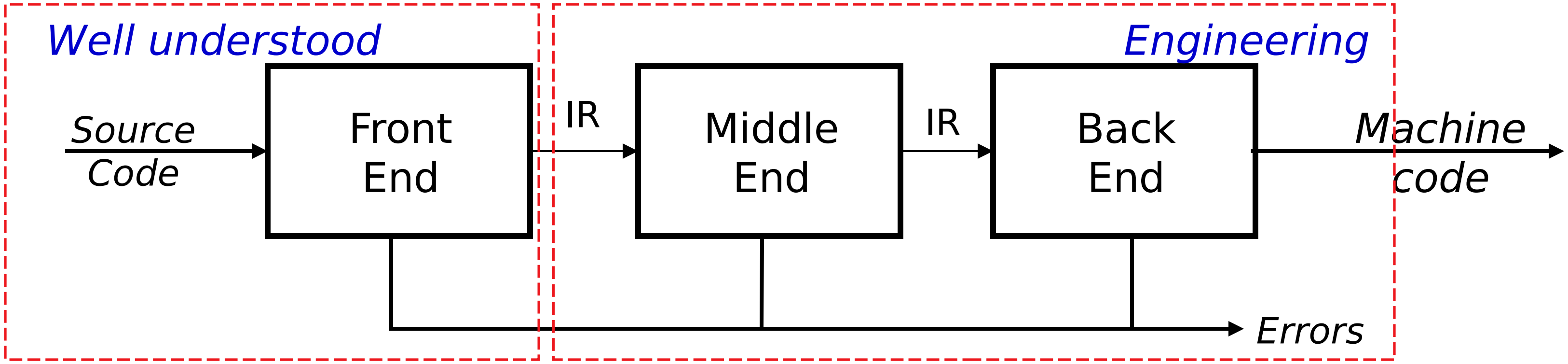
The Procedure Abstraction

Procedure Abstraction

- Begins Chapter 6 in EAC
- The compiler must deal with interface between compile time and run time (static versus dynamic)
  - → Most of the tricky issues arise in implementing “procedures”

- Issues
  - → Compile-time versus run-time behaviour
  - → Finding storage for EVERYTHING, and mapping names to addresses
  - → Generating code to compute addresses that the compiler cannot know !
  - → Interfaces with other programs, other languages, and the OS
  - → Efficiency of implementation

Where are we?



The latter half of a compiler contains more open problems, more challenges, and more grey areas than the front half

- This is “compilation,” as opposed to “parsing” or “translation”
- Implementing promised behaviour
  - → What defines the meaning of the program
- Managing target machine resources
  - → Registers, memory, issue slots, locality, power, …
  - → These issues determine the quality of the compiler

The Procedure: Three Abstractions

- Control Abstraction
  - → Well defined entries & exits
  - → Mechanism to return control to caller
  - → Some notion of parameterisation (usually)

- Clean Name Space
  - → Clean slate for writing locally visible names
  - → Local names may obscure identical, non-local names
  - → Local names cannot be seen outside

- External Interface
  - → Access is by procedure name & parameters
  - → Clear protection for both caller & callee
  - → Invoked procedure can ignore calling context

- Procedures permit a critical separation of concerns

The Procedure (Realist's View)

Procedures are the key to building large systems

- Requires system-wide contract
  - → Conventions on memory layout, protection, resource allocation calling sequences, & error handling
  - → Must involve architecture (**ISA**), **OS**, & compiler
- Provides shared access to system-wide facilities
  - → Storage management, flow of control, interrupts
  - → Interface to input/output devices, protection facilities, timers, synchronization flags, counters, …
- Establishes a private context
  - → Create private storage for each procedure invocation
  - → Encapsulate information about control flow & data abstractions

The Procedure (Realist's View)

Procedures allow us to use separate compilation

- Separate compilation allows us to build non-trivial programs
- Keeps compile times reasonable
- Lets multiple programmers collaborate
- Requires independent procedures

Without separate compilation, we *would not* build large systems

The procedure linkage convention

- Ensures that each procedure inherits a valid run-time environment and that the callers environment is restored on return
  - → The compiler must generate code to ensure this happens according to conventions established by the system

The Procedure (More Abstract View)

A procedure is an abstract structure constructed via software

Underlying hardware directly supports little of the abstraction—it understands bits, bytes, integers, reals, and addresses, but not:

- Entries and exits
- Interfaces
- Call and return mechanisms
  - → may be a special instruction to save context at point of call
- Name space
- Nested scopes

All these are established by a carefully-crafted system of mechanisms provided by compiler, run-time system, linkage editor and loader, and OS

Run Time versus Compile Time

These concepts are often confusing to the newcomer

- Linkages execute at run time
- Code for the linkage is emitted at compile time
- The linkage is designed long before either of these

