CS 4240: Compilers

Lecture 15: Procedure Abstraction, Calling Convention

Instructor: Vivek Sarkar (vsarkar@gatech.edu)

March 4, 2019

ANNOUNCEMENTS & REMINDERS

- **Homework 2 due by 11:59pm TODAY**
  - 5% of course grade
- Project 2 released on Feb 27th
  - Due by 11:59pm on Wednesday, April 3rd
  - 15% of course grade
- **MIDTERM EXAM: Wednesday, March 13, 4:30pm - 5:45pm**
  - 20% of course grade
  - Scope of exam: Lectures 1-8, 10-14 (Lecture 9 on MIPS processor is excluded)
  - Chapters 5 and 8-13 of textbook (restricted to sections covered in class)
  - **March 6th and March 11th lectures will review midterm material**
  - **Practice midterm will be released on March 6th**
- FINAL EXAM: Wednesday, May 1, 2:40 pm - 5:30 pm
  - 30% of course grade

Worksheet-14 Solution

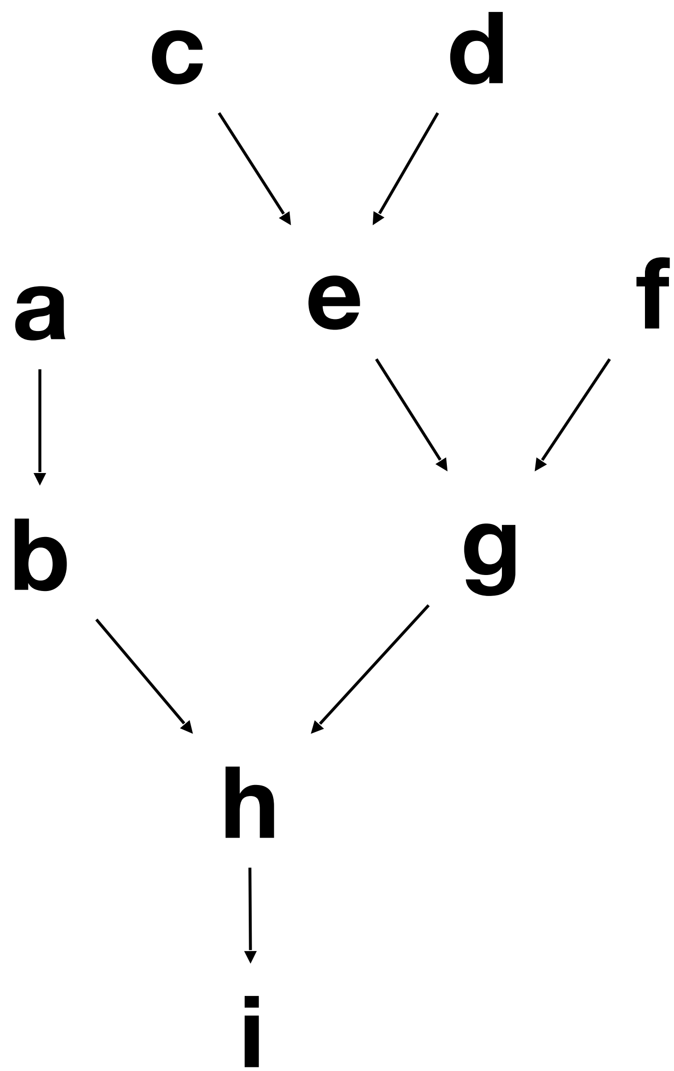
From lecture given on 02/27/2019

A simple schedule for **w ← x * 2 + 19 * y * z** is included below on the left, along with operation latencies. Produce an optimized schedule with the minimum completion time, using as many registers as you choose.

a	loadAI	r0, @x → r1
b	add	r1, r1 → r1
c	loadI	19 → r2
d	loadAI	r0, @y → r3
e	mult	r2, r3 → r2
f	loadAI	r0, @z → r3
g	mult	r2, r3 → r2
h	add	r1, r2 → r1
i	storeAI	r1 → r0,@w

