RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers

Procedure Abstractions and Inter-procedure Analysis

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

Announcement

- Project 3 deadline extended
  - → This Saturday 4/30 11:59pm EDT
  - → No late submission allowed after 5/1 11:59pm EDT unless you are previously late for project 2 (you still get the same amount of time for project 3)
  - → Start working on it asap if you haven't
- You can safely assume all test cases are syntactically and semantically correct in project 3
- Homework 10 and the sample solution will be posted
- Final exam – May 6th

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, and operating system

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 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 "grep" at a command line
  - → OS finds the executable
  - → OS creates a process and arranges for it to run "grep"
  - → "grep" 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 "grep" needs system services
  - → It makes a system call, such as fopen()

UNIX/Linux specific discussion

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 Control Abstraction

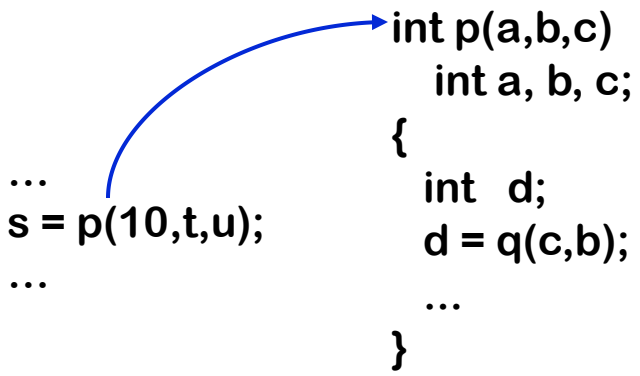
Procedures have well-defined control-flow

- 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

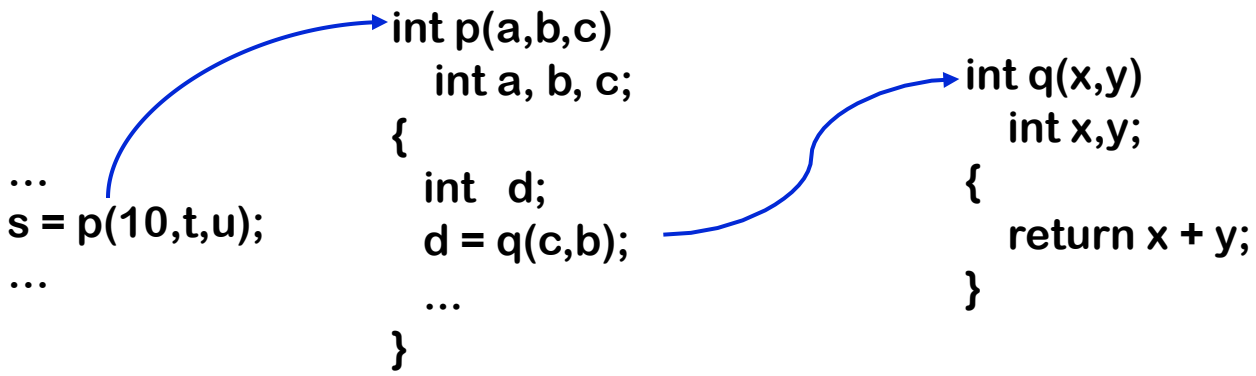
- Invoked at a call site, with some set of *actual parameters*
- Control returns to call site, immediately after invocation



The Procedure as a Control Abstraction

Procedures have well-defined control-flow

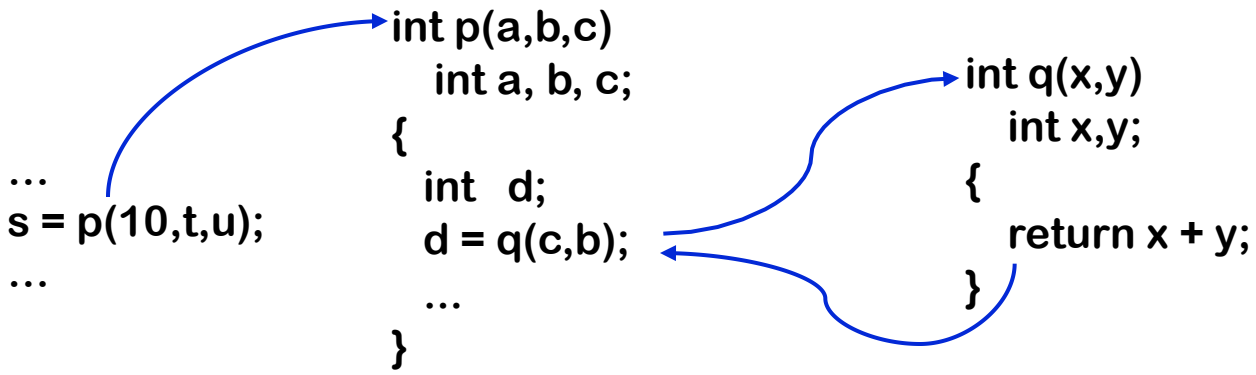
- Invoked at a call site, with some set of *actual parameters*
- Control returns to call site, immediately after invocation