"This issue (compile time versus run time) confuses students more than *any other* issue in Comp 412"—Keith Cooper

The Procedure as a Control Abstraction

The Procedure as a Control Abstraction

Procedures have well-defined control-flow

The Algol-60 procedure call

- Invoked at a call site, with some set of *actual parameters*
- Control returns to call site, immediately after invocation

The Procedure as a Control Abstraction

Procedures have well-defined control-flow

The Algol-60 procedure call

- Invoked at a call site, with some set of *actual parameters*
- Control returns to call site, immediately after invocation

```
...
s = p(10,t,u);
...
```

```
int p(a,b,c)
    int a, b, c;
{
   int   d;
   d = q(c,b);
   ...
}
```

The Procedure as a Control Abstraction

Procedures have well-defined control-flow

The Algol-60 procedure call

- Invoked at a call site, with some set of *actual parameters*
- Control returns to call site, immediately after invocation

```
…
s = p(10,t,u);
…
```

```
int p(a,b,c)
    int a, b, c;
{
   int   d;
   d = q(c,b);
   ...
}
```

```
int q(x,y)
    int x,y;
{
    return x + y;
}
```

The Procedure as a Control Abstraction

Procedures have well-defined control-flow

The Algol-60 procedure call

- Invoked at a call site, with some set of *actual parameters*
- Control returns to call site, immediately after invocation

```
…
s = p(10,t,u);
…
```

```
int p(a,b,c)
    int a, b, c;
{
   int   d;
   d = q(c,b);
   ...
}
```

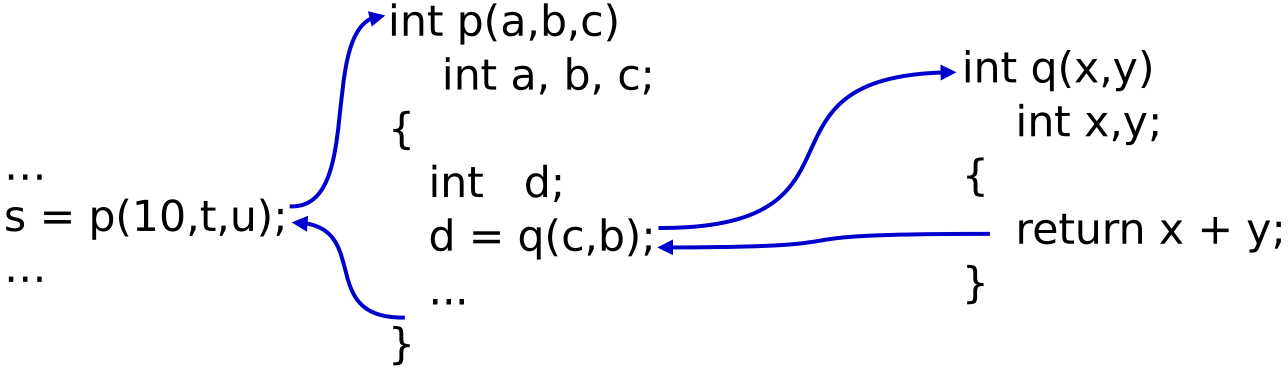
```
int q(x,y)
    int x,y;
{
    return x + y;
}
```

The Procedure as a Control Abstraction

Procedures have well-defined control-flow

The Algol-60 procedure call

- Invoked at a call site, with some set of *actual parameters*
- Control returns to call site, immediately after invocation



The Procedure as a Control Abstraction

Procedures have well-defined control-flow

The Algol-60 procedure call

- Invoked at a call site, with some set of *actual parameters*
- Control returns to call site, immediately after invocation

```
...
s = p(10,t,u);
...
```

```
int p(a,b,c)
    int a, b, c;
{
   int   d;
   d = q(c,b);
   ...
}
```

```
int q(x,y)
    int x,y;
{
    return x + y;
}
```

- Most languages allow recursion

The Procedure as a Control Abstraction

Implementing procedures with this behaviour

- Requires code to save and restore a “return address”
- Must map actual parameters to formal parameters   ($c \rightarrow x$, $b \rightarrow y$)
- Must create storage for local variables (&, maybe, parameters)
  - → *p* needs space for *d* (&, maybe, *a, b*, & *c*)
  - → where does this space go in recursive invocations?

```
…
s = p(10,t,u);
…
```

```
int p(a,b,c)
    int a, b, c;
{
   int   d;
   d = q(c,b);
   ...
}
```

```
int q(x,y)
    int x,y;
{
    return x + y;
}
```

