CS415 Compilers

Procedure abstractions

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

- Project 1 (code + reports) has been graded
- Homework #7 is due on Friday, April 26
- Project 2 is due tomorrow, Thursday, April 24, at 3:00pm
- Project 3 will be posted no later than Friday
- Homework #8 (last) will be posted next Monday, April 29
- Final exam: May 14, noon – 3:00pm, SEC-118 (our current room)

CONFLICTS?
If you have a conflict, please send me the details of your conflict:

   class, email of instructor, time of scheduled exam
The Procedure: Three Abstractions

EaC: Chapter 6.1 - 6.5

- **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
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 caller's environment is restored on return
  - The compiler must generate code to ensure this happens according to conventions established by the system
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
These concepts are often confusing

- Procedure linkages execute at run time
- Code for the procedure linkage is emitted at compile time
- The procedure linkage is designed long before either of these

“This issue (compile time versus run time) confuses students more than any other issue” —Keith Cooper (Rice University)
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

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

... s = p(10,t,u);
...
```
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
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

```c
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;
    }
```

... s = p(10,t,u);
...
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

```plaintext
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;
}
```

... s = p(10, t, u);
...
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

```c
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
Implementing procedures with this behavior

- Requires code to save and restore a “return address”
- Must map actual parameters to formal parameters $q:(c\to x, b\to y)$
- 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?

```
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;
}
```
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
  - Can use a “stack” of memory blocks to hold local storage and return addresses

```c
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;
    }
```

\[\text{s} = p(10,t,u);\]

**Compiler emits code that causes all this to happen at run time**
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 \( (\text{Fortran-like}) \)
  → Blocks can be nested, procedures cannot
• Scheme has global, procedure-wide, and nested scopes \( (\text{let}) \)
  → Procedure scope (typically) contains formal parameters
The Procedure as a Name Space

Why introduce lexical scoping?
• Provides a compile-time mechanism for binding “free” variables
• Simplifies rules for naming & resolves conflicts

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
• Lexically scoped symbol tables (see § 5.7.3)
Activation Record Basics

- **parameters**
  - Space for parameters to the current routine

- **register save area**
  - Saved register contents

- **return value**
  - If function, space for return value

- **return address**
  - Address to resume caller

- **addressability**
  - Help with non-local access

- **caller’s ARP**
  - To restore caller’s AR on a return (control link)

- **local variables**
  - Space for local values & variables (including spills)

One AR for each invocation of a procedure
Example: Dynamic vs. Static Views

```c
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
}
```
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
}

(1) dynamic
activation tree

(2) dynamic activation records
in runtime stack

(3) static
symbol table inside proc q
symbol table inside proc p

Example: Dynamic vs. Static Views

(1) dynamic
symbol table inside proc q

(3) static
symbol table inside proc p

d:0.0
d:0.0
s:0.1
s:0.1
x:1.0
a:1.0
y:1.1
b:1.1
c:1.2
d:1.3

nesting level
offset

lexical scoping
More on the runtime environment

Work on the project!