The Procedure as a Control Abstraction

Procedures have well-defined control-flow

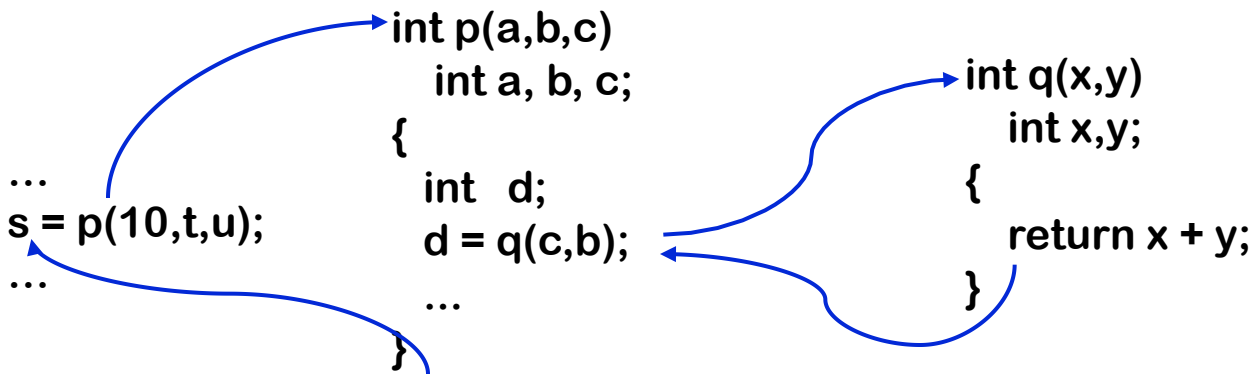
- Invoked at a call site, with some set of *actual parameters*
- Control returns to call site, immediately after invocation



The Procedure as a Control Abstraction

Procedures have well-defined control-flow

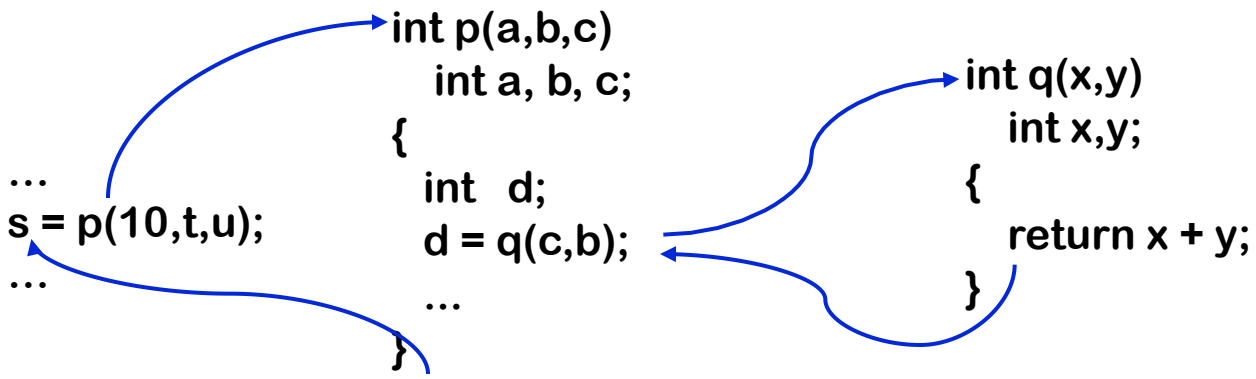
- Invoked at a call site, with some set of *actual parameters*
- Control returns to call site, immediately after invocation