The Procedure as a Control Abstraction

Implementing procedures with this behaviour

- Must preserve p’s state while q executes
  - → recursion causes the real problem here
- Strategy: Create unique location for each procedure activation
  - → Can use a “stack” of memory blocks to hold local storage and return addresses

```
…
s = p(10,t,u);
…
```

```
int p(a,b,c)
    int a, b, c;
{
   int   d;
   d = q(c,b);
   ...
}
```

```
int q(x,y)
    int x,y;
{
    return x + y;
}
```

Compiler <u>emits</u> code that causes all this to happen at run time

The Procedure as a Name Space

The Procedure as a Name Space

Each procedure creates its own name space

- Any name (almost) can be declared locally
- Local names obscure identical non-local names
- Local names cannot be seen outside the procedure
  - → Nested procedures are "inside" by definition
- We call this set of rules & conventions "lexical scoping"

Examples

- C has global, static, local, and *block* scopes *(Fortran-like)*
  - → Blocks can be nested, procedures cannot
- Scheme has global, procedure-wide, and nested scopes *(let)*
  - → Procedure scope (typically) contains formal parameters

The Procedure as a Name Space

Why introduce lexical scoping?

- Provides a compile-time mechanism for binding “free” variables
- Simplifies rules for naming & resolves conflicts

How can the compiler keep track of all those names?

The Problem

- At point *p*, which declaration of *x* is current?
- At run-time, where is *x* found?
- As parser goes in & out of scopes, how does it delete *x*?

The Answer

- Lexically scoped symbol tables (see § 5.7.3)

§ 5.7 in EaC

Lexically-scoped Symbol Tables

The problem

- The compiler needs a distinct record for each declaration
- Nested lexical scopes admit duplicate declarations

The interface

- insert(*name, level*) – creates record for *name* at *level*
- lookup(*name, level*) – returns pointer or index
- delete(*level*) – removes all names declared at *level*

Many implementation schemes have been proposed (see § B.4)

- We'll stay at the conceptual level
- Hash table implementation is tricky, detailed, & fun

Symbol tables are compile-time structures the compiler use to resolve references to names. We'll see the corresponding run-time structures that are used to establish addressability later.

Example

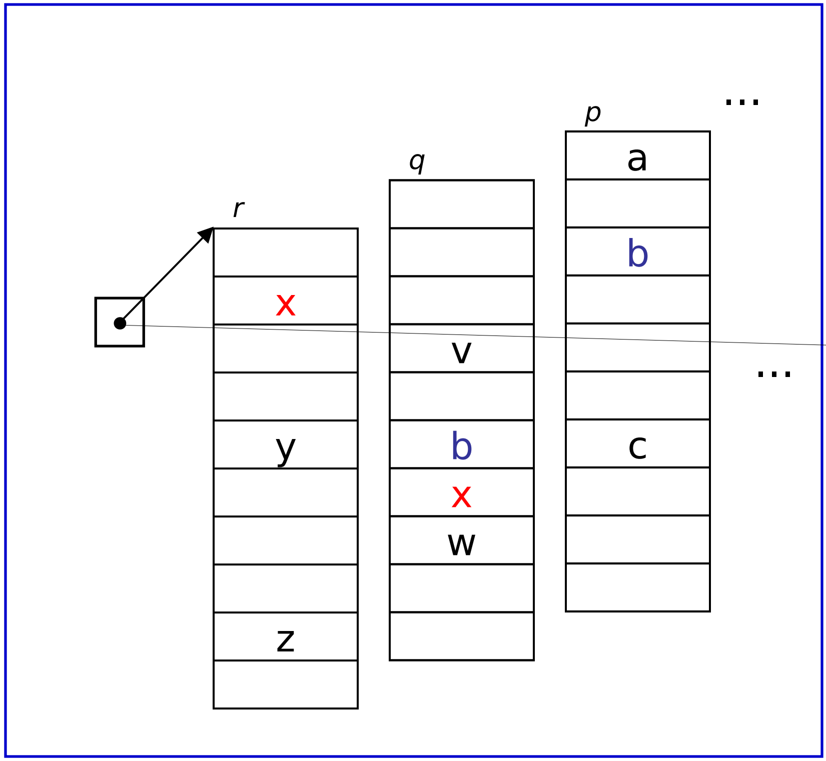
```
procedure p {
      int a, b, c
      procedure q {
            int v, b, x, w
            procedure  r {
                  int x, y, z
                  ....
            }
            procedure s {
                  int x, a, v
                  ...
            }
            … r … s
      }
    … q …
}
```