Instruction	Cycles	Meaning
loadAI rx, c ⇒ rz	3	MEM(rx + c) → rz
storeAI rx ⇒ ry, c	3	rx → MEM(ry + cz)
loadI c ⇒ rx	1	c → rx
add rx, ry ⇒ rz	1	rx + ry → rz
mult rx, ry ⇒ rz	2	rx * ry → rz
Shift rx, c ⇒ ry	1	(rx << c) → ry

Dependence Graph




```
a  loadAI   r0, @x → r1
b  add      r1, r1 → r1
c  loadI    19 → r2
d  loadAI   r0, @y → r3
e  mult     r2, r3 → r2
f  loadAI   r0, @z → r3
g  mult     r2, r3 → r2
h  add      r1, r2 → r1
i  storeAI  r1 → r0,@w
```

Instruction	Cycles	Meaning
loadAI rx, c ⇒ rz	3	MEM(rx + c) → rz
storeAI rx ⇒ ry, c	3	rx → MEM(ry + cz)
loadI c ⇒ rx	1	c → rx
add rx, ry ⇒ rz	1	rx + ry → rz
mult rx, ry ⇒ rz	2	rx * ry → rz
Shift rx, c ⇒ ry	1	(rx << c) → ry

d: loadAI	r0, @y → r1	latency 3
f: loadAI	r0, @z → r2	latency 3
c: loadI	19 → r3	latency 1
a: loadAI	r0, @x → r4	latency 3
e: mult	r1, r3 → r5	latency 2
NO-OP (due to use of r5 in g)		
g: mult	r2, r5 → r6	latency 2
b: add	r4, r4 → r7	latency 1
h: add	r6, r7 → r8	latency 1
i: storeAI	r8 → r0,@w	latency 3
NO-OP		
NO-OP		

The shortest possible schedule(assuming at most 1 instruction is issued per cycle) is 12 cycles. Attempt to re-order instructions to remove the NO-OP in the middle of the code may remove the NO-OP itself, but results in a new NO-OP due to other data dependencies.

Procedural Abstraction: Conceptual Overview

Procedures provide the fundamental abstractions that make programming useful and practical

- Information hiding
- Distinct and separable name spaces
- Uniform interfaces

Hardware does little to support these abstractions

- Part of the compiler's job is to implement them
  - ♦ Compiler makes good on lies that we tell programmers
- Part of the compilers job is to make it efficient
  - ♦ Role of code optimization

Practical Overview

The compiler must decide almost everything

- Location for each value (named and unnamed)
- Method for computing each result
  - For example, how should it compute y^x or a case statement?
- Compile-time versus runtime behavior
- How to locate objects & values created & manipulated by code that the compiler cannot see? (other files, libraries)

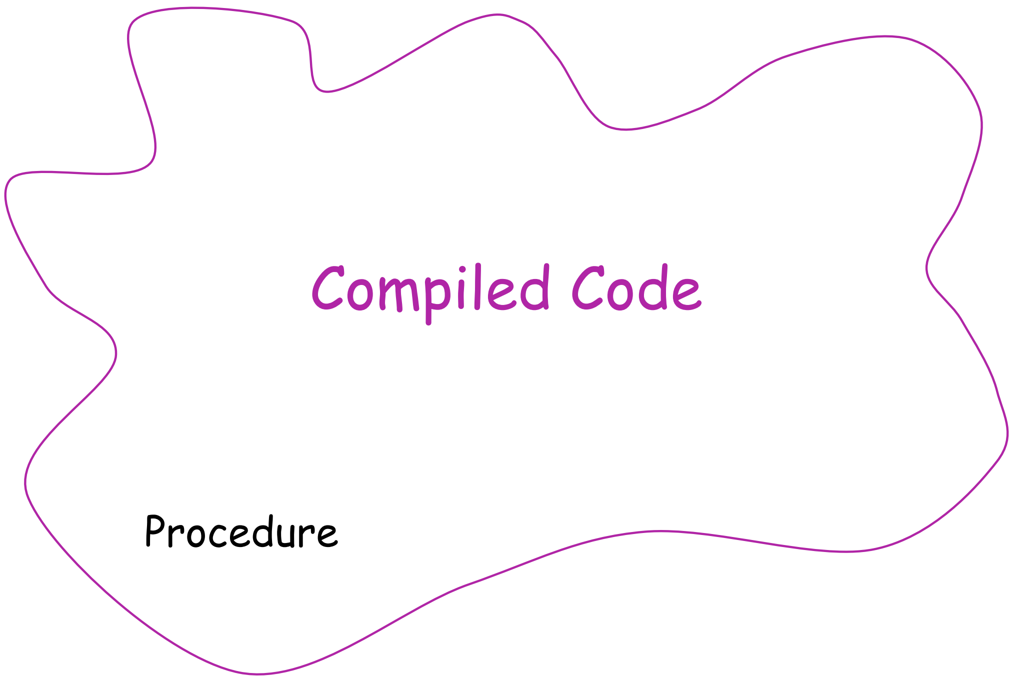
All of these issues come together in the implementation of procedures

