CS415 Compilers

Procedure Abstraction

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

• Homework 8 has been posted, due Wednesday, April 25

• Second project new due date: Wednesday, April 25

• Third project (local CSE) will be posted later this week, due May 2

Final exam: Tuesday, May 8, noon-3:00pm, Physics Lecture Hall, Busch Campus

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 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
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)
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
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

```
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

```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

```
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 \textit{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;
   }
...
s = p(10, t, u);
...
```

- 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 \( q: (c \rightarrow x, b \rightarrow y) \)
- 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?

```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;
}
```

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

```
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;
}
```

Compiler *emits* code that causes all this to happen at run time
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
}
```
```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
            d = p (a, d, c); // call
        }

    s = p(10, d, s); // call
    s = p(11, s, d); // call
}
```

**Example: Dynamic vs. Static Views**

1. **Dynamic activation tree**
   - Symbol table inside proc q
   - Symbol table inside proc p
   - d: 0.0
   - s: 0.1
   - x: 1.0
   - y: 1.1
   - a: 1.0
   - b: 1.1
   - c: 1.2
   - d: 1.3

2. **Dynamic activation records in runtime stack**

3. **Static**
   - Lexical scoping
   - Nesting level
   - Offset
Work on the project!

More on procedure abstraction
Read EaC: Chapter 6.1 - 6.5