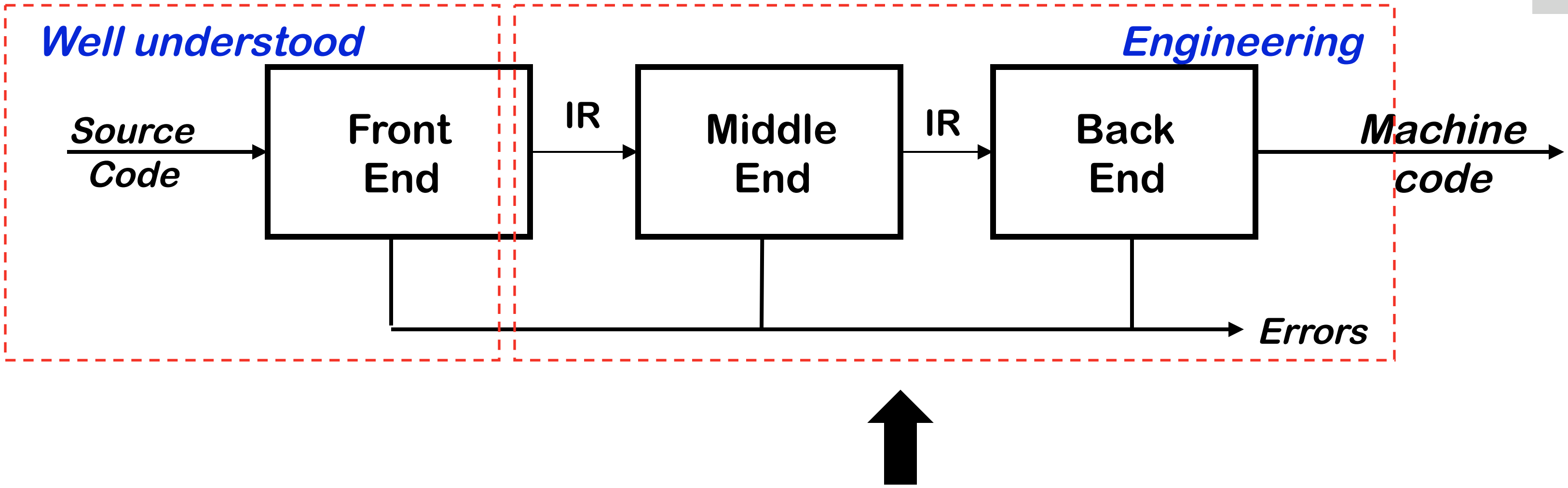
The Procedure Abstraction
Part I: Basics

Procedure Abstraction

- Begins Chapter 6 in EAC
- The compiler must deal with interface between compile time and run time
  - → Most of the tricky issues arise in implementing "procedures"
- Issues
  - → Compile-time versus run-time behavior
  - → Finding storage for EVERYTHING and mapping names to addresses
  - → Generating code to compute addresses
  - → Interfaces with other programs, other languages, and the OS
  - → Efficiency of implementation

Where are we?



Contains more open problems and more challenges

- This is "compilation," as opposed to "parsing" or "translation"
- Implementing promised behavior
  - 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