Pay close attention to compile-time versus runtime

- Confuses students more than any other issue

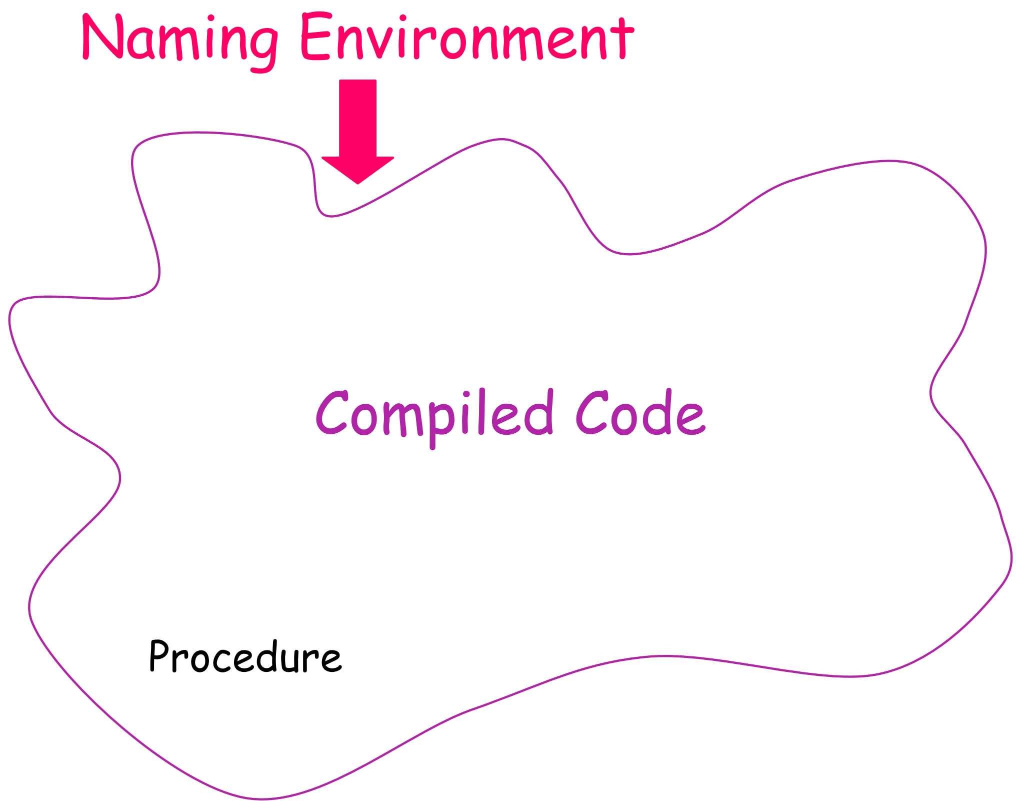
The Procedure & Its Three Abstractions

The compiler produces code for each procedure