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

Lexically-scoped Symbol Tables

High-level idea

- Create a new table for each scope
- Chain them together for lookup

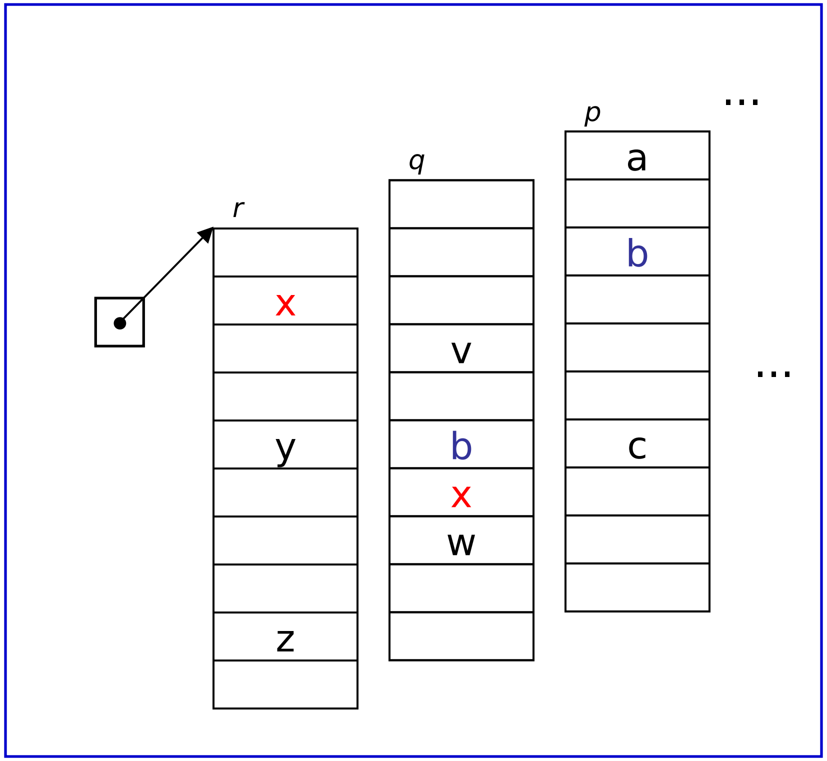



```
procedure p {
    int a, b, c
    procedure q {
        int v, b, x, w
        procedure  r {
            int x, y, z
            ....
        }
        procedure s {
            int x, a, v
            ...
        }
        ... r ... s
    }
    ... q ...
}
```

Lexically-scoped Symbol Tables

High-level idea

- Create a new table for each scope
- Chain them together for lookup



"Sheaf of tables" implementation

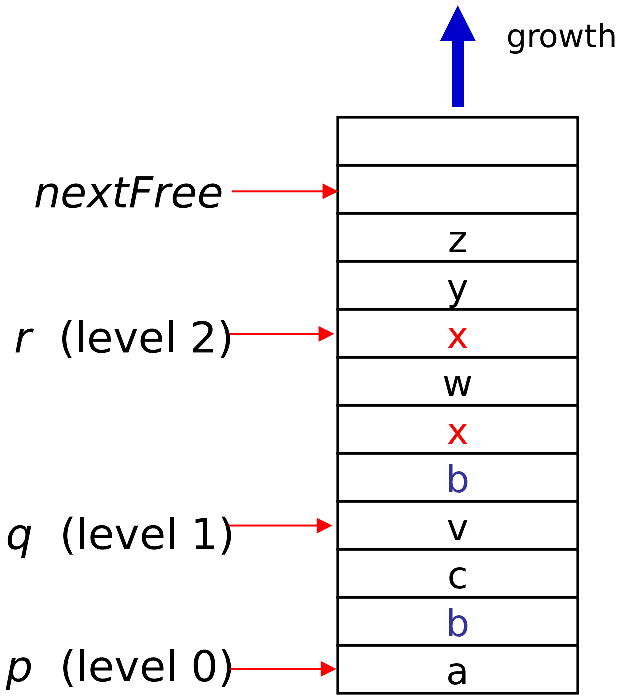
- ***insert()*** **may need to create table**
- **it always inserts at current level**
- ***lookup()*** **walks chain of tables & returns first occurrence of name**
- ***delete()*** **throws away table for level** ***p*****, if it is top table in the chain**

If the compiler must preserve the table (***for, say, the debugger*** **), this idea is actually practical.**

Individual tables can be hash tables.

Implementing Lexically Scoped Symbol Tables

Stack organization



Implementation

- **insert () creates new level pointer if needed and inserts at nextFree**
- **lookup () searches linearly from nextFree–1 forward**
- **delete () sets nextFree to the equal the start location of the level deleted.**

Advantage

- **Uses <u>much</u> less space**

Disadvantage

- **Lookups can be expensive**

The Procedure as an External Interface

The Procedure as an External Interface

OS needs a way to start the program's execution

- Programmer needs a way to indicate where it begins
  - → The "main" procedure in most languages
- When user invokes "ls" at a command line
  - → OS finds the executable
  - → OS creates a process and arranges for it to run "ls"
  - → "ls" is code from the compiler, linked with run-time system
    - ♦ Starts the run-time environment & calls "main"
    - ♦ After main, it shuts down run-time environment & returns
- When "ls" needs system services
  - → It makes a system call, such as fopen()

UNIX/Linux specific discussion

Where Do All These Variables Go?