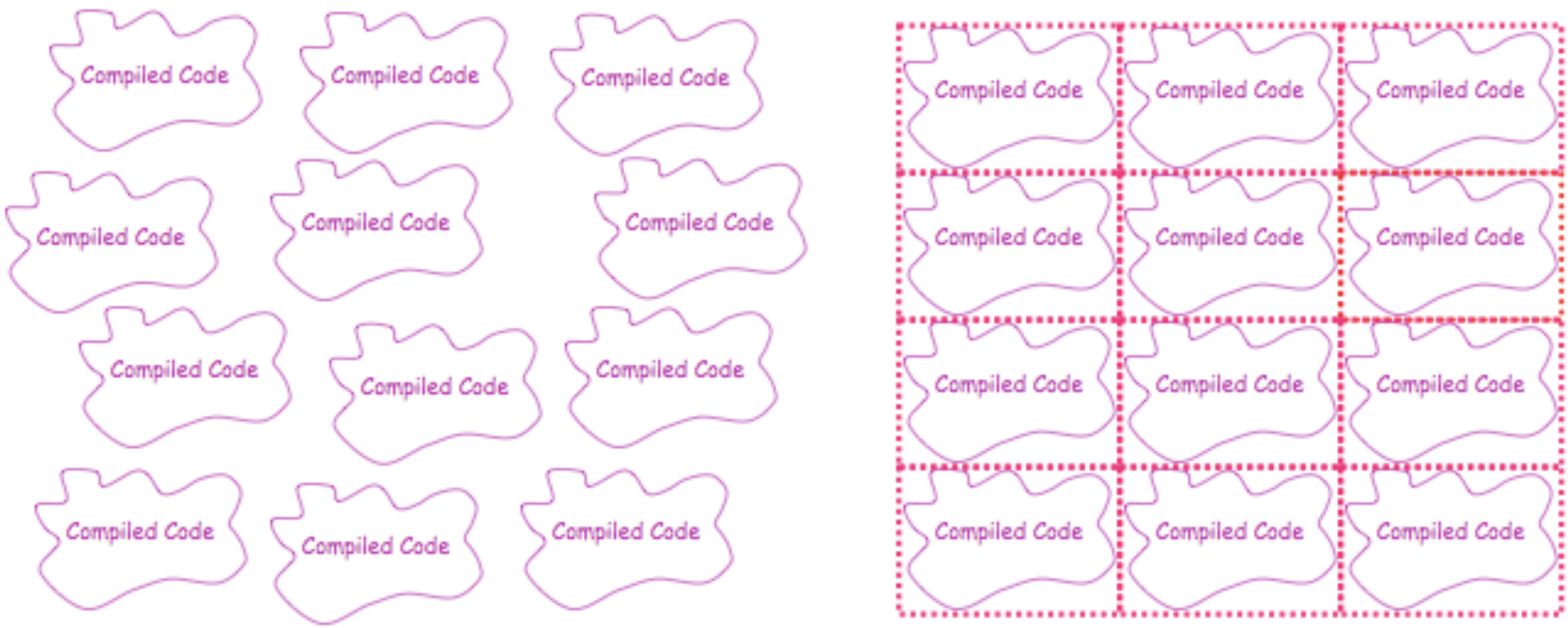
The Procedure & Its Three Abstractions

The compiler produces code for each procedure

The individual code bodies must fit together to form a working program

The Procedure as a Name Space

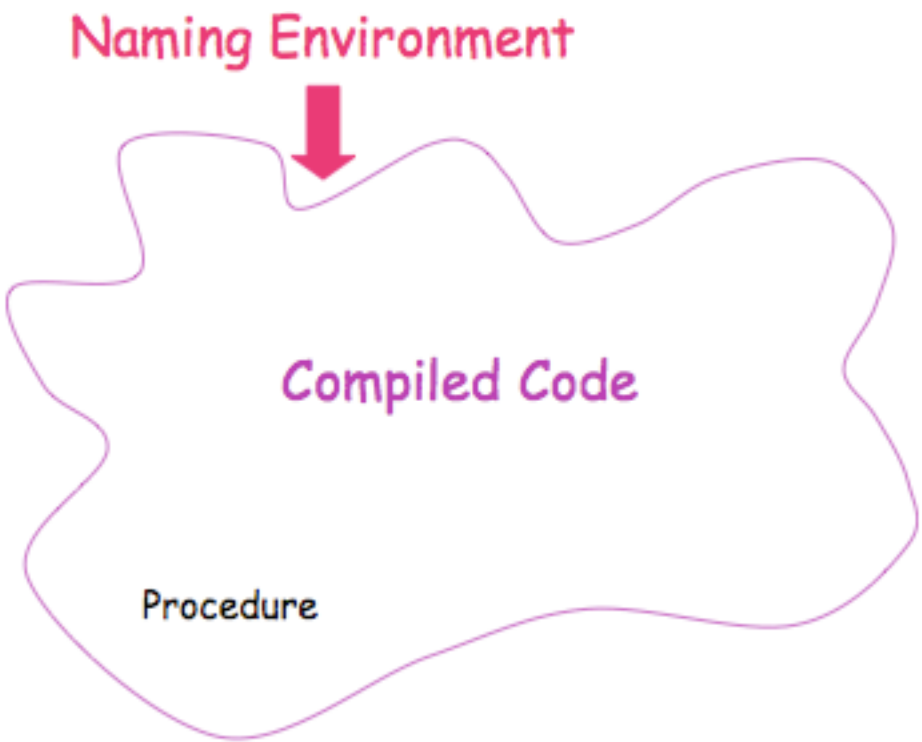
In essence, the procedure linkage wraps around the unique code of each procedure to give it a uniform interface



