RICE

COMP 412
FALL 2009

The Procedure Abstraction: Parts I & II

Comp 412

This lecture starts the second major section of the course, which deals with how to implement source language features.

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 compiler's 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*)
  - *Dynamic loading and linking add more complications*

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*

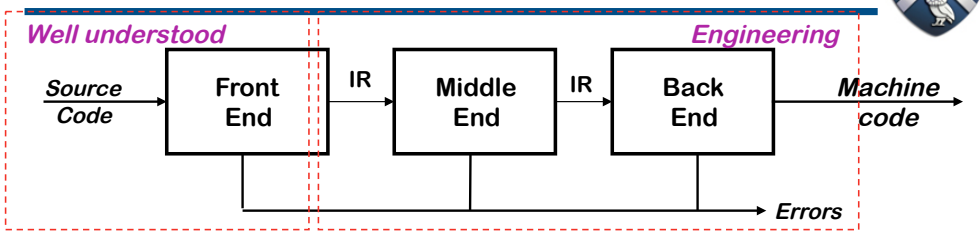
Procedure Abstraction

- 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 behavior
- Assign storage for EVERYTHING, & map names to addresses
- Generate code to compute addresses that the compiler cannot know!
- Interfaces with other programs, other languages, & the OS
- Efficiency of implementation

Where are we?

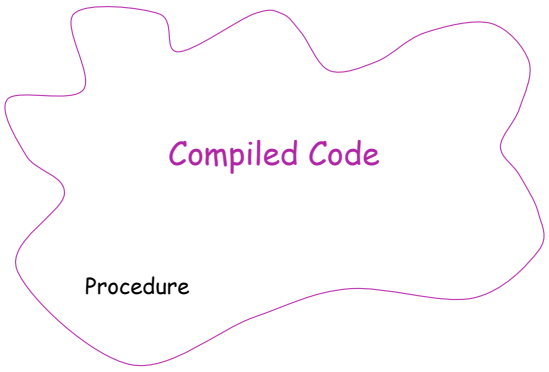


The latter half of a compiler contains more open problems, more challenges, and more gray areas than the front half

- This is "compilation," as opposed to "parsing" or "translation"
- Implementing promised behavior
  - Defining and preserving the meaning of the program
- Managing target machine resources
  - Registers, memory, issue slots, locality, power, …
  - These issues determine the quality of the compiled code

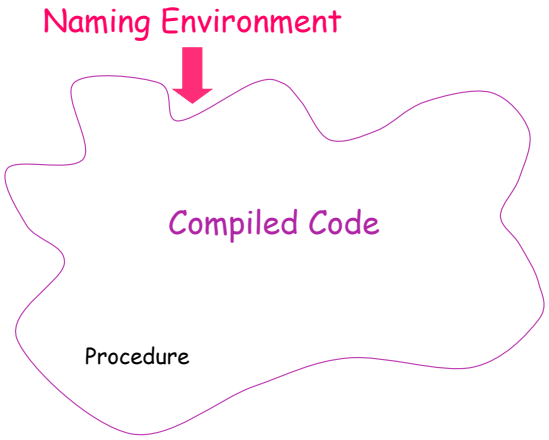
The Procedure & Its Three Abstractions

The compiler produces code for each procedure



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

The Procedure & Its Three Abstractions



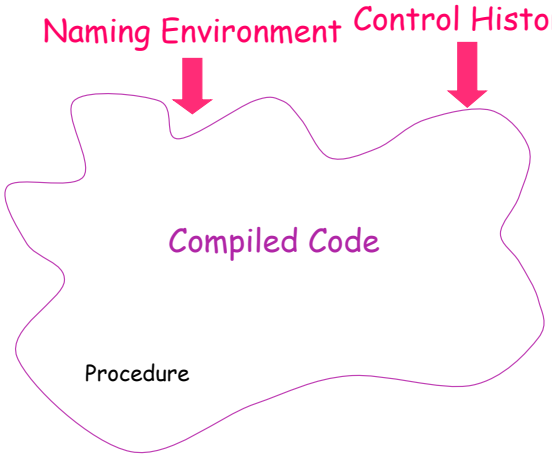
"Naming" includes the ability to find and access objects in memory

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



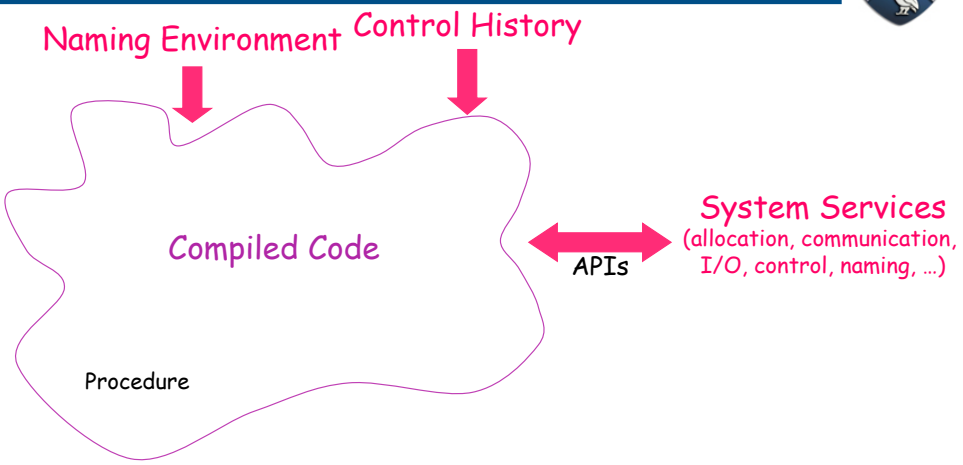
Each procedure inherits a control history

⇒ 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

- 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, & 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!

