RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers

Procedure Abstractions

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

Announcements

- Homework 9 posted
- Project 3: Has been posted; due on April 27
- Final exam: May 6 (conflict?)
- Homework 10 will be posted next week; will not be graded, so only sample solution

Review: Data-flow Analysis

Definition

Data-flow analysis is a collection of techniques for *compile-time* reasoning about the *run-time* flow of values

- We use the results of DFA to prove safety & identify opportunities
- Almost always involves building a graph
  - → Control-flow graph, call graph, or derivatives thereof
  - → Sparse evaluation graphs to model flow of values *(efficiency)*
- Desired result is usually *meet over all paths* solution
  - → "What is true on every path from the entry?"
  - → "Can this happen on any path from the entry?"

Review: Information Chains

A tuple that connects 2 data-flow events is a *chain*

event ≅ definition or use

- Chains express data-flow relationships directly
- Chains provide a graphical representation
- Chains jump across unrelated code, simplifying search

We can build chains efficiently

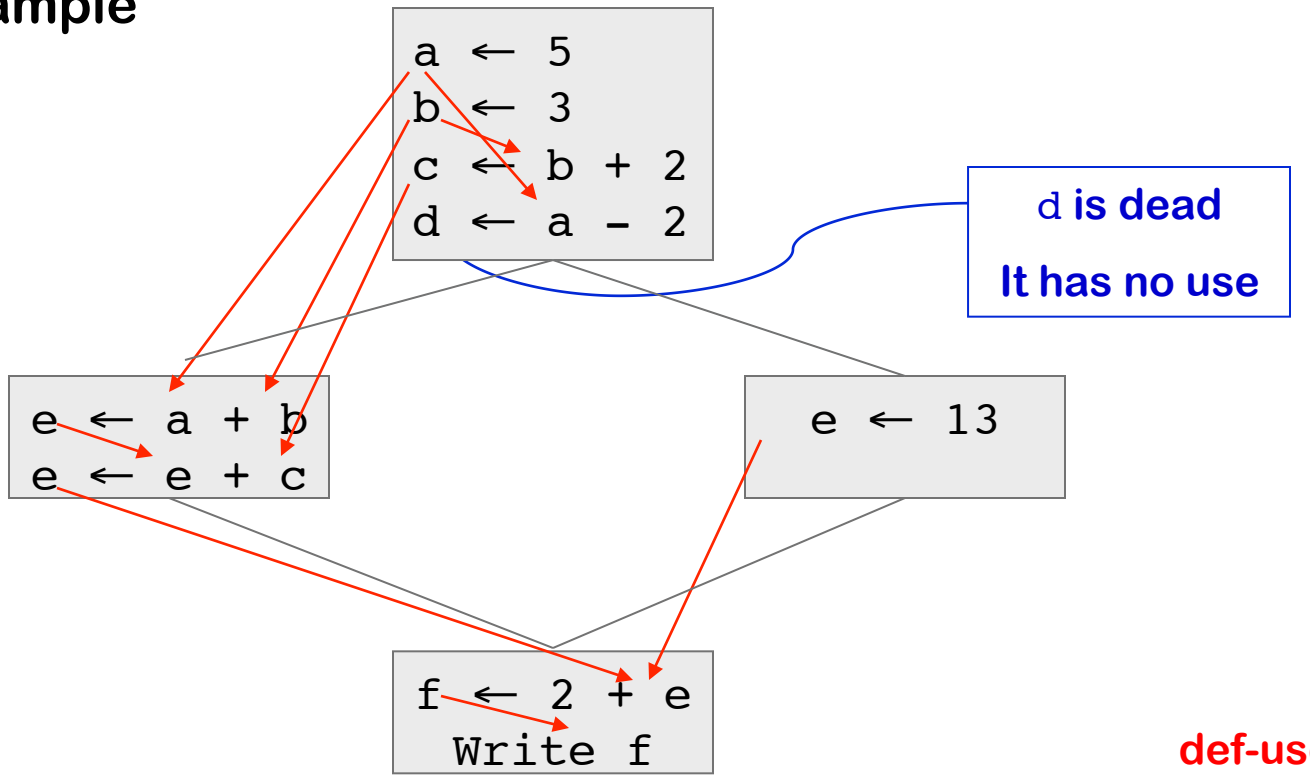
Four interesting types of chain

Def-Use chains are the most common

Source	Sink	Dependence Type
def	use	true, flow
use	def	anti
def	def	output
use	use	input

Review: Information Chains

Example



def-use chains

Review: Notation

Assume that, ∀ operation *i* and each variable *v*,

- *DEFS(v,i)* is the set of operations that may have defined *v* most recently before *i*, along some path in the CFG
- *USES(v,i)* is the set of operations that may use the value of *v* computed at *i*, along some path in the CFG

$$x \in DEFS(A,y) \Leftrightarrow y \in USES(A,x)$$

To construct DEF-USE chains, we solve *reaching definitions* (*YADFP*)

- A definition *d* of some variable *v* *reaches* an operation *i* if and only if *i* reads *v* and there is a *v-clear* path from *d* to *i*
  - → *v-clear* ⇒ no definition of *v* on the path
- Prior definition of *v* in same block ⇒ $|DEFS(v,i)| = 1$
- No prior definition ⇒ $|DEFS(v,i)| \geq 1$

The chains are non-local in this case

Review: Reaching Definitions

The equations

$$REACHES(n) = \emptyset, \forall\ n \in N$$

$$REACHES(n) = \cup_{p \in preds(n)} (DEDEF(p) \cup (REACHES(p) \cap \overline{DEFKILL(p)}))$$

- *REACHES(n)* is the set of definitions that reach block *n*
- *DEDEF(N)* is the set of definitions in *n* that reach the end of *n*
- *DEFKILL(n)* is the set of defs obscured by a new def in *n*

