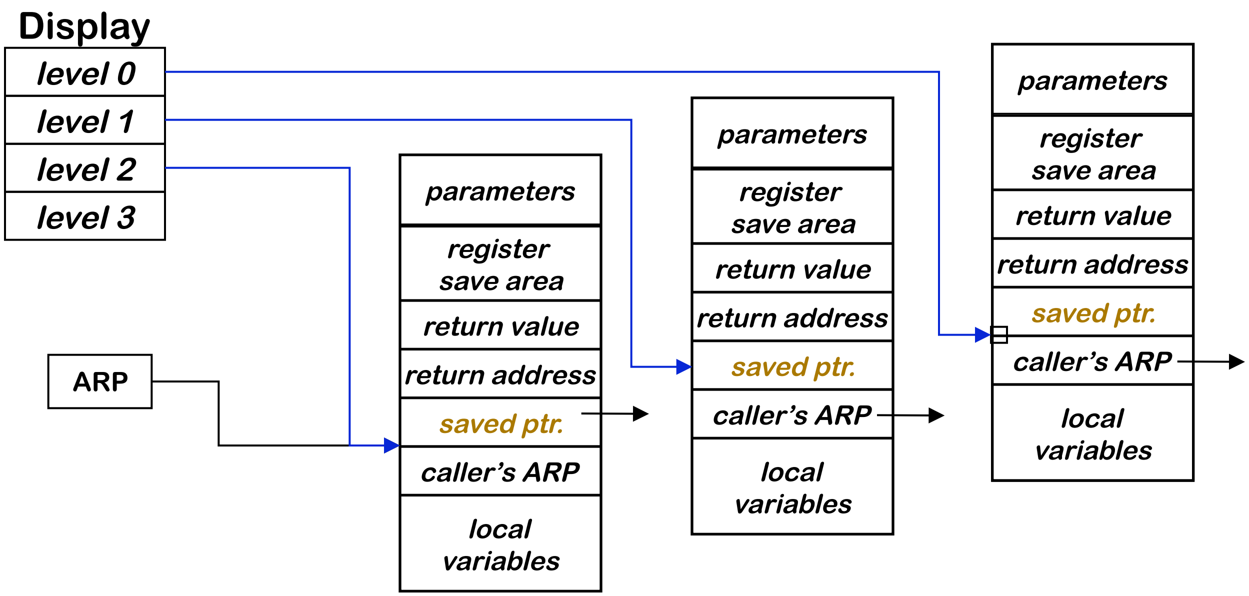
Access & maintenance cost varies with level

All accesses are relative to ARP (r_0)

Establishing Addressability

Using a display

- Global array of pointer to nameable ARs
- Needed ARP is an array access away



- Reference to ‹*p*,16› looks up *p*'s ARP in display & adds 16
- Cost of access is constant (ARP + offset)

Establishing Addressability

Using a display

SC	Generated Code
<2,8>	loadAI r_0, 8 ⇒ r_2
<1,12>	loadI _disp ⇒ r_1 loadAI r_1, 4 ⇒ r_1 loadAI r_1, 12 ⇒ r_2
<0,16>	loadI _disp ⇒ r_1 loadAI r_1, 16 ⇒ r_2

Desired AR is at _disp + 4 x *level*

Assume
- Current lexical level is 2
- Display is at label _disp

Maintaining access link
- On entry to level *j*
  → Save level *j* entry into AR (Saved Ptr field)
  → Store ARP in level *j* slot
- On exit from level *j*
  → Restore level *j* entry

Access & maintenance costs are fixed
Address of display may consume a register

Establishing Addressability

Access links versus Display

- Each adds some overhead to each call
- Access links costs vary with level of reference
  - → Overhead only incurred on references & calls
  - → If ARs outlive the procedure, access links still work
- Display costs are fixed for all references
  - → References & calls must load display address
  - → Typically, this requires a register *(rematerialization)*

Your mileage will vary

- Depends on ratio of non-local accesses to calls
- Extra register can make a difference in overall speed

For either scheme to work, the compiler must insert code into each procedure call & return

Procedure Linkages

How do procedure calls actually work?