Similar to building a brick wall rather than a rock wall

There is a strict constraints that each procedure must adhere to!

The Procedure & Its Three Abstractions



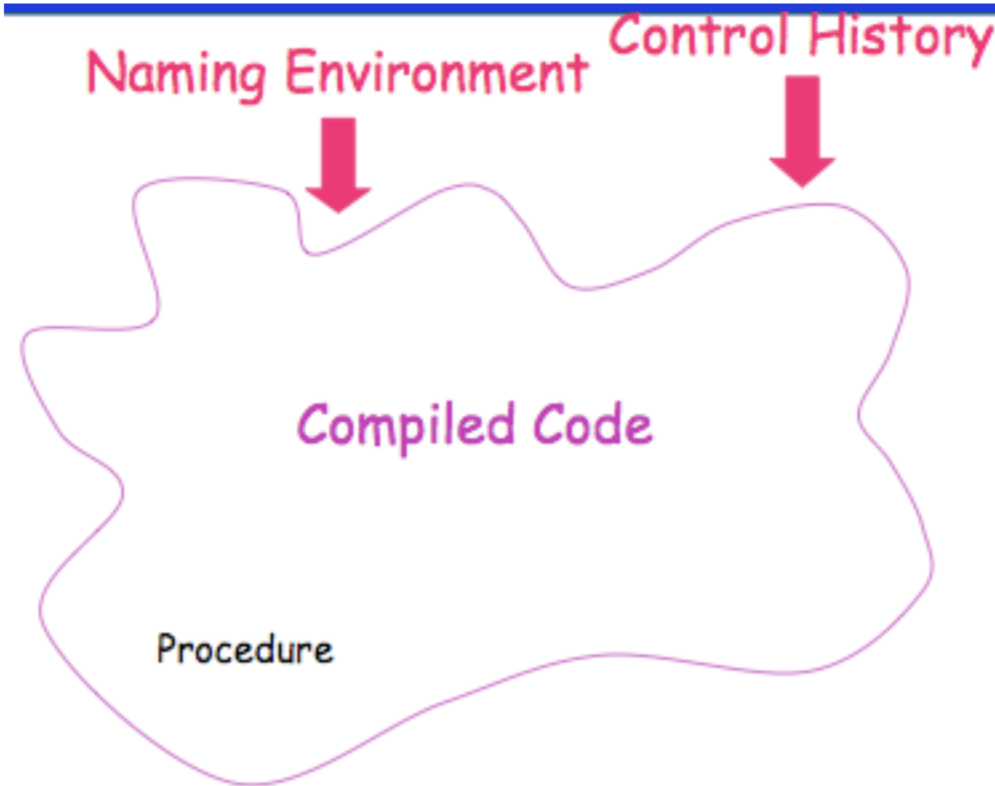
"Naming" includes the ability to find and access the object in memory