The Procedure as a Control Abstraction

Procedures have well-defined control-flow

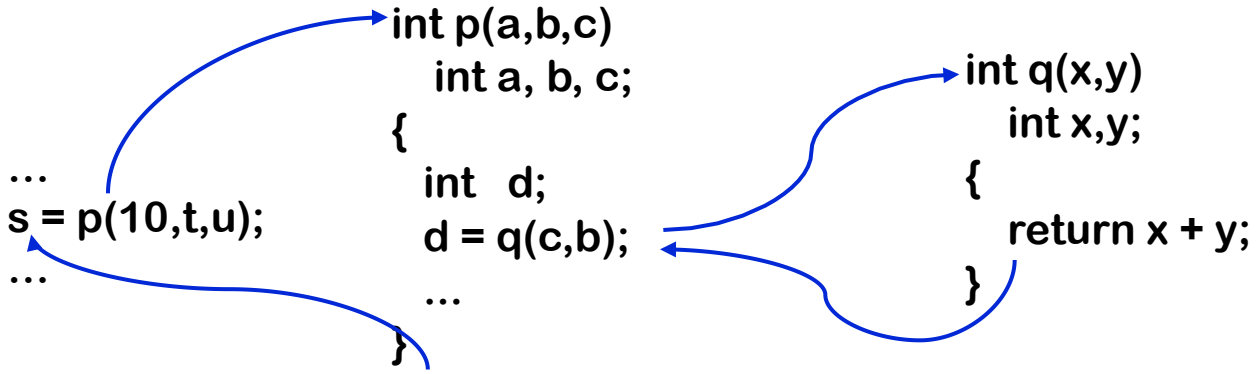
- Invoked at a call site, with some set of *actual parameters*
- Control returns to call site, immediately after invocation



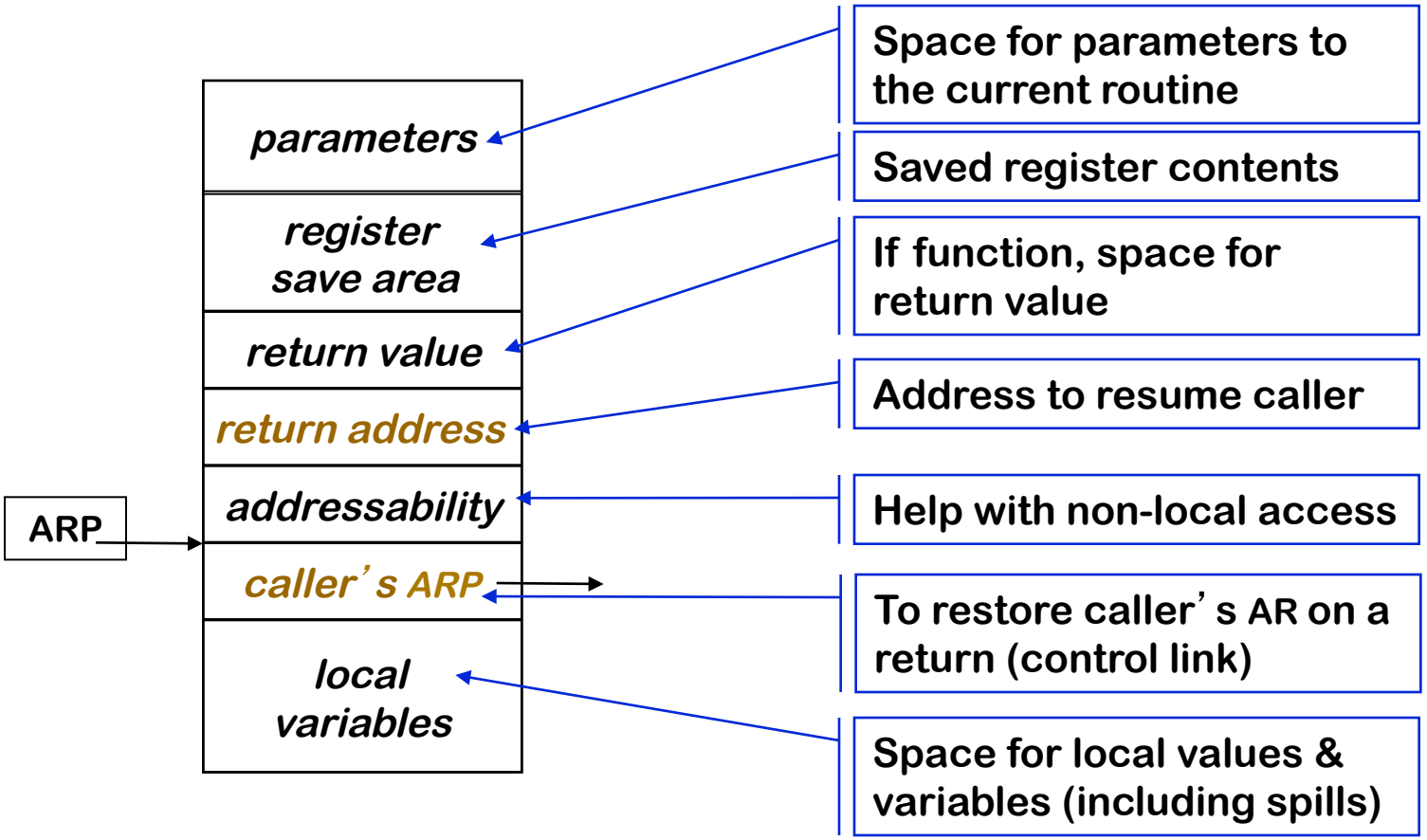
The Procedure as a Control Abstraction

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?