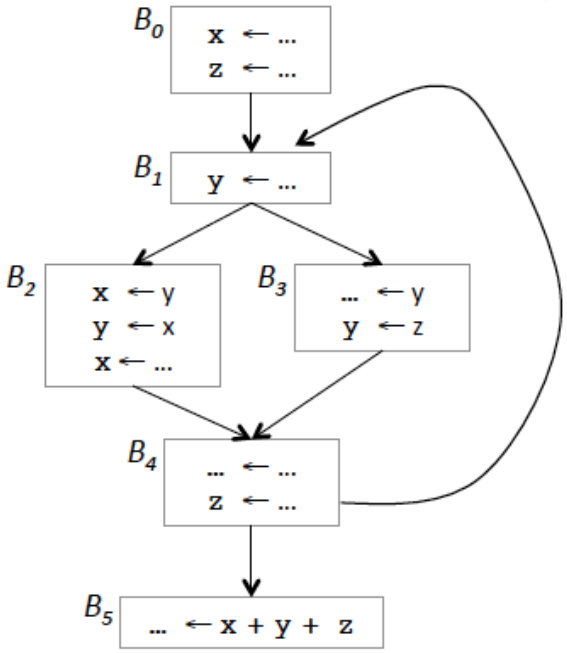
Computing *REACHES(n)*

- Use any data-flow analysis method

Review: Global Register Allocation

Global register allocation requires

- Global analysis, global information
- Local spill code insertion



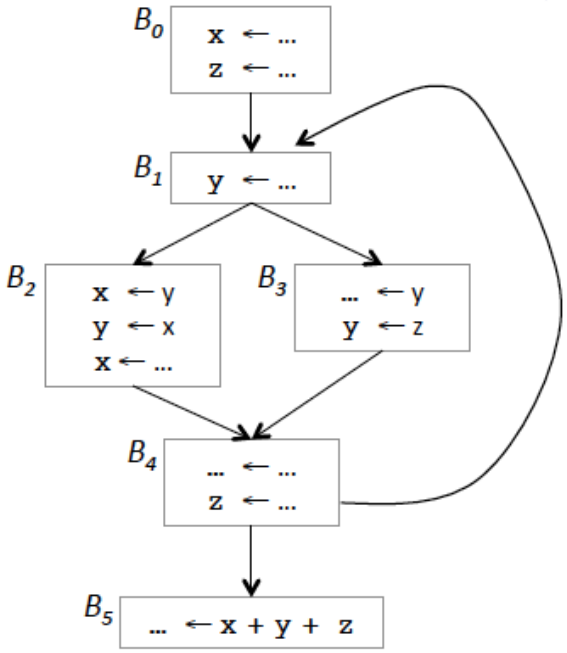
What is the live range of x, y and z?

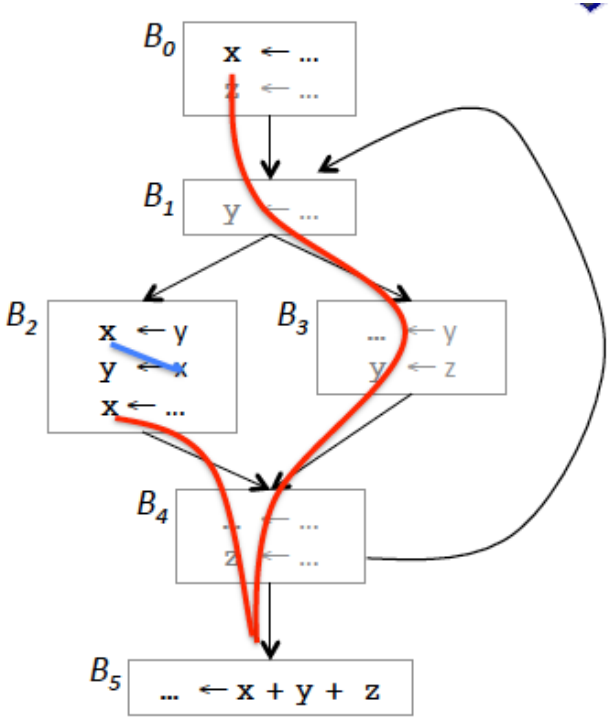
Need to use data flow analysis to determine whether a variable is live after a basic block

Review: Global Register Allocation

Global register allocation requires

- Global analysis, global information
- Local spill code insertion





Review: Static Single Assignment Form (SSA)

- Definition: each name defined exactly once in program
- Introduce ϕ-functions to make it work

Original

```
x ← ...
y ← ...
while (x < k)
   x ← x + 1
   y ← y + x
```

Strengths of SSA-form

- Sharper analysis
- ϕ-functions give hints about placement

Use data flow analysis to determine the definition every use corresponds to

Review: Static Single Assignment Form (SSA)

- Definition: each name defined exactly once in program
- Introduce ϕ-functions to make it work

Original

```
x ← ...
y ← ...
while (x < k)
   x ← x + 1
   y ← y + x
```

SSA-form

```
      x0 ← ...
      y0 ← ...
      if (x0 > k) goto next
loop: x1 ← φ(x0,x2)
      y1 ← φ(y0,y2)
      x2 ← x1 + 1
      y2 ← y1 + x2
      if (x2 < k) goto loop
next:    ...
```

Strengths of SSA-form

- Sharper analysis
- ϕ-functions give hints about placement

Use data flow analysis to determine the definition every use corresponds to

Global Register Allocation

- **A classical stack based coloring approach (Chaitin's algorithm)**

Observation:

Assuming a k-coloring problem, any vertex n that has fewer than k neighbors in the interference graph can always be colored!

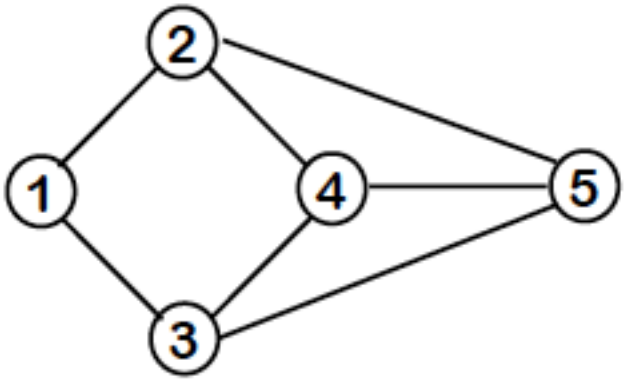
-- Pick any color not used by its neighbors — there must be one

- **Coloring algorithm**
  - → Step 1: pick any vertex n such that deg(n) < k and put it on the stack
  - → Step 2: Remove that vertex and all its incident edges from the interference graph
  - → If there does not exist such vertex, pick one vertex m and spill, remove the vertex m and all its incident edges from the graph. Repeat until we can go back to step 1.