Each procedure inherits a set of names

⇒ Variables, values, procedures, objects, locations, …

⇒ Clean slate for new names, "scoping" can hide other names

The Procedure & Its Three Abstractions

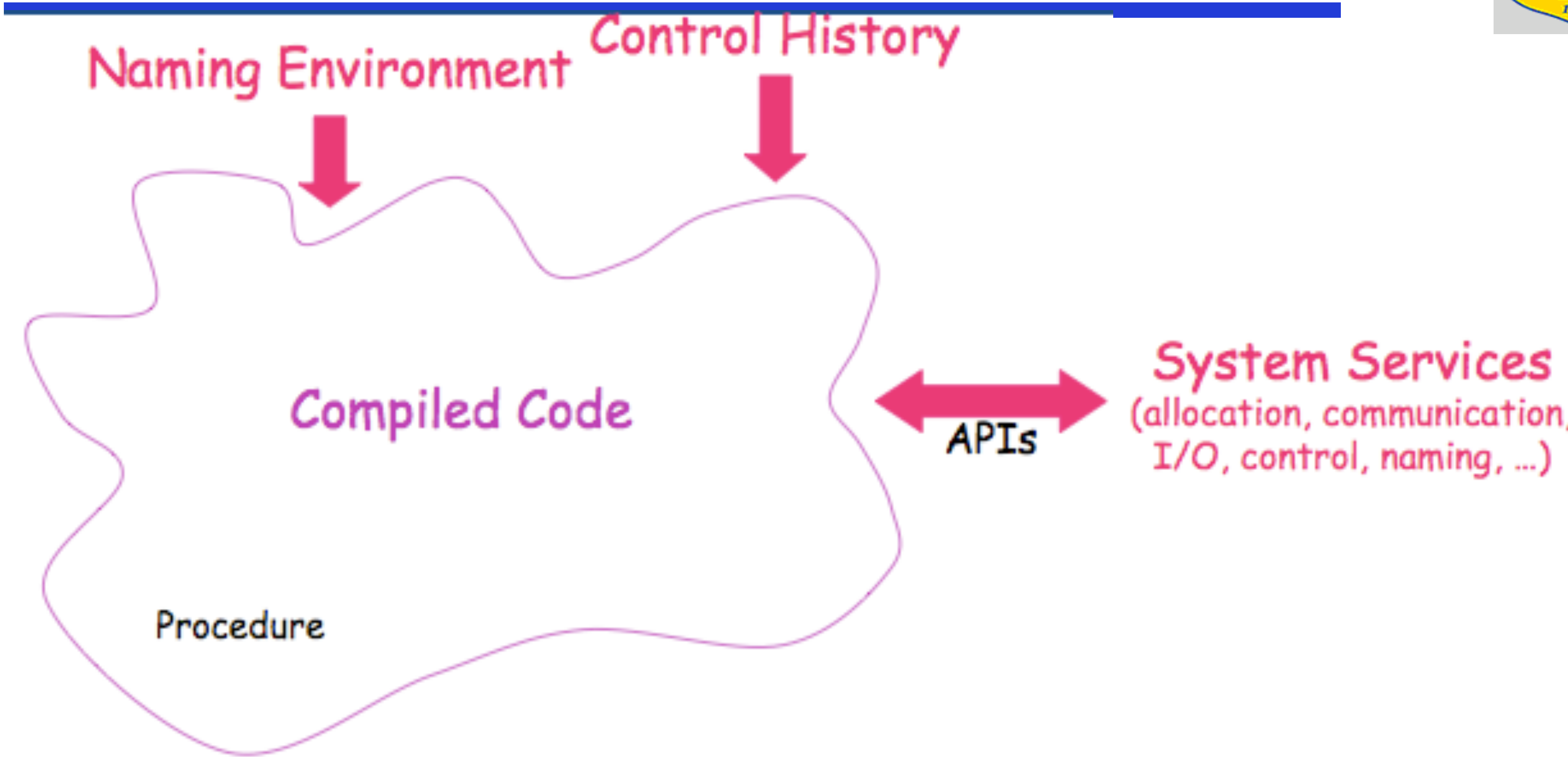


Each procedure inherits a control history

⇒ Chain of calls that led to its invocation

⇒ Mechanism to return control to caller

The Procedure & Its Three Abstractions



Each procedure has access to external interfaces

⇒ Access by name, with parameters *(may include dynamic link & load)*

⇒ Protection for both sides of the interface

The Procedure: Three Abstractions

- Control Abstraction
  - → Well defined entries & exits