Automatic & Local

- Keep them in the procedure activation record or in a register
- Automatic $\Rightarrow$ lifetime matches procedure's lifetime

Static

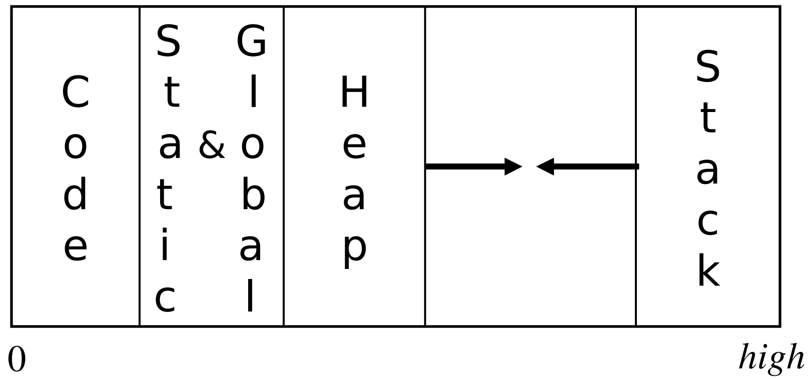
- Procedure scope $\Rightarrow$ storage area affixed with procedure name
  - $\rightarrow$ &_p.x
- File scope $\Rightarrow$ storage area affixed with file name
- Lifetime is entire execution

Global

- One or more named global data areas
- One per variable, or per file, or per program, …
- Lifetime is entire execution