Review: Global Register Allocation

3 Registers

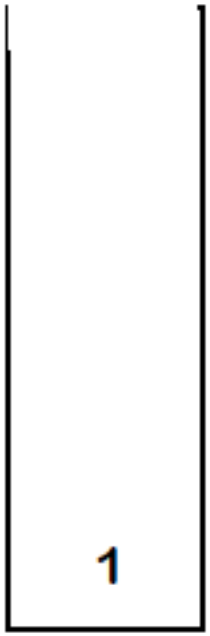
Stack



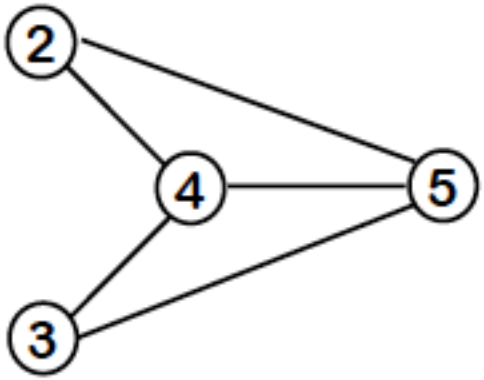
1 is the only node with degree < 3

Review: Global Register Allocation

3 Registers



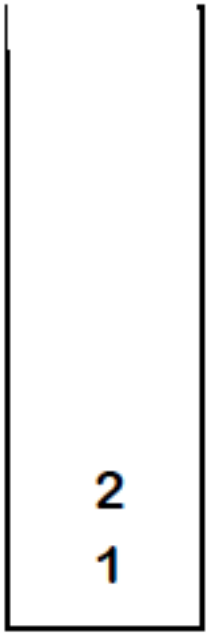
Stack



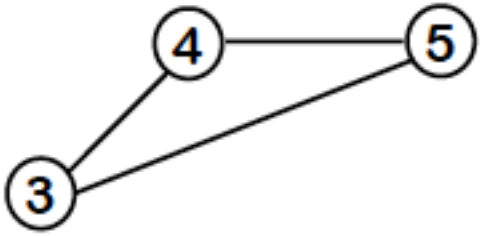
Now, 2 & 3 have degree < 3

Review: Global Register Allocation

3 Registers



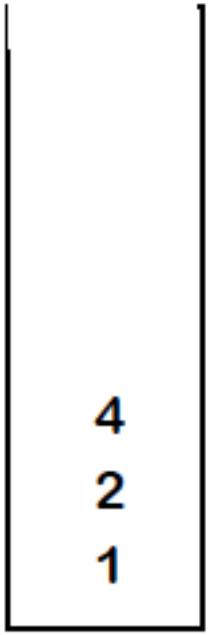
Stack



Now all nodes have degree < 3

Review: Global Register Allocation

3 Registers



Stack

Review: Global Register Allocation

3 Registers

Colors:

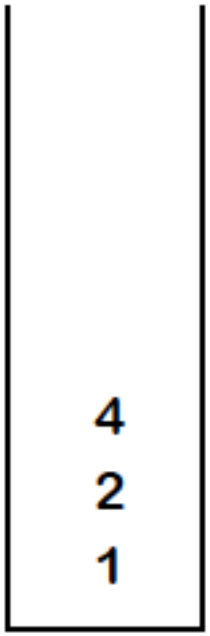
1:

2:

3:

Review: Global Register Allocation

3 Registers



Stack

Colors:

1:

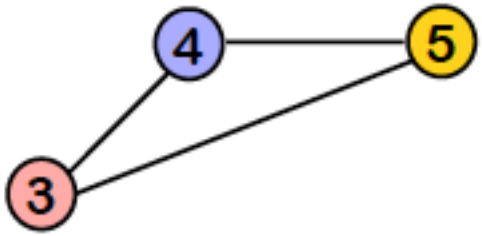
2:

3:

Review: Global Register Allocation

3 Registers

Stack



Colors:

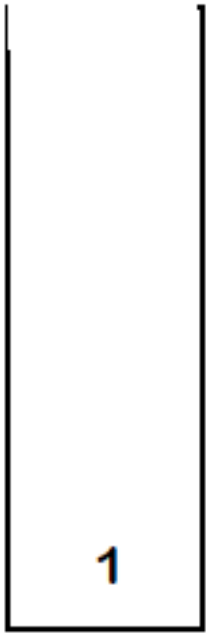
1:

2:

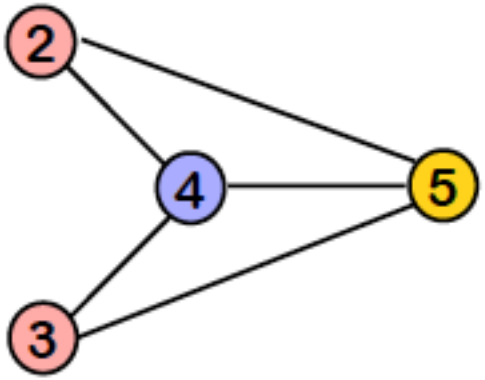
3:

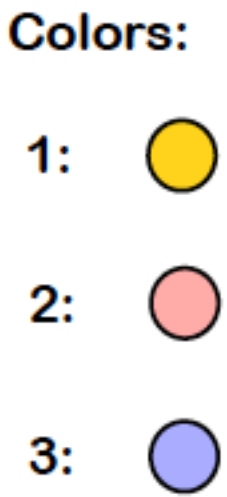
Review: Global Register Allocation

3 Registers



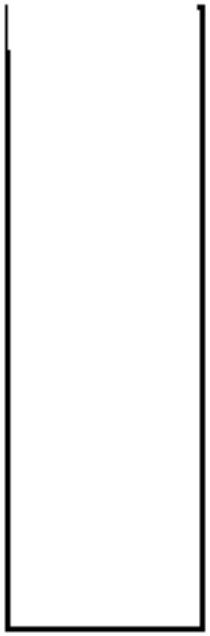
Stack



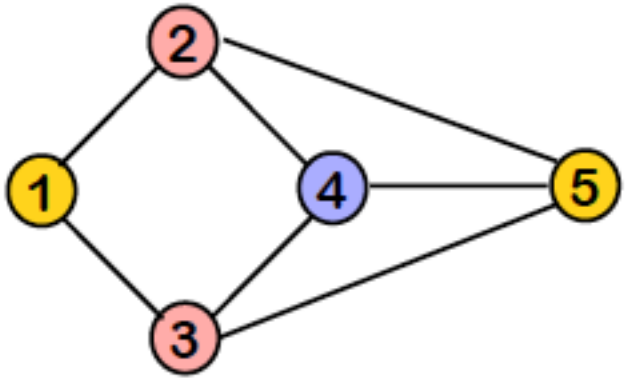


Review: Global Register Allocation

3 Registers



Stack



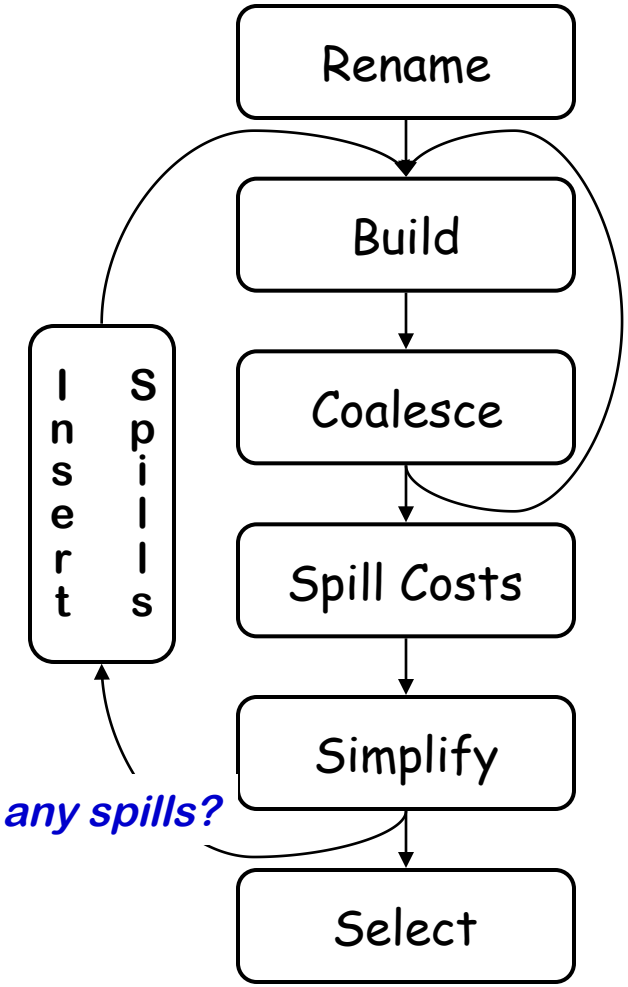
Colors:

1:

2:

3:

Review: Coloring Based Register Allocation



Find live ranges and systematically rename them

Build the interference graph

Coalesce copies ($\langle i,j \rangle \notin G$ & $i \leftarrow j \Rightarrow$ combine i & j)

Estimate cost of spill for each live range (*complex*)

While G is non-empty
if $\exists\ n \ni n^\circ < k$, pull n out of G, & push it on stack
else pick an n to spill & pull it out of G

While stack is non-empty
pop n, insert n into G, assign a color to n

Procedure Abstraction

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

- Procedures permit a critical separation of concerns

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

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

The Procedure as a Control Abstraction

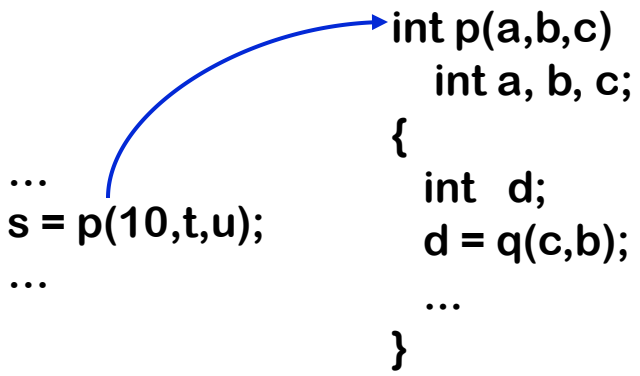
Procedures have well-defined control-flow

- 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

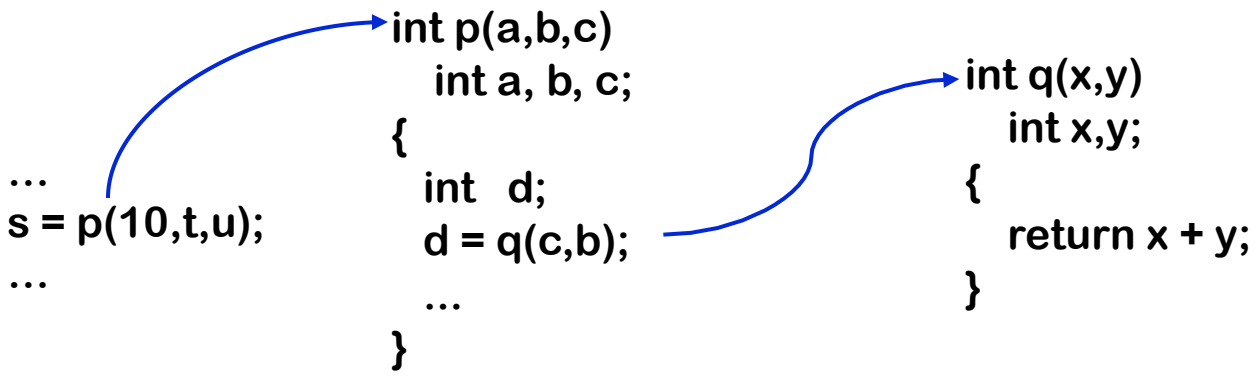
- Invoked at a call site, with some set of *actual parameters*
- Control returns to call site, immediately after invocation