  - → Mechanism to return control to caller
- 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, linker 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

Compile time versus run time can be confusing to CISC672 students. We will emphasize the distinction between them.

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);
   ...
}
```

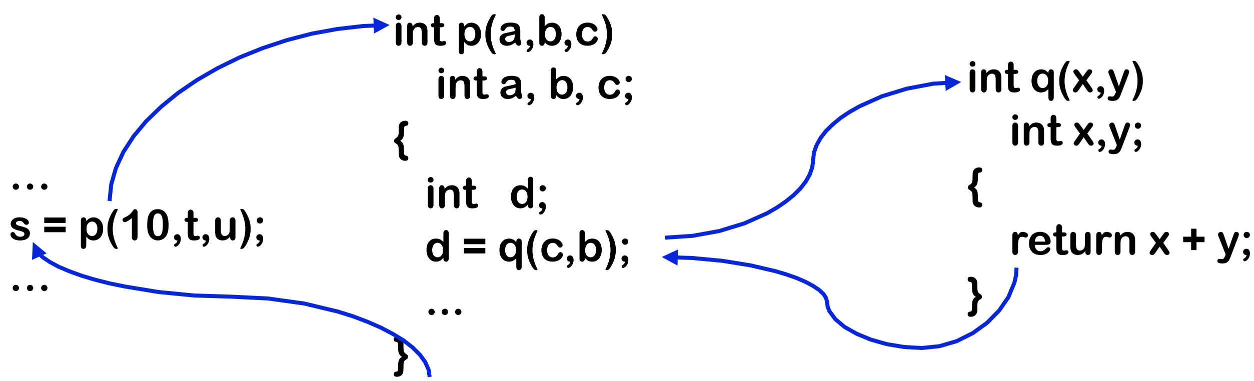
```
int q(x,y)
   int x,y;
{
   return x + y;
}
```

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 behavior

- 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);
   ...
}
```

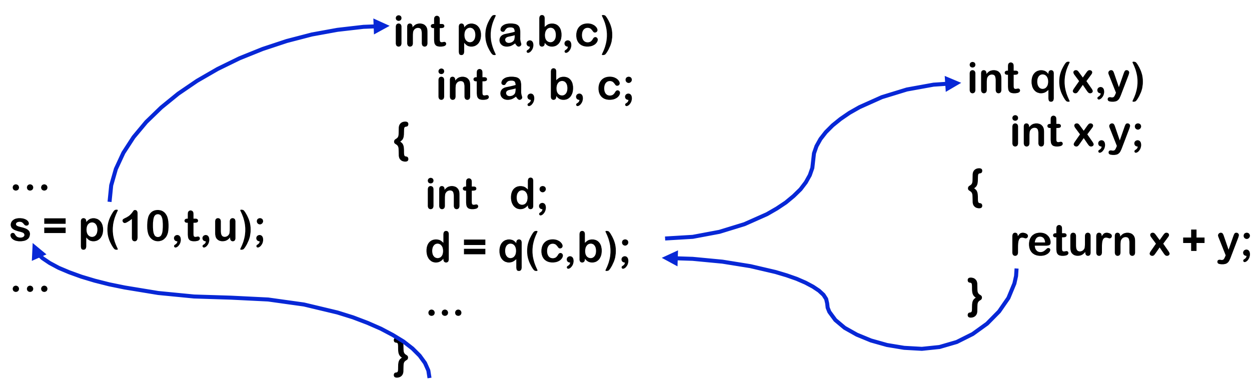
```
int q(x,y)
   int x,y;
{
   return x + y;
}
```