Placing Run-time Data Structures

Classic Organization



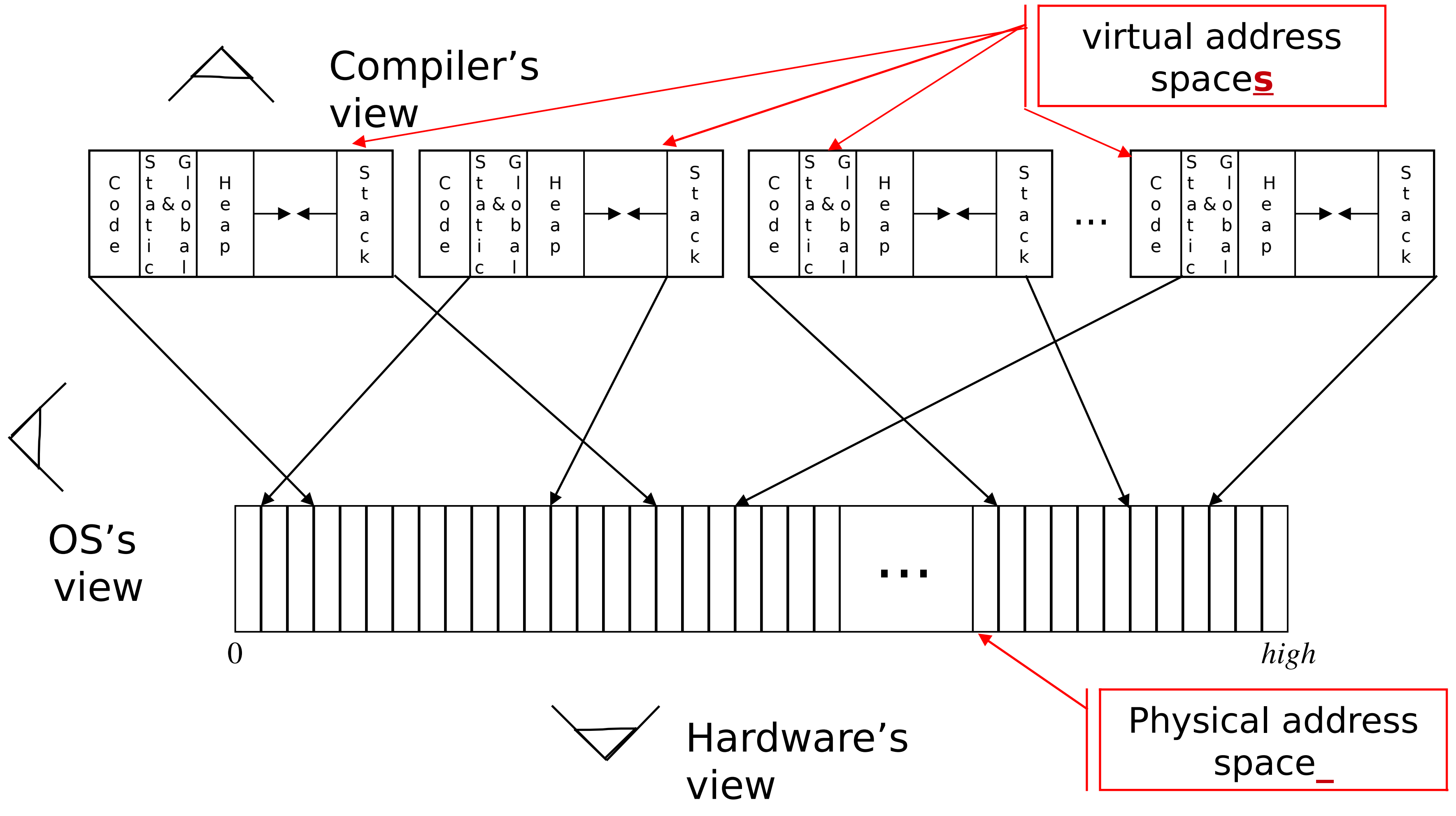
Single Logical Address Space

- 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 virtual 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
                    …
                }
            …
      }
    …
}
```

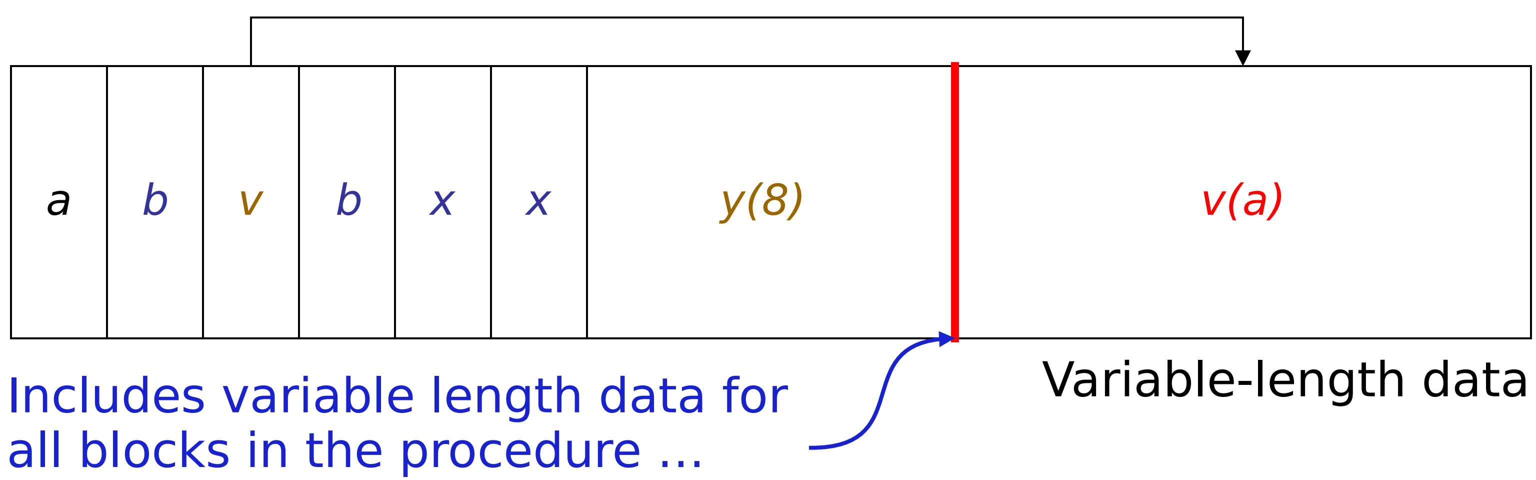
- Fixed length data can always be at a constant offset from the beginning of a procedure
  - → In our example, the *a* declared at level 0 will always be the first data element, stored at byte 0 in the fixed-length data area
  - → The *x* declared at level 1 will always be the sixth data item, stored 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

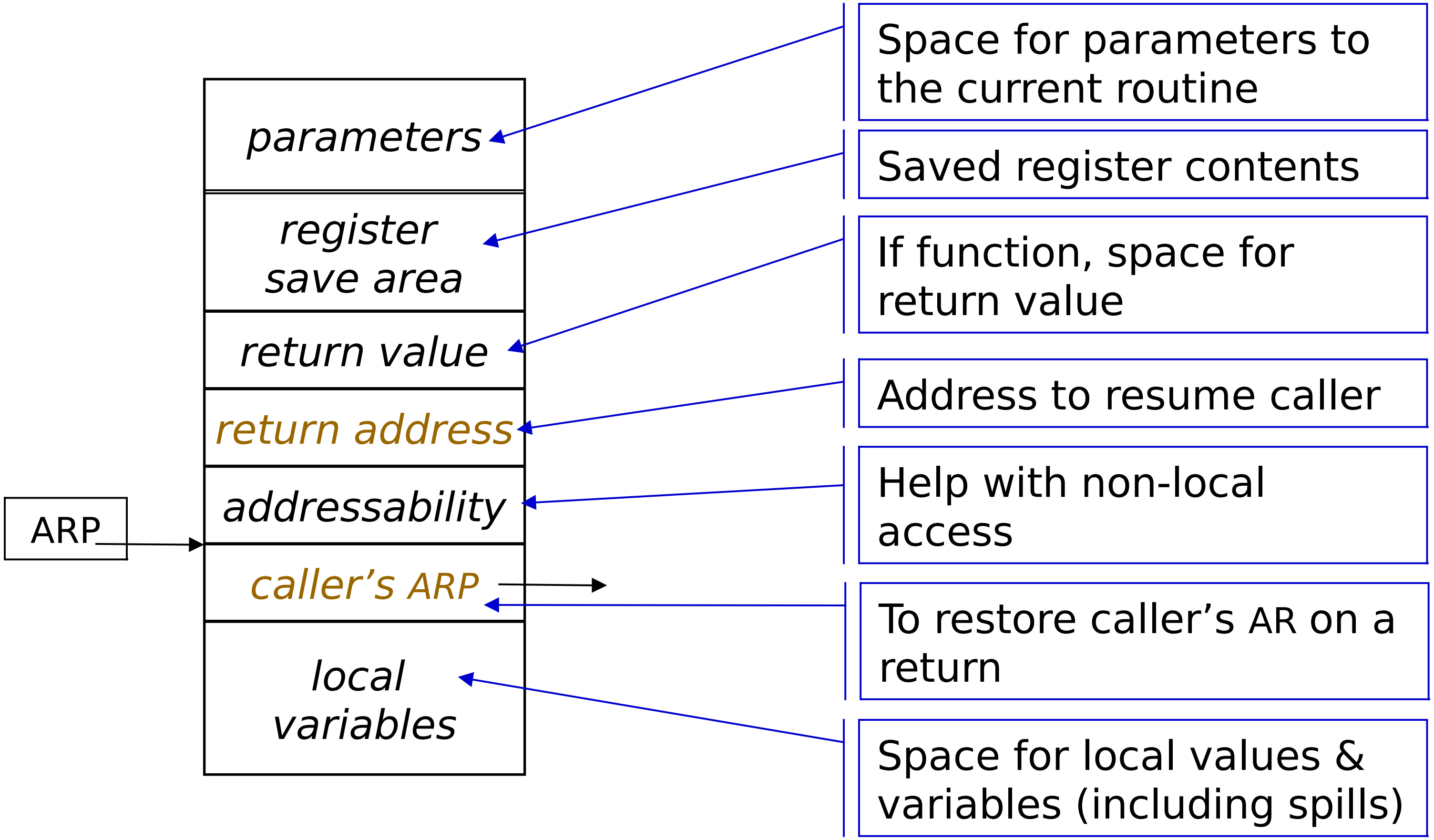
```
B0: {
      int a, b
      … assign value to
    a
B1:       {
            int v(a), b, x