The Procedure as a Control Abstraction

Procedures have well-defined control-flow

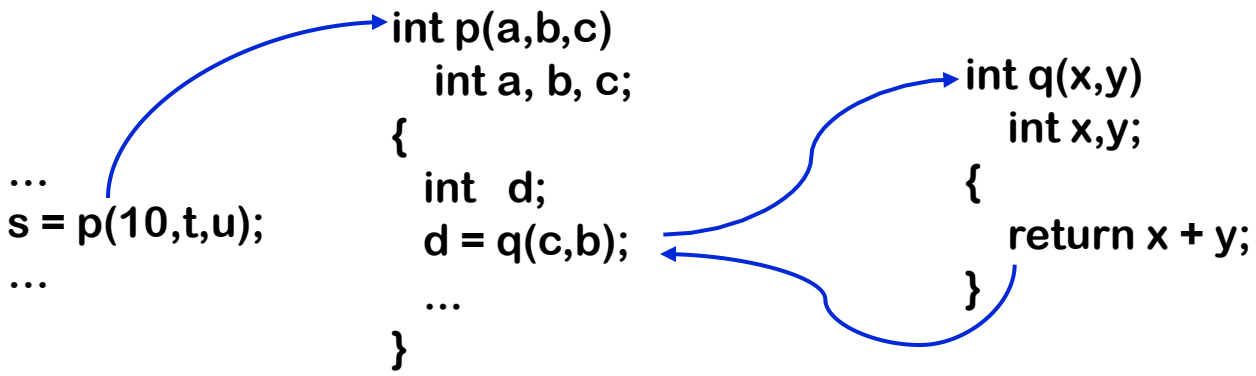
- Invoked at a call site, with some set of *actual parameters*
- Control returns to call site, immediately after invocation



The Procedure as a Control Abstraction

Procedures have well-defined control-flow

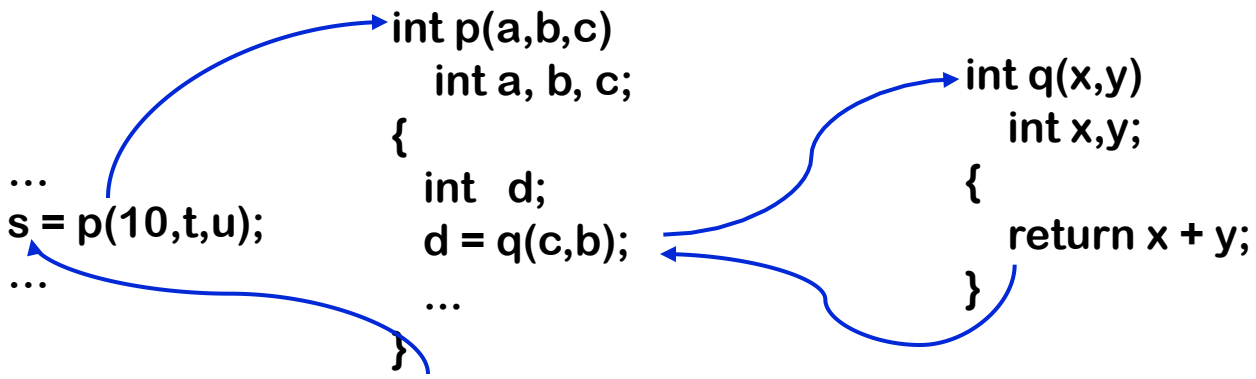
- Invoked at a call site, with some set of *actual parameters*
- Control returns to call site, immediately after invocation



The Procedure as a Control Abstraction

Procedures have well-defined control-flow

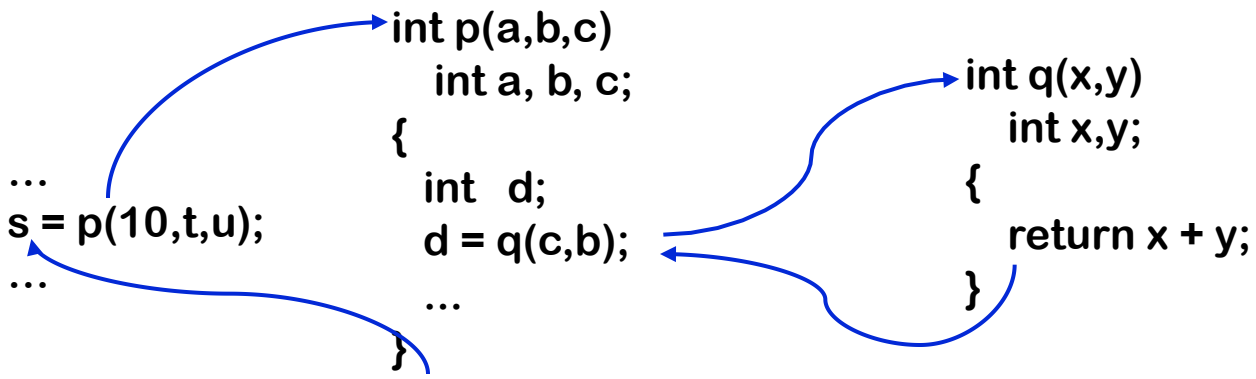
- Invoked at a call site, with some set of *actual parameters*
- Control returns to call site, immediately after invocation



The Procedure as a Control Abstraction

Procedures have well-defined control-flow

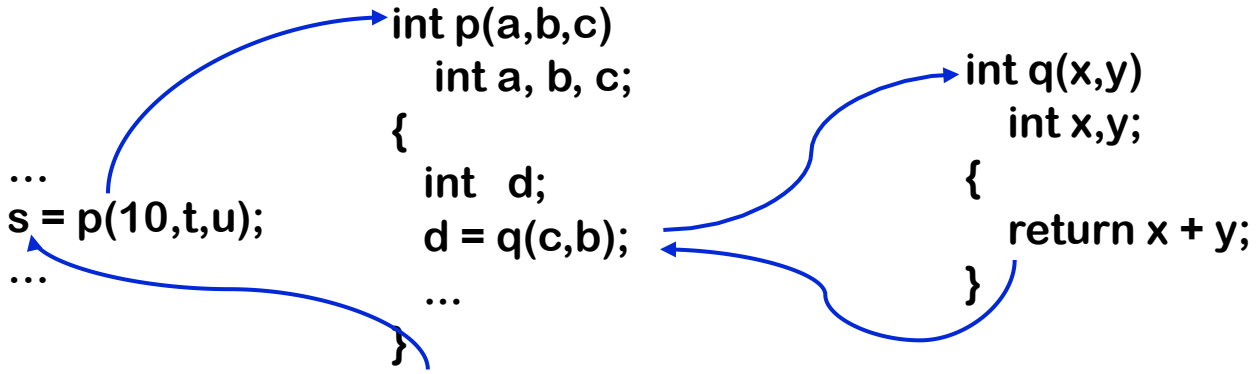
- Invoked at a call site, with some set of *actual parameters*
- Control returns to call site, immediately after invocation



The Procedure as a Control Abstraction

Implementing procedures with this behavior

- Requires code to save and restore a "return address"
- Must map actual parameters to formal parameters q:(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?