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

The Procedure as a Control Abstraction

Implementing procedures with this behavior

- Must preserve *p*'s state while *q* executes
- *Strategy*: Create unique location for each procedure activation
  - → Can use a "stack" of memory blocks to hold local storage and return addresses



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

The Procedure as a Name Space

Why introduce lexical scoping?

- Provides a compile-time mechanism for binding variables
- Simplifies rules for naming & resolves conflicts
- Lets the programmer introduce "local" names

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

- The compiler must model the name space
- Lexically scoped symbol tables (see § 5.7.3)

Do People Use This Stuff ?

C macro from the MSCP compiler

```
#define fix_inequality(oper, new_opcode)              \
    if (value0 < value1)                              \
    {                                                 \
        Unsigned_Int temp = value0;                   \
        value0 = value1;                              \
        value1 = temp;                                \
        opcode_name = new_opcode;                     \
        temp = oper->arguments[0];                    \
        oper->arguments[0] = oper->arguments[1];      \
        oper->arguments[1] = temp;                    \
        oper->opcode = new_opcode;                    \
    }
```

Even in C, a language not known for abstraction, people do!

Declares a new name

Lexically-scoped Symbol Tables

§ 5.7 in EaC

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 and detailed

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

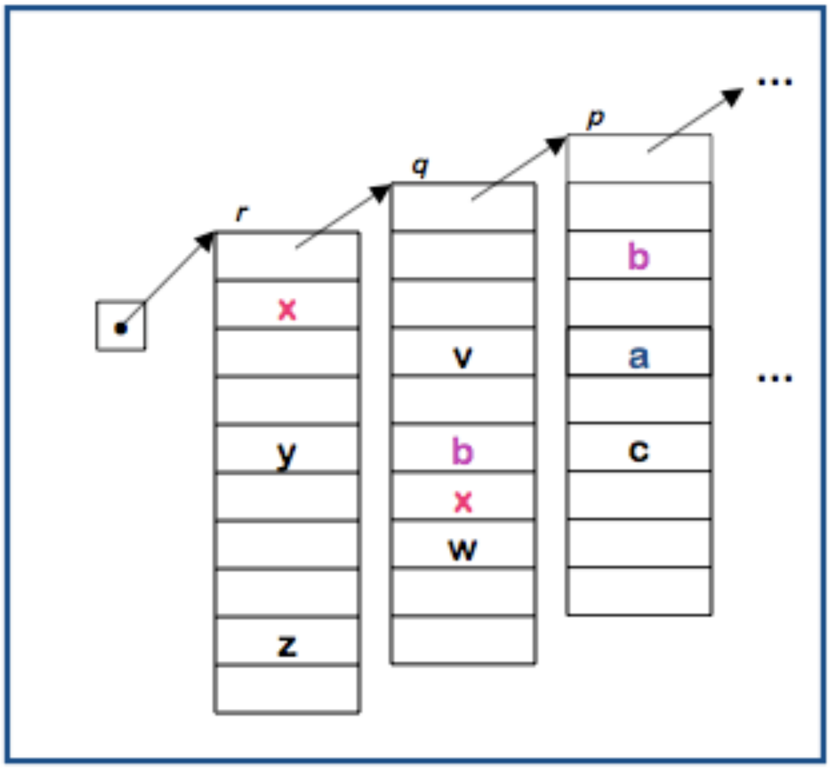
```
procedure p {
        int a, b, c
        procedure q {
          int v, b, x, w
          procedure  r {
                int x, y, z
                ....
          }
          procedure s {
                int x, a, v
                …
          }
          … r … s
        }
    … q …
}
```

```
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



"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.