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

loadAI r1, c1 ⇒ r2 : MEM(r1 + c1) → r2

Variable-length data

- Leave a pointer at a known offset from ARP
- Otherwise, put variable-length data on the heap

Initializing local variables

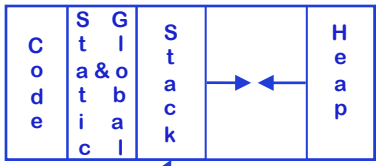
- Must generate explicit code to store the values
- Among the procedure's first actions

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



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

⇒ ARs must 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 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

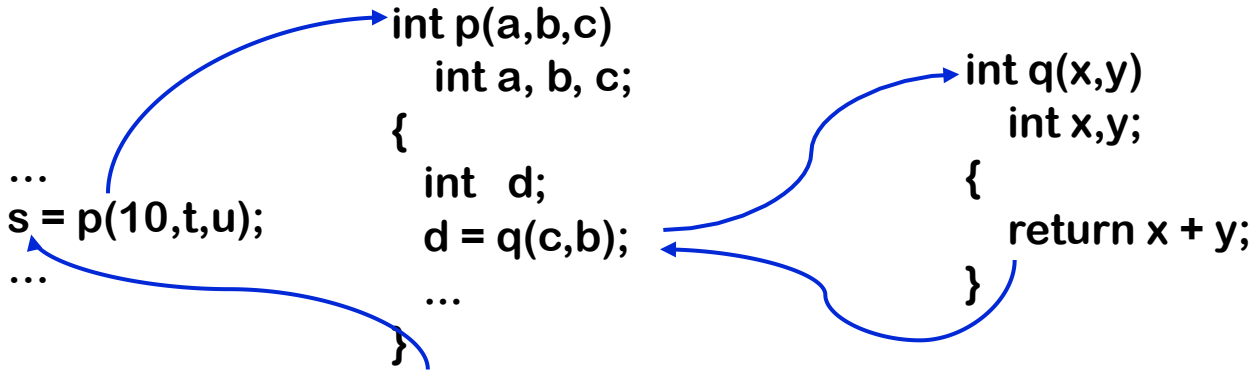
- Can always use global variables … (this is called a side effect)