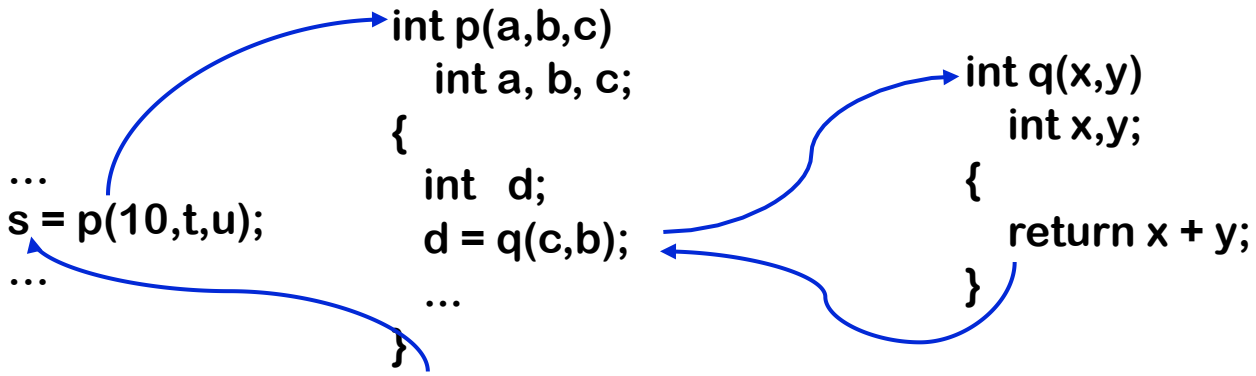


The Procedure as a Control Abstraction

Implementing procedures with this behavior

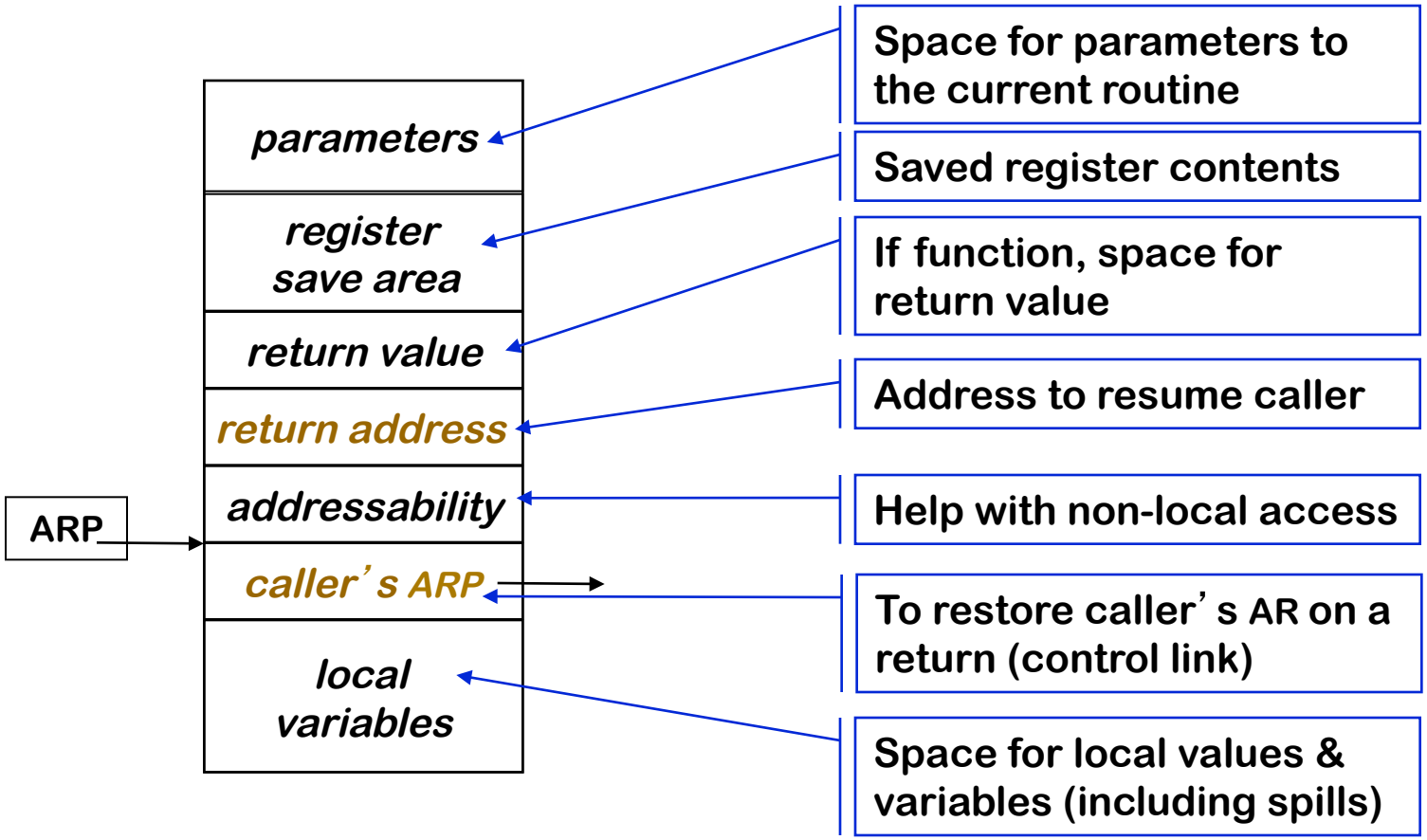
- Must preserve *p*'s state while *q* executes

- *Strategy*: Create unique location for each procedure activation
  - → Can use a "stack" of memory blocks to hold local storage and return addresses



Compiler emits code that causes all this to happen at run time

Activation Record Basics



One AR for each invocation of a procedure

Communicating Between Procedures

Most languages provide a parameter passing mechanism

⇒ Expression used at "call site" becomes a 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 (aliasing)?

e.g: 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
  - → Arrays are mostly passed by reference, not value