B2:           {
                int x, y(8)
                ….
            }
```

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

load
address
immediate

Variable-length data

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

Initializing local variables

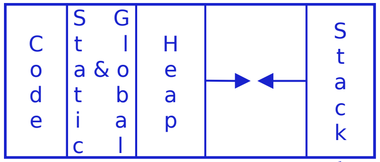
- Must generate explicit code to store the values
- Among the procedure's first actions

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 <u>stack</u>



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 fee(x,x,x);

- Call-by-value passes a copy of its value at time of call
  - → Requires slot in the AR
  - → Each name gets a unique location ***(may have same value)***
  - → Arrays are mostly passed by reference, not 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

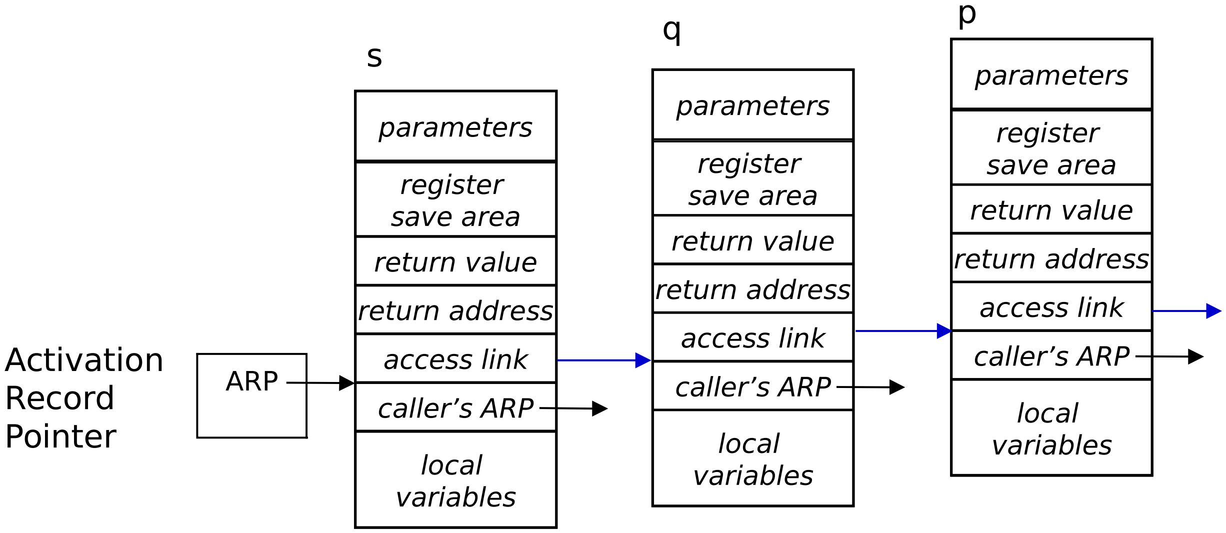
Establishing Addressability