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

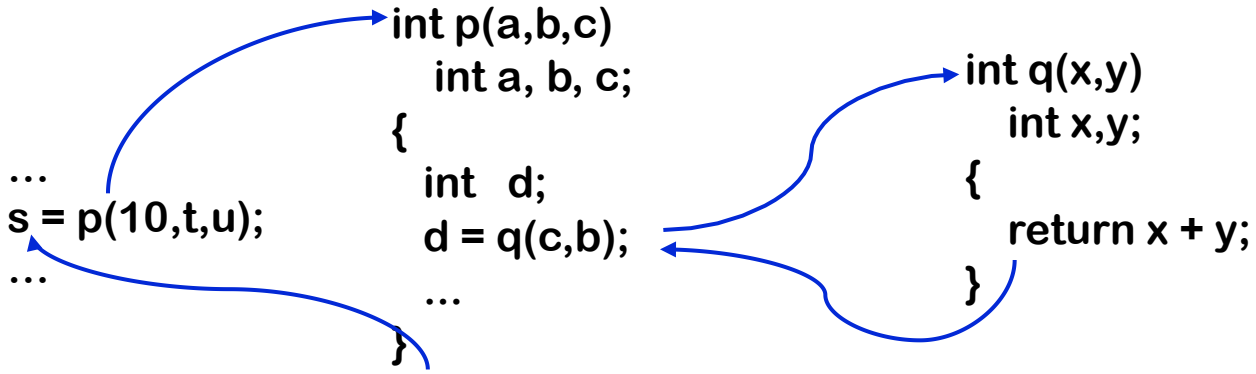


The Procedure as a Control Abstraction

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

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

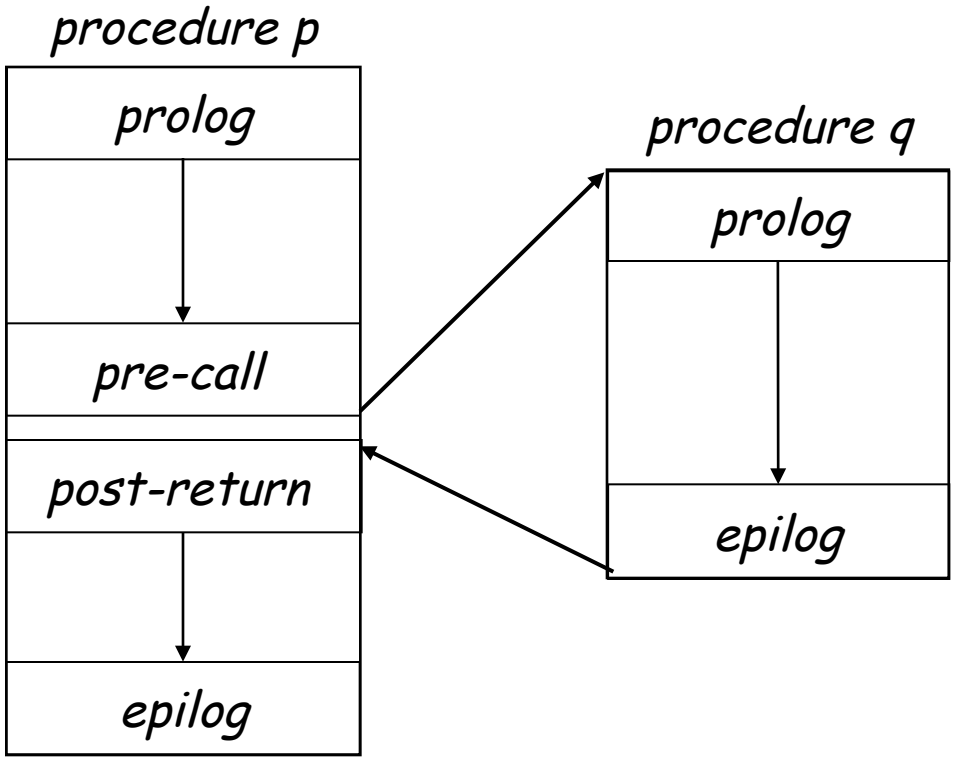
Compiler must use a standard sequence of operations