- Can always use global variables …

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

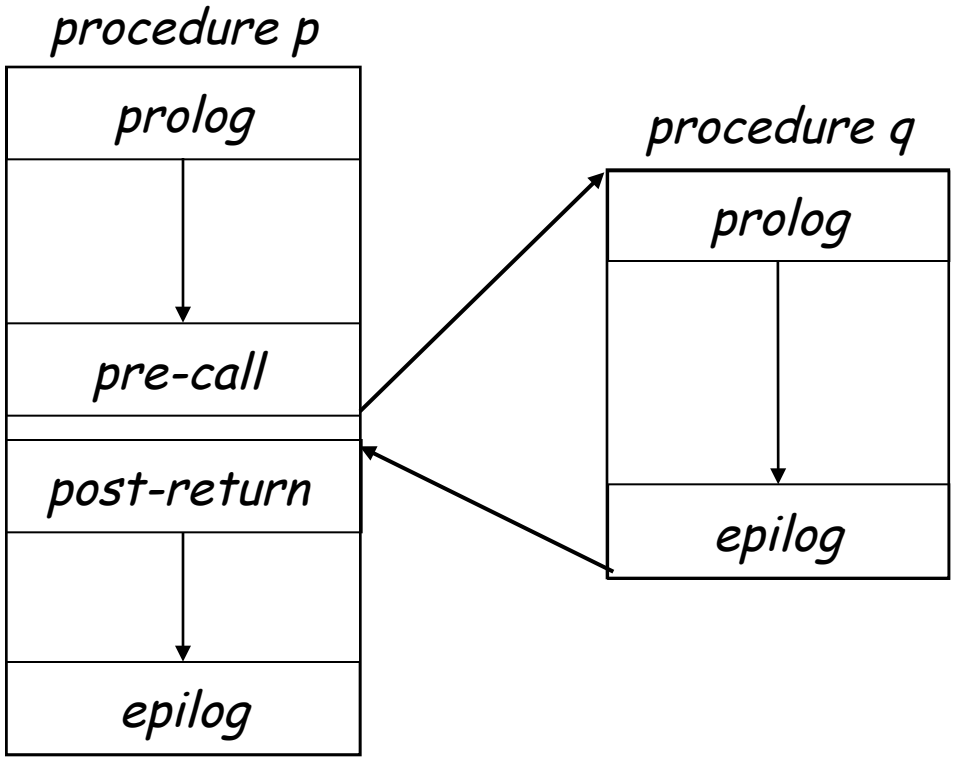
Compiler must use a standard sequence of operations

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

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

Procedure Linkages

Standard procedure linkage

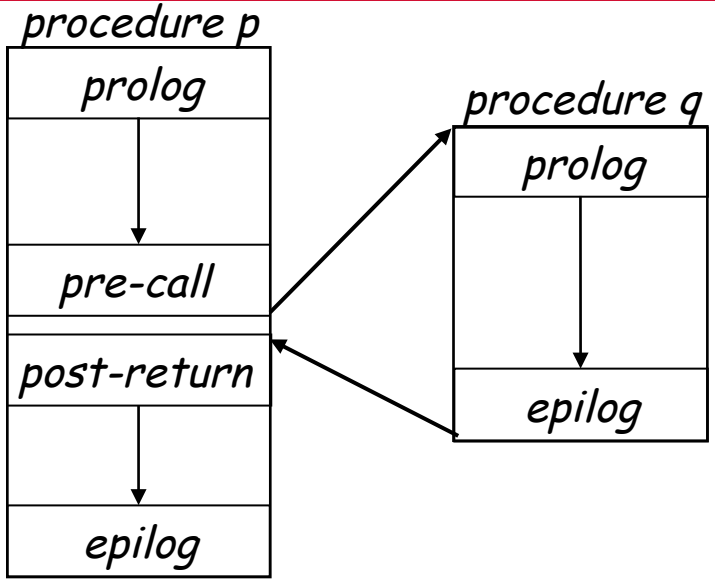


Procedure has
- standard prolog
- standard epilog

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

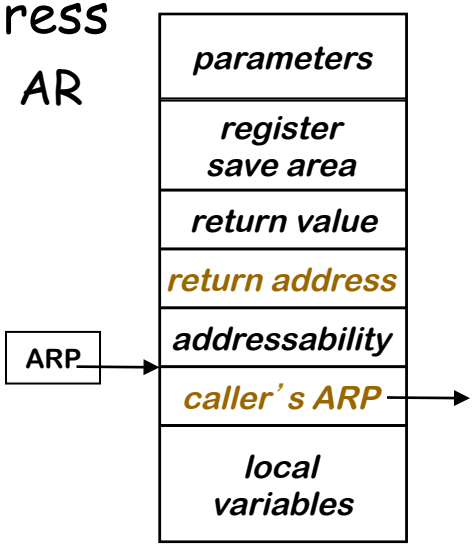


Pre-call Sequence

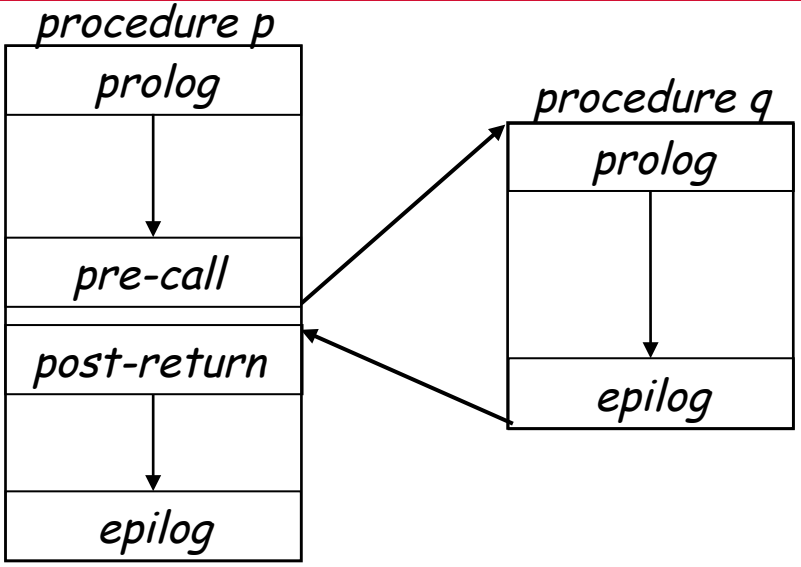
- Sets up callee's basic AR
- Helps preserve its own environment

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
- Save any caller-save registers
  - → Save into space in caller's AR
- Jump to address of callee's prolog code