Run Time versus Compile Time

These concepts are often confusing to the newcomer

- Linkages (*and code for procedure body*) 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

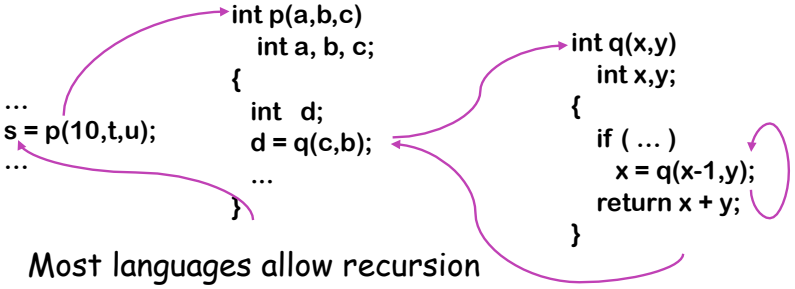
- We will emphasize the distinction between them

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

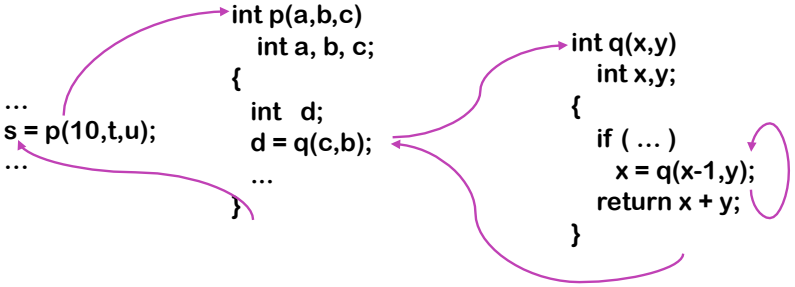


- Most languages allow recursion

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?

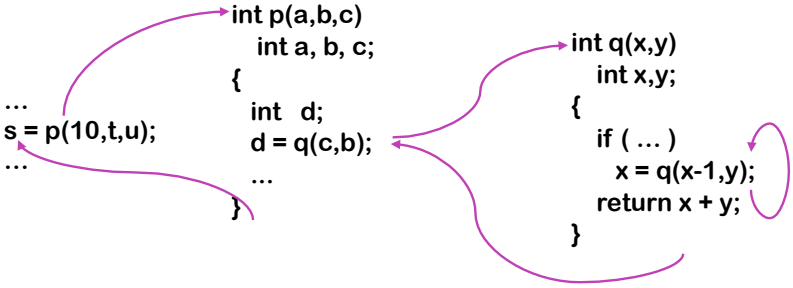


Compiler emits code that causes all this to happen at run time

The Procedure as a Control Abstraction

Implementing procedures with this behavior

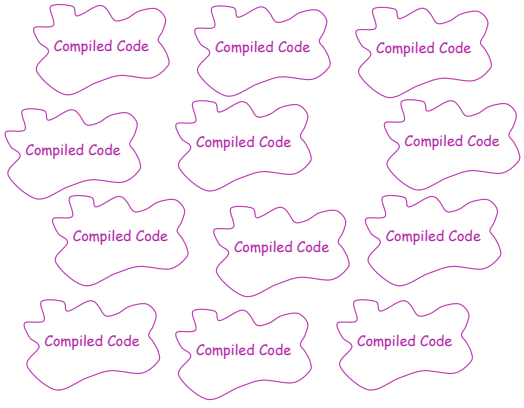
- Must preserve *p*'s state while *q* executes
  - recursion causes the real problem here
- *Strategy:* Create unique location for each procedure activation
  - In simple situations, can use a "stack" of memory blocks to hold local storage and return addresses (*closures ⇒ heap allocate*)

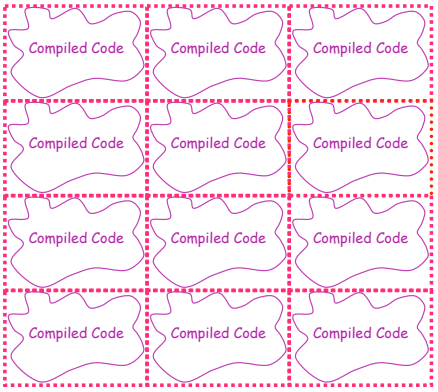


Compiler emits code that causes all this to happen at run time

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