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

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

Procedure Linkages

Standard procedure linkage

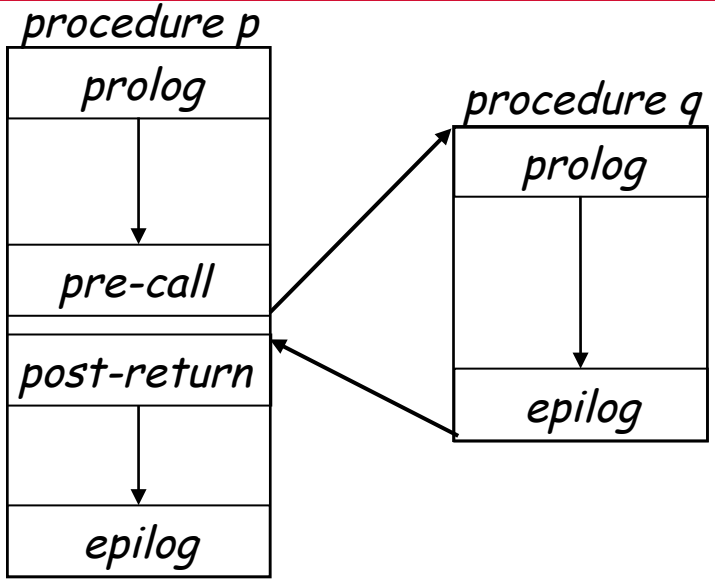


Procedure has
- standard prolog
- standard epilog

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

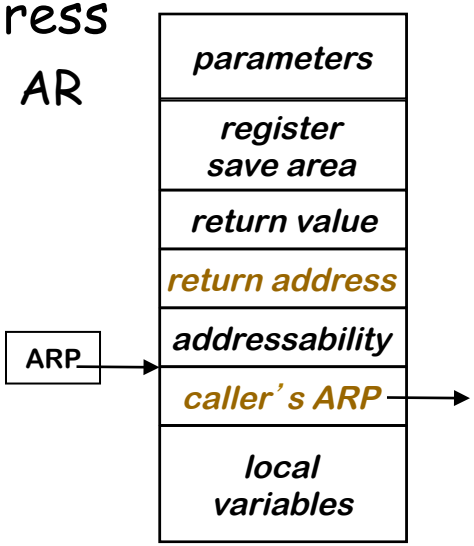


Pre-call Sequence

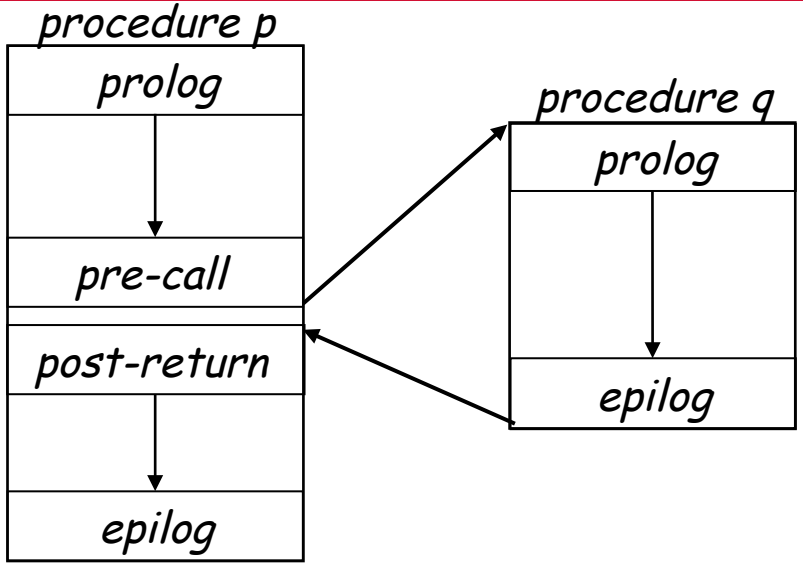
- Sets up callee's basic AR
- Helps preserve its own environment

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

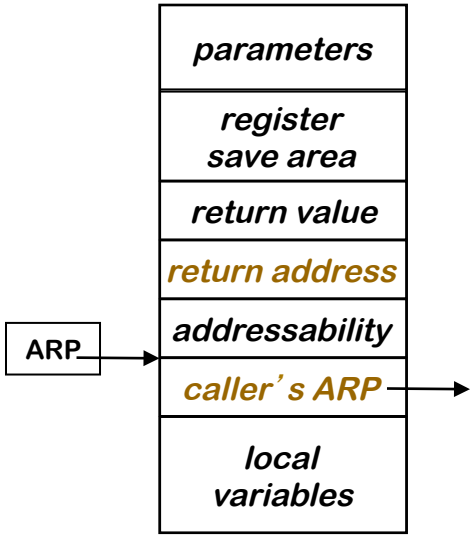


Post-return Sequence

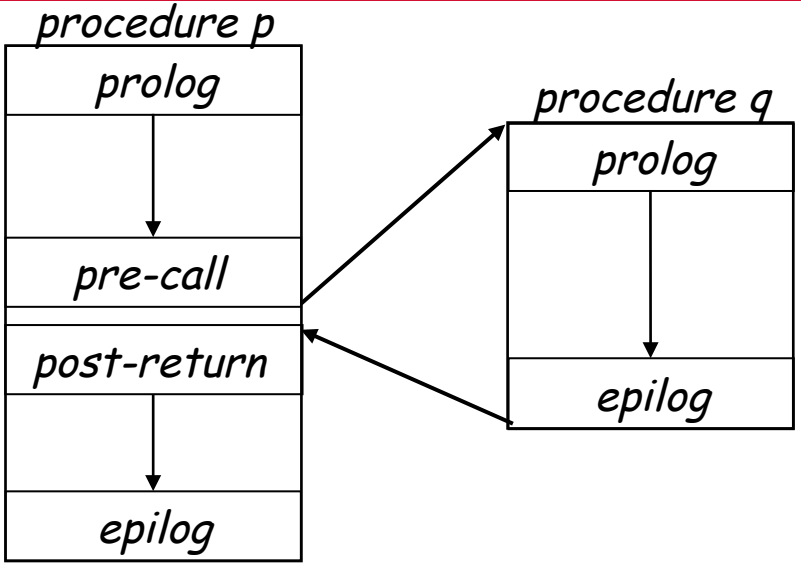
- Finish restoring caller's environment
- Place any value back where it belongs