Procedure Linkages

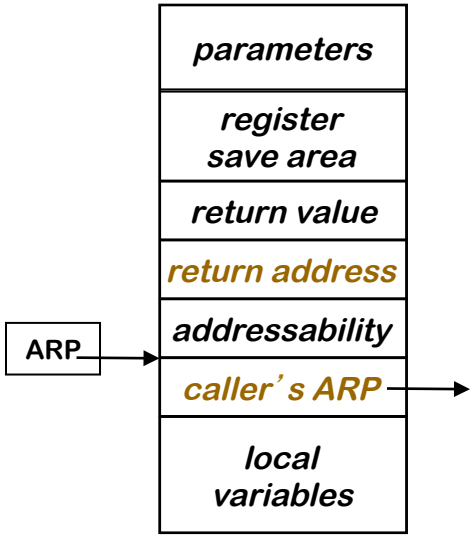


Post-return Sequence

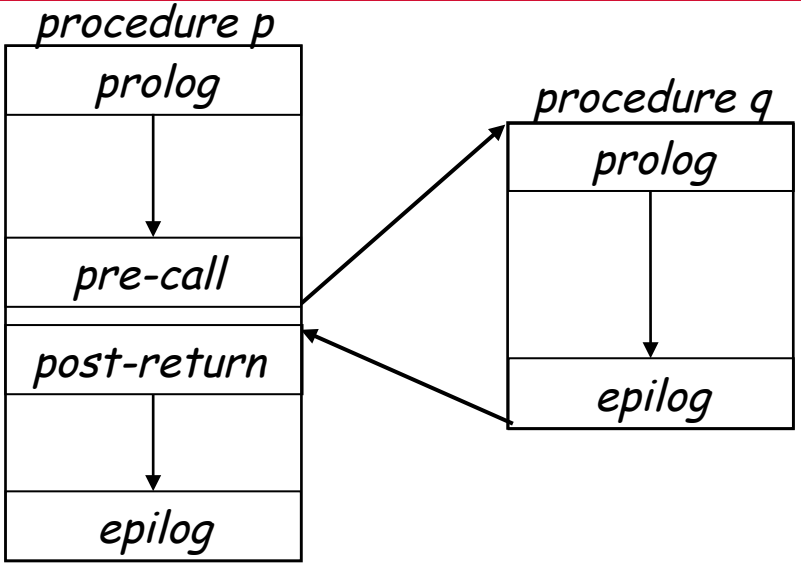
- Finish restoring caller's environment
- Place any value back where it belongs

The Details

- Copy return value from callee's AR, if necessary
- Free the callee's AR
- Restore any caller-save registers
- Copy back call-by-value-result parameters
- Continue execution after the call



Procedure Linkages

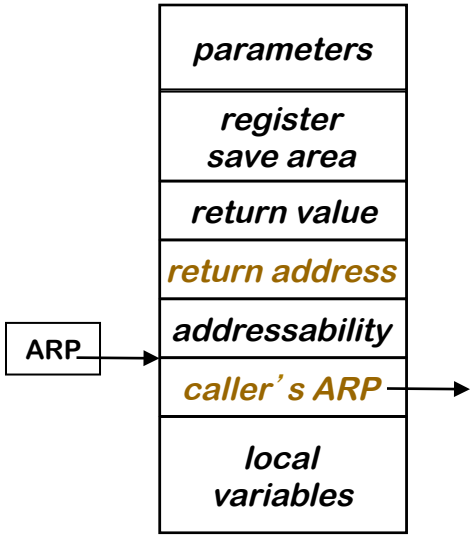


Prolog Code

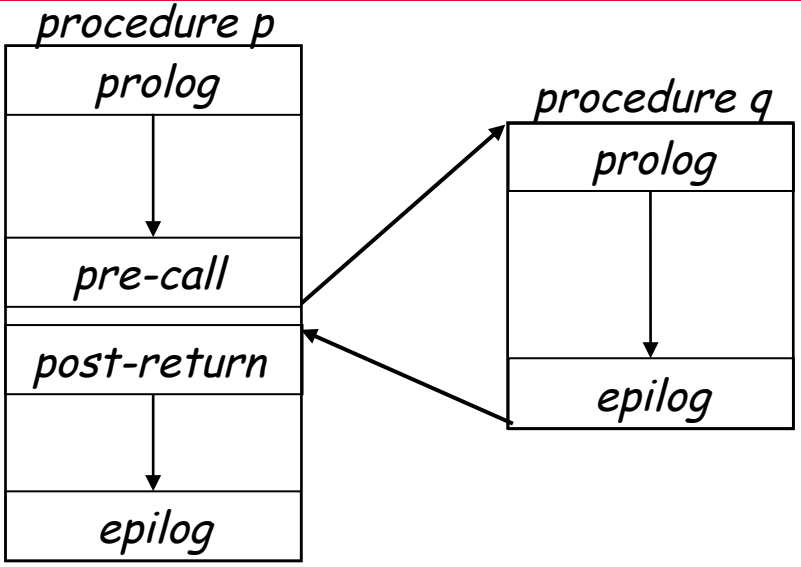
- Finish setting up callee's environment
- Preserve parts of caller's environment that will be disturbed

The Details

- Preserve any callee-save registers
- Allocate space for local data
  - → Easiest scenario is to extend the AR
- Handle any local variable initializations



Procedure Linkages

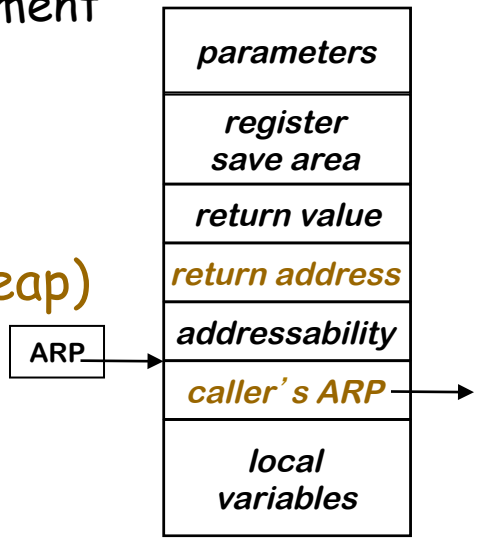


Epilog Code

- 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



Next Class

Inter-procedure Analysis and Optimization

Read EaC: Chapter 9.1 – 9.3, ALSU: Chapter 12