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

The Procedure & Its Three Abstractions



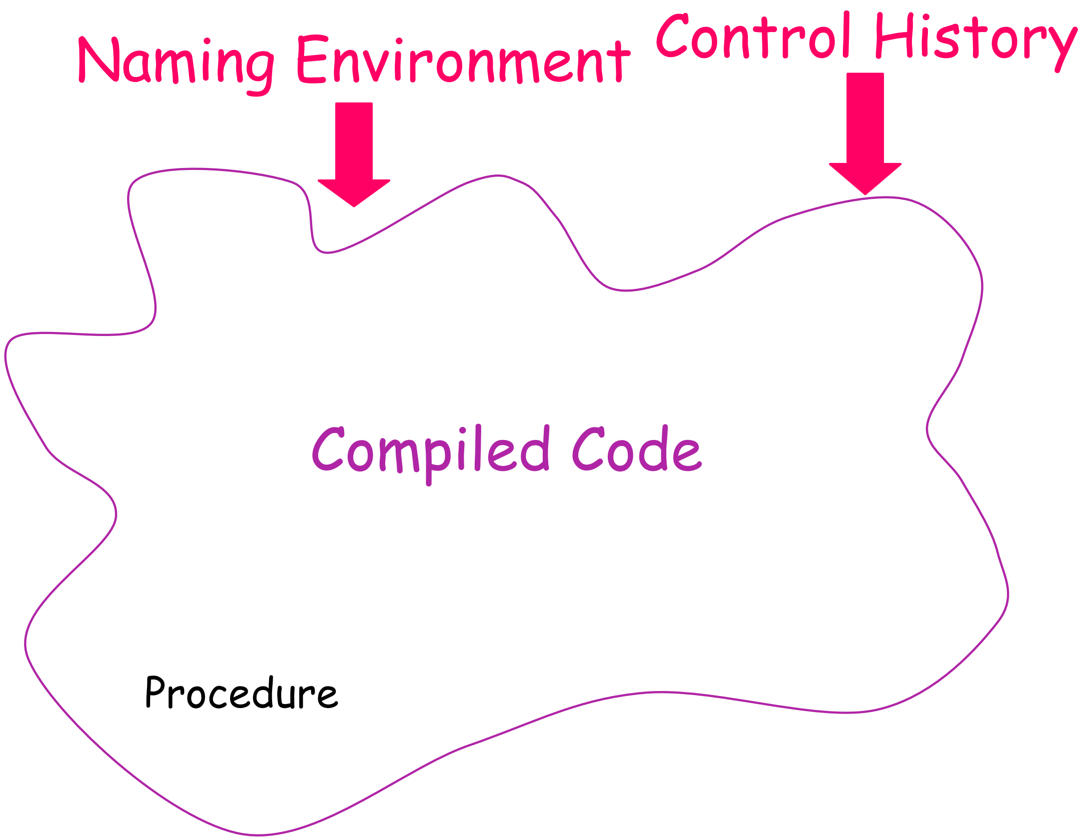
"Naming" includes the ability to find and access objects in memory