```
procedure p {
    int a, b, c
    procedure q {
        int v, b, x, w
        procedure  r {
            int x, y, z
            ....
        }
        procedure s {
            int x, a, v
            ...
            c          <----------  access to c
        }
        ... r ... s
    }
   ... q ...
}
```

Establishing Addressability

Using access links

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



Some setup cost on each call

- 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** $\Rightarrow$ r_2
<1,12>	**loadAI** r_0, **-4** $\Rightarrow$ r_1 **loadAI** r_1, **12** $\Rightarrow$ r_2
<0,16>	**loadAI** r_0, **-4** $\Rightarrow$ r_1 **loadAI** r_1, **-4** $\Rightarrow$ r_1 **loadAI** r_1, **16** $\Rightarrow$ 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*
  → <u>Find ARP for *j* –1</u>
  → <u>Use that</u> ARP as link

Access & maintenance cost varies with level

All accesses are relative to ARP (r_0)

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

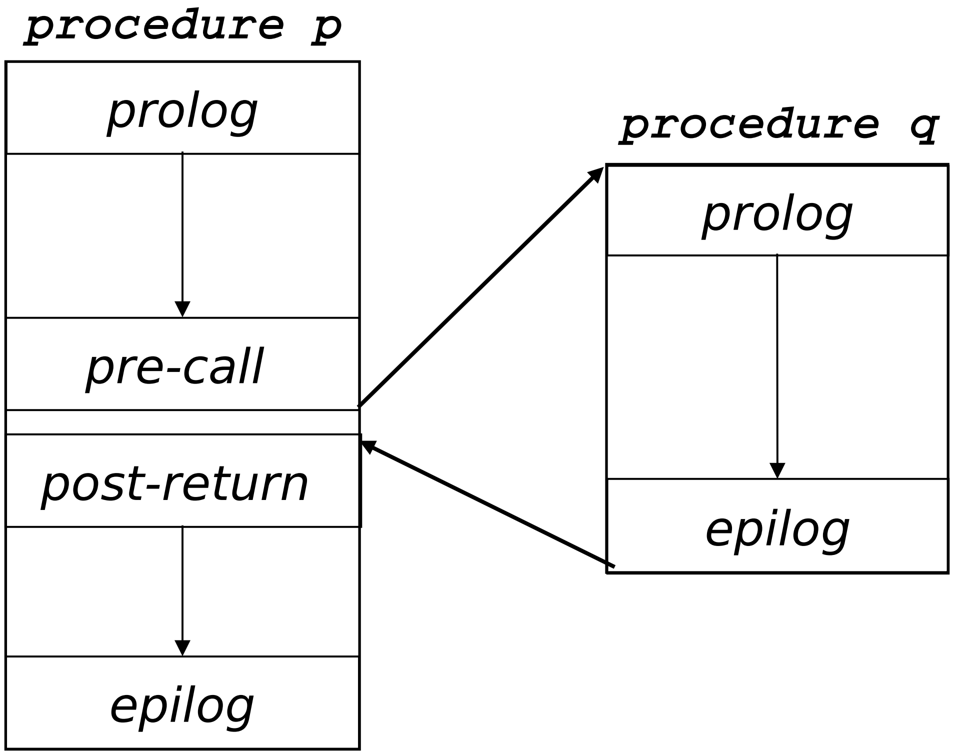
Compiler must use a standard sequence of operations

- 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 the callee's environment
- Preserve parts of the caller's environment that will be disturbed

The Details

- Preserve any callee-save registers
- 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

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

The Details

- Store return value? No, this happens on the return statement
- 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