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

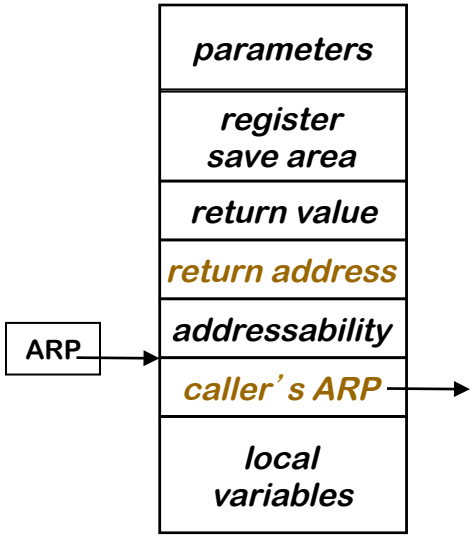


Prolog Code

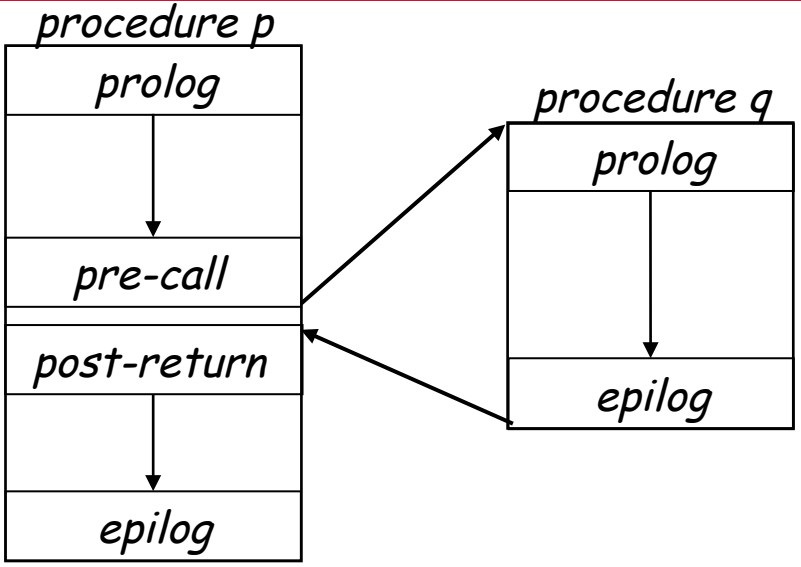
- Finish setting up callee's environment
- Preserve parts of caller's environment that will be disturbed

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

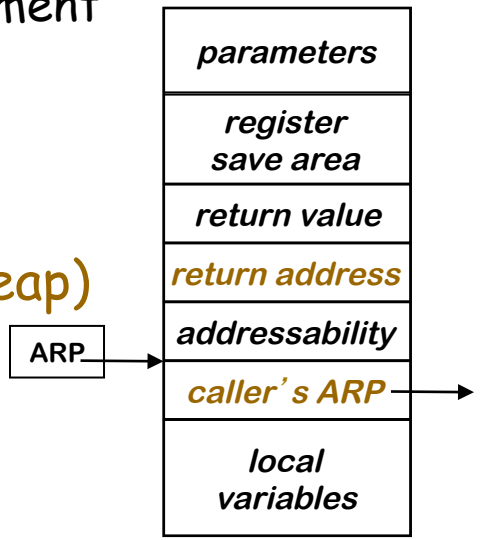


Epilog Code

- 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



A Coin Has Two Sides

Compiler uses a **standard** sequence of operations

- Enforces control & data abstractions
- Divided responsibility between caller & callee
- Caller assumes no knowledge of callee, vice versa

Separation of concerns v.s. Whole program optimization

- Benefits of whole program analysis
  - → Can produce better code around call sites (saves and stores)
  - → Can provide shaper global analysis
  - → Can present the optimizer with more context
  - → Can provide tailored copies of procedures

What can be so difficult?

```
procedure joe(i,j,k)
        l <- 2 * k
        if (j == 100)
            then m <- 10 * j
            else m <- i
        call ralph(l,m,k)
        o <- m * 2
        q <- 2
        call ralph(o,q,k)
        write q, m, o, l
```

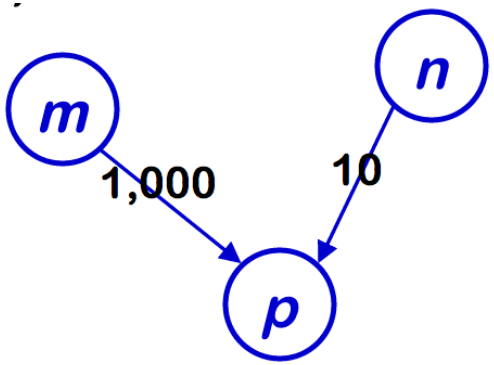
```
procedure main
    call joe( 10, 100, 1000)

procedure ralph(a,b,c)
    b <- a * c / 2000
```

What can be so difficult?

Different calls to p have different properties

- Frequency of the call
- Environment that p inherits
  - → Mapping from parameters to values (& locations)
  - → Constant values & known values
  - → Size of task
- Surrounding execution context
  - → Is call in a parallel loop?
  - → Which registers are unused?
- Procedure-valued parameters

May want to optimize distinct calls separately!



***p* must function correctly in *both* contexts**

What can be so difficult?

```
procedure joe(i,j,k)
        l <- 2 * k
        if (j == 100)
            then m <- 10 * j
            else m <- i
        call ralph(l,m,k)
        o <- m * 2
        q <- 2
        call ralph(o,q,k)
        write q, m, o, l
```

```
procedure main
    call joe( 10, 100, 1000)

procedure ralph(a,b,c)
    b <- a * c / 2000
```

What happens at a procedure call?