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



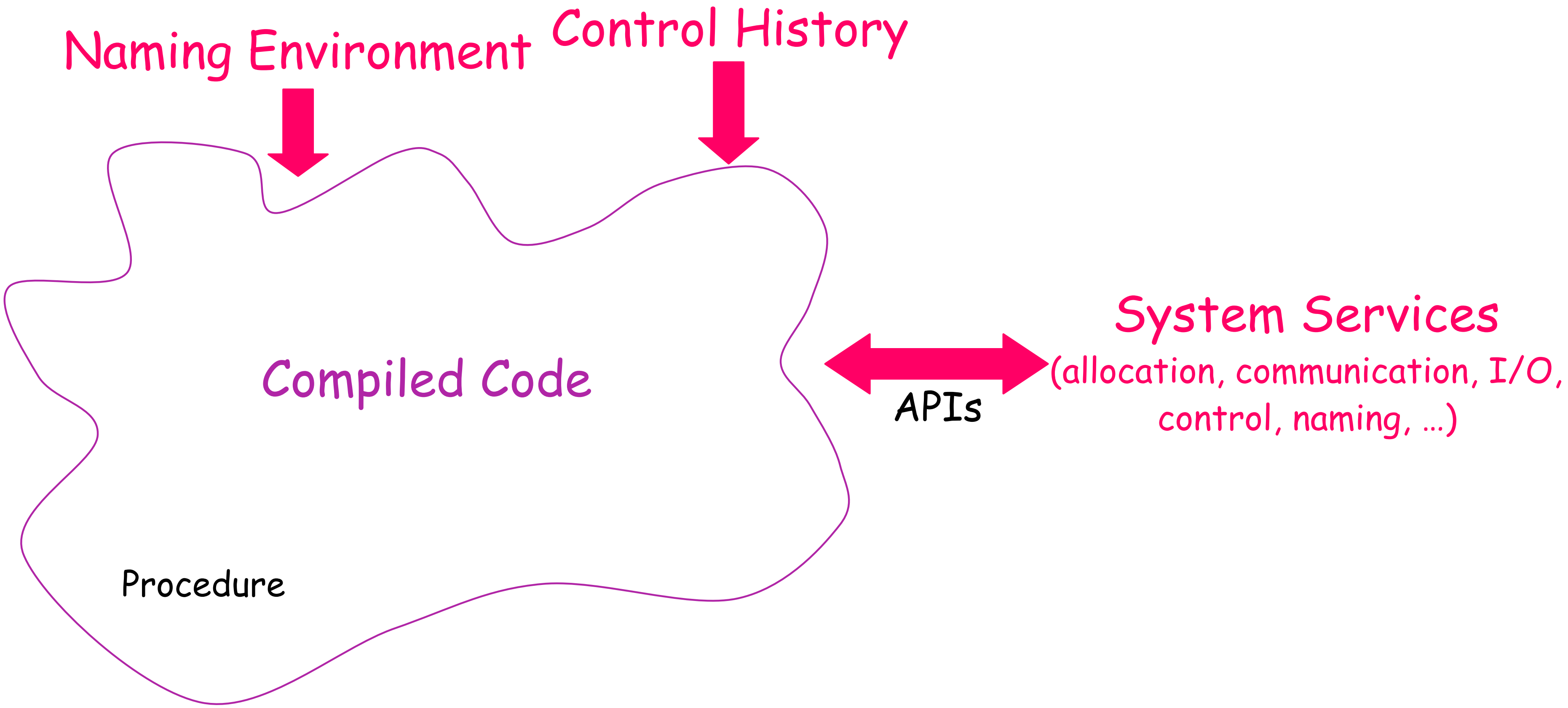
Each procedure inherits a control history

⇒Chain of calls that led to its invocation

⇒Mechanism to return control to caller

Some notion of parameterization (ties back to naming)

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
  - ♦Some notion of parameterization (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 compact
  - 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 (or calling 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

The compiler's job is to make good on the lies told by the programming language design!

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;
{
   if ( … )
     x = q(x-1,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→x, b→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;
{
   if ( ... )
     x = q(x-1,y);
   return x + y;
}
```

Compiler emits 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
  - ♦ recursion causes the real problem here
- Strategy: Create unique location for each procedure activation
  - ♦ In simple situations, can use a “stack” of memory blocks to hold local storage and return addresses (closures ⇒ heap allocate)