- At compile time, callee may not be available for inspection
  - → Different calls may be in different compilation units
  - → Compiler may not know system code from user code
  - → All calls must use the same protocol

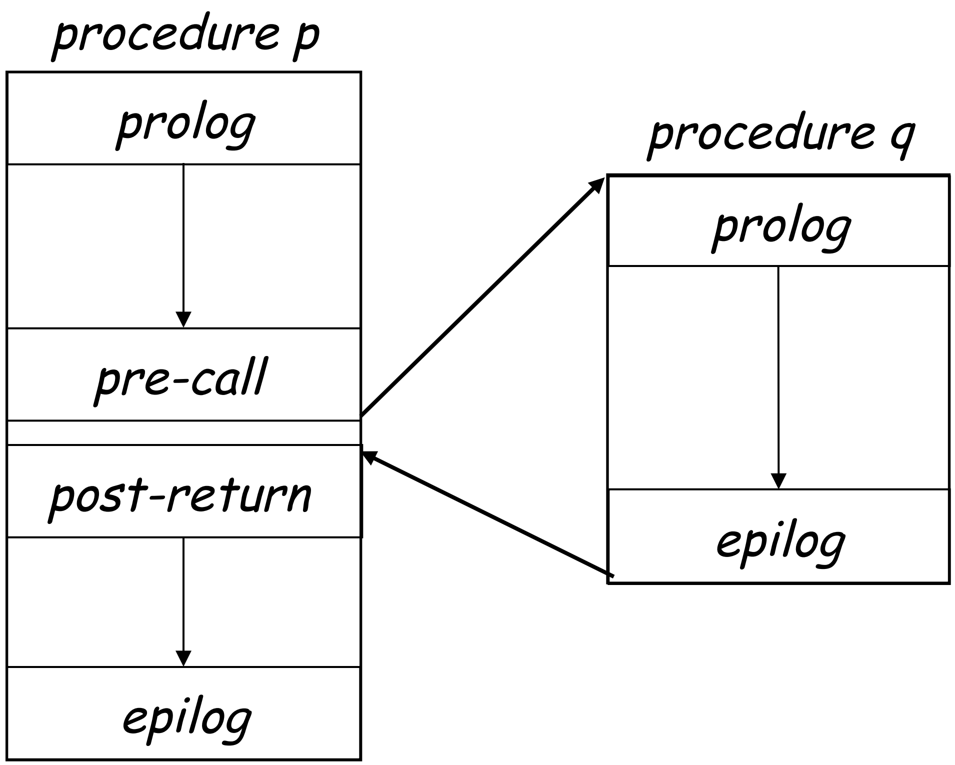
Compiler must use a standard sequence of operations

- Enforces control & data abstractions
- Divides responsibility between caller & callee

Usually a system-wide agreement *(for interoperability)*

Procedure Linkages

Standard procedure linkage



Procedure has
- standard prolog
- standard epilog

Each call involves a
- pre-call sequence
- post-return sequence

These are completely predictable from the call site ⇒ depend on the number & type of the actual parameters

Procedure Linkages

Pre-call Sequence

- Sets up callee's basic AR
- Helps preserve its own environment

The Details

- Allocate space for the callee's AR
  - → except space for local variables
- Evaluates each parameter & stores value or address
- Saves return address, caller's ARP into callee's AR
- If access links are used
  - → Find appropriate lexical ancestor & copy into callee's AR
- Save any caller-save registers
  - → Save into space in caller's AR
- Jump to address of callee's prolog code

Procedure Linkages

Post-return Sequence

- Finish restoring caller's environment
- Place any value back where it belongs

The Details

- Copy return value from callee's AR, if necessary
- Free the callee's AR
- Restore any caller-save registers
- Restore any call-by-reference parameters to registers, if needed
  - → Also copy back call-by-value/result parameters
- Continue execution after the call

Procedure Linkages

Prolog Code

- Finish setting up callee's environment
- Preserve parts of caller's environment that will be disturbed

The Details

- Preserve any callee-save registers
- If display is being used
  - → Save display entry for current lexical level
  - → Store current ARP into display for current lexical level
- Allocate space for local data
  - → Easiest scenario is to extend the AR
- Find any static data areas referenced in the callee
- Handle any local variable initializations

With heap allocated AR, may need to use a separate heap object for local variables

Procedure Linkages

Epilog Code

- Wind up the business of the callee
- Start restoring the caller's environment

The Details

- Store return value?
  - → Some implementations do this on the return statement
  - → Others have return assign it & epilog store it into caller's AR
- Restore callee-save registers
- Free space for local data, if necessary (on the heap)
- Load return address from AR
- Restore caller's ARP
- Jump to the return address

If ARs are stack allocated, this may not be necessary. (Caller can reset stacktop to its pre-call value.)

Back to Activation Records

If activation records are stored on the stack

- Easy to extend — simply bump top of stack pointer
- Caller & callee share responsibility
  - → Caller can push parameters, space for registers, return value slot, return address, addressability info, & its own **ARP**
  - → Callee can push space for local variables (fixed & variable size)

If activation records are stored on the heap

- Hard to extend
- Caller passes everything it can in registers
- Callee allocates AR & stores register contents into it
  - → Extra parameters stored in caller's **AR** !

Static is easy