- Use worst case assumptions about side effects
- Leads to imprecise intra-procedural information
- Leads to explosion in intra procedural def-use chains

Interprocedure Analysis

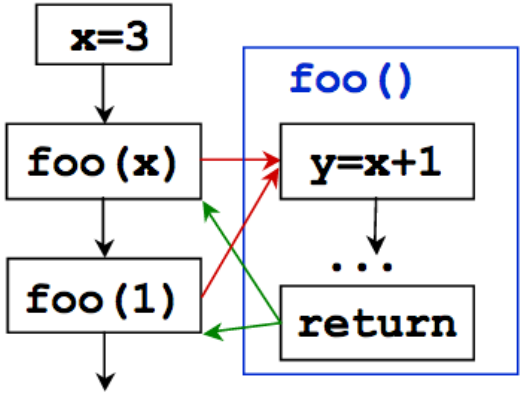
Types of Interprocedure Analysis

- Flow-sensitive v.s. flow-insensitive
  - → Flow-sensitive: analysis at every program point
    Need iterative data-flow analysis (IDFA)
  - → Flow-insensitive: computer answer for an entire procedure
    Might be less accurate than flow-sensitive analysis
- Context-sensitive v.s. context-insensitive
  - → Context-sensitive: assumes knowing context of each caller
  - → Context-insensitive: analysis independent of callers
- Path-sensitive v.s. path-insensitive
  - → Path-sensitive: Computes one answer for every execution path
  - → Path-insensitive: merge information from all paths

Supergraph Construction

Compose CFGs for the entire program

- Connect call nodes to entry of callees
- Connect return nodes of callees back to calls
- Control-flow supergraph



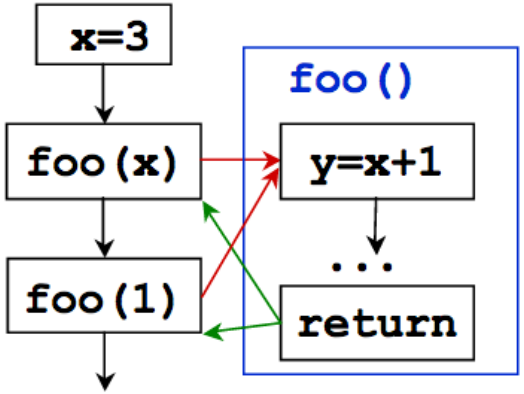
Supergraph Construction

Compose CFGs for the entire program

- Connect call nodes to entry of callees
- Connect return nodes of callees back to calls
- Control-flow supergraph

Pros

- Simple
- Direct use of intraprocedure analysis



Cons

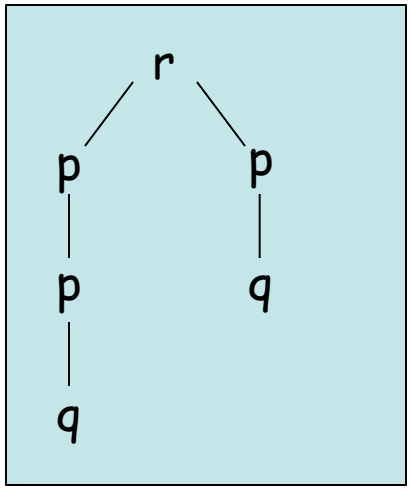
- Loss of accuracy
- Performance
- Requires no separate compilation

Full Context-sensitive Analysis

```
int  r (…) { // declaration
     int   d, s;

     int q (x,y) // declaration
           int x,y;
     {
        return x + y + d;
     }

     int p (a,b,c) // declaration
           int a, b, c;
     {
       int   d;
       if (…)
          d = q (c,b); // call
       else
          d = p (a, d, c); // call
     }
     s = p(10, d, s); // call
     s = p(11, s, d); // call
}
```

Invocation graph

- Distinguish all call-chains
- Analyze callee for all distinct call paths
- Precise but expensive



Procedure Summaries

Compute summary information for each procedure

- Summarize effect of called procedure for callers
- Summarize effect of callers for called procedure
- Tracks information flow into or out of a procedure

- Examples
  - → MAY-ALIAS: the set of formals that may be aliased to globals and each other
  - → MUST-ALIAS, CONST
  - → Side-effect summaries
    - MOD: the set of variables possibly modified (def) by a call
    - REF: the set of variables possibly read (use) by a call
    - KILL: the set of variables killed by a procedure (liveness)

Alternatives to Interprocedure Analysis

- Inlining
  - → Replace a call with procedure body
  - → Pros: reduce call overhead, exposed call context & side effects
  - → Cons: code bloat, library source not necessarily available, not for recursion!

- Procedure cloning/specialization
  - → Create a customized version of procedure for particular call sites
  - → Middle grounds between inlining and interprocedure optimization
  - → Pros: less code bloating
  - → Cons: interprocedure analysis still necessary

Next Class

More inter-procedure Analysis and Optimization

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