```
…
s = p(10,t,u);
…
```

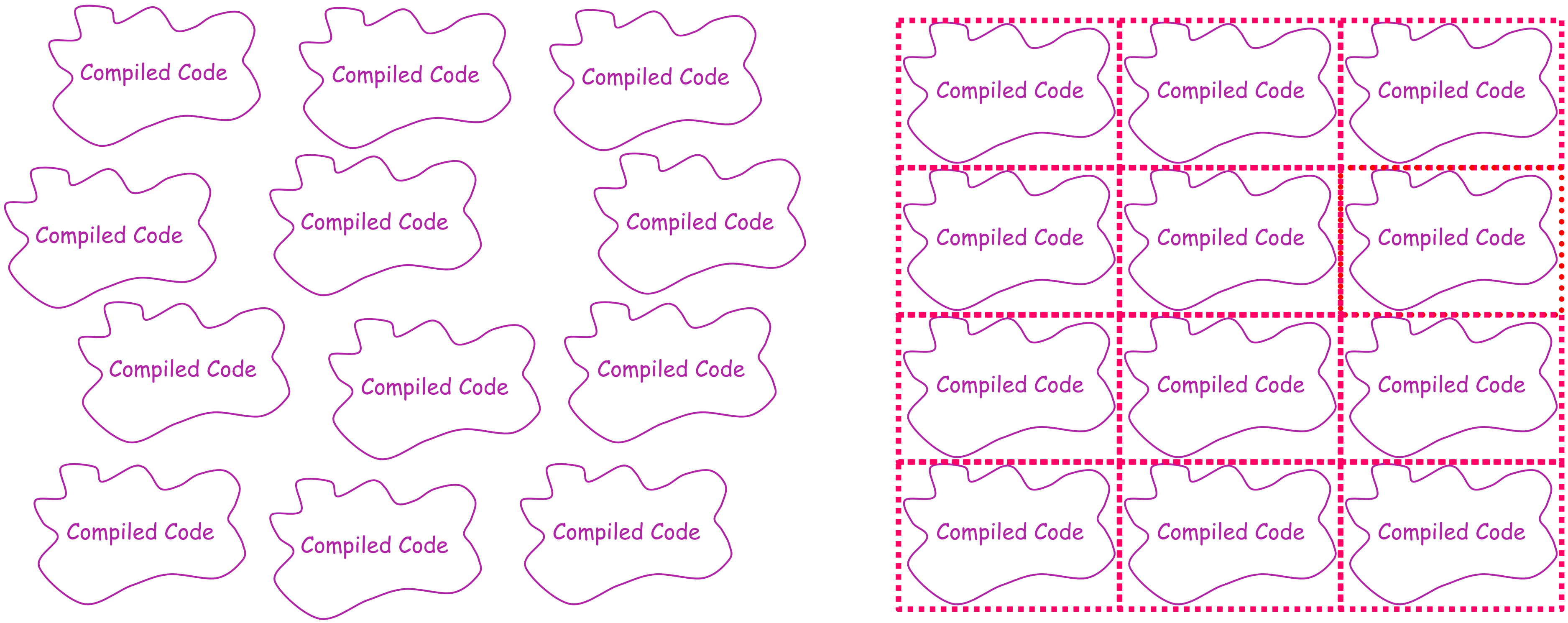
```
int p(a,b,c)
   int a, b, c;
{
   int  d;
   d = q(c,b);
   …
}
```

```
int q(x,y)
   int x,y;
{
   if ( … )
     x = q(x-1,y);
   return x + y;
}
```

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

The Procedure as a Control Abstraction

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

Where Do All The Variables Go?

Automatic & Local

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

Static

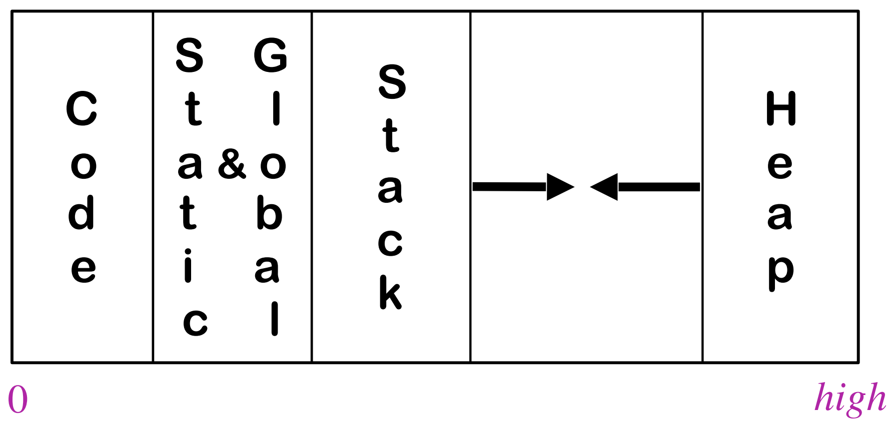
- Procedure scope ⇒ storage area affixed with procedure name
- File scope ⇒ storage area affixed with file name
- Lifetime is entire execution

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 <u>virtual</u> address space