RUTGERS

THE STATE UNIVERSITY OF NEW JERSEY

CS415 Compilers

Procedure Abstraction

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

Roadmap for the remainder of the course

- Project #2 – Bottom-up parser and compiler
  New due date: Friday April 22

- Homework #5 due today. Homework #6 has been posted.

- Project #3 – Local Dead Code Elimination for ILOC
  Will be posted by tomorrow

- Final exam on May 10 , 1:00pm (60 minutes in class)

- Grading Scheme
  - → Exams: 2 x 30% (best two exams count)
  - → Projects: 3 x 10%
  - → Homeworks: 5 x 2% (best five homeworks count)

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

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 caller's 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

Run Time versus Compile Time

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

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

```
…
s = p(10,t,u);
…
```

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

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

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

```
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;
}
```

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

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

```
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;
}
```

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

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

```
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;
}
```

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

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

```
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

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:($c \rightarrow x$, $b \rightarrow y$)
- Must create storage for local variables (and, maybe, parameters)
  - → *p* needs space for *d* (and, maybe, *a, b,* and *c*)
  - → where does this space go in recursive invocations?

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

```
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;
}
```

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
  - → Can use a "stack" of memory blocks to hold local storage and return addresses

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

```
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 <u>emits</u> 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 *(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 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

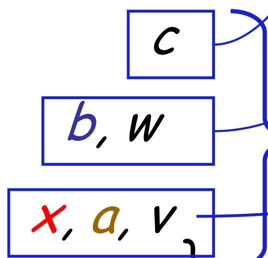
- At point *p*, which declaration of *x* is current?
- At run-time, where is *x* found?
- As parser goes in & out of scopes, how does it delete *x*?

The Answer

- Lexically scoped symbol tables (see § 5.7.3)

Lexically-scoped Symbol Tables

```
procedure p {
    int a, b, c
    procedure q {
        int v, b, x, w
        procedure r {
            int x, y, z
            ....
        }
        procedure s {
            int x, a, v
            ...
        }
        ... r ... s
    }
    ... q ...
}
```

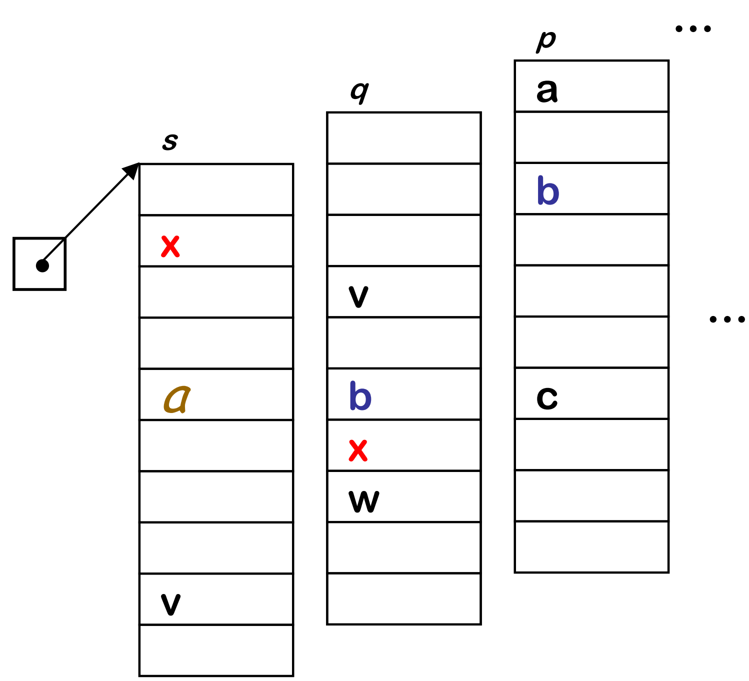


Picturing it as a series of Algol-like procedures

Lexically-scoped Symbol Tables

High-level idea (one possible implementation option – see lecture 19)

- Create a new table for each scope
- Chain them together for lookup



"Chain of tables" implementation

- *insert*() may need to create table
- it always inserts at current level
- *lookup*() walks chain of tables & returns first occurrence of name
- *delete*() throws away table for level *p*, if it is top table in the chain

Individual tables can be hash tables.

RUTGERS

Where Do All These Variables Go?

Automatic & Local

- Keep them in the procedure activation record or in a register
- Automatic ⇒ lifetime matches procedure's lifetime

Static

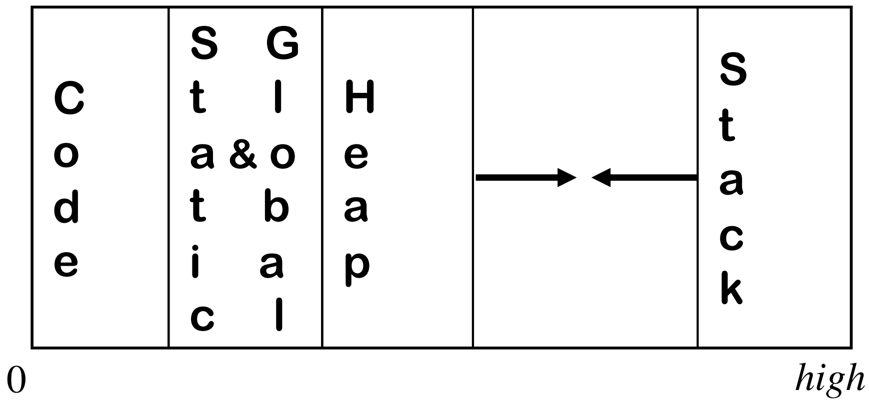
- Procedure scope ⇒ storage area affixed with procedure name
- File scope ⇒ storage area affixed with file name
- Lifetime is entire execution

Global

- One or more named global data areas
- One per program, …
- Lifetime is entire execution

Placing Run-time Data Structures

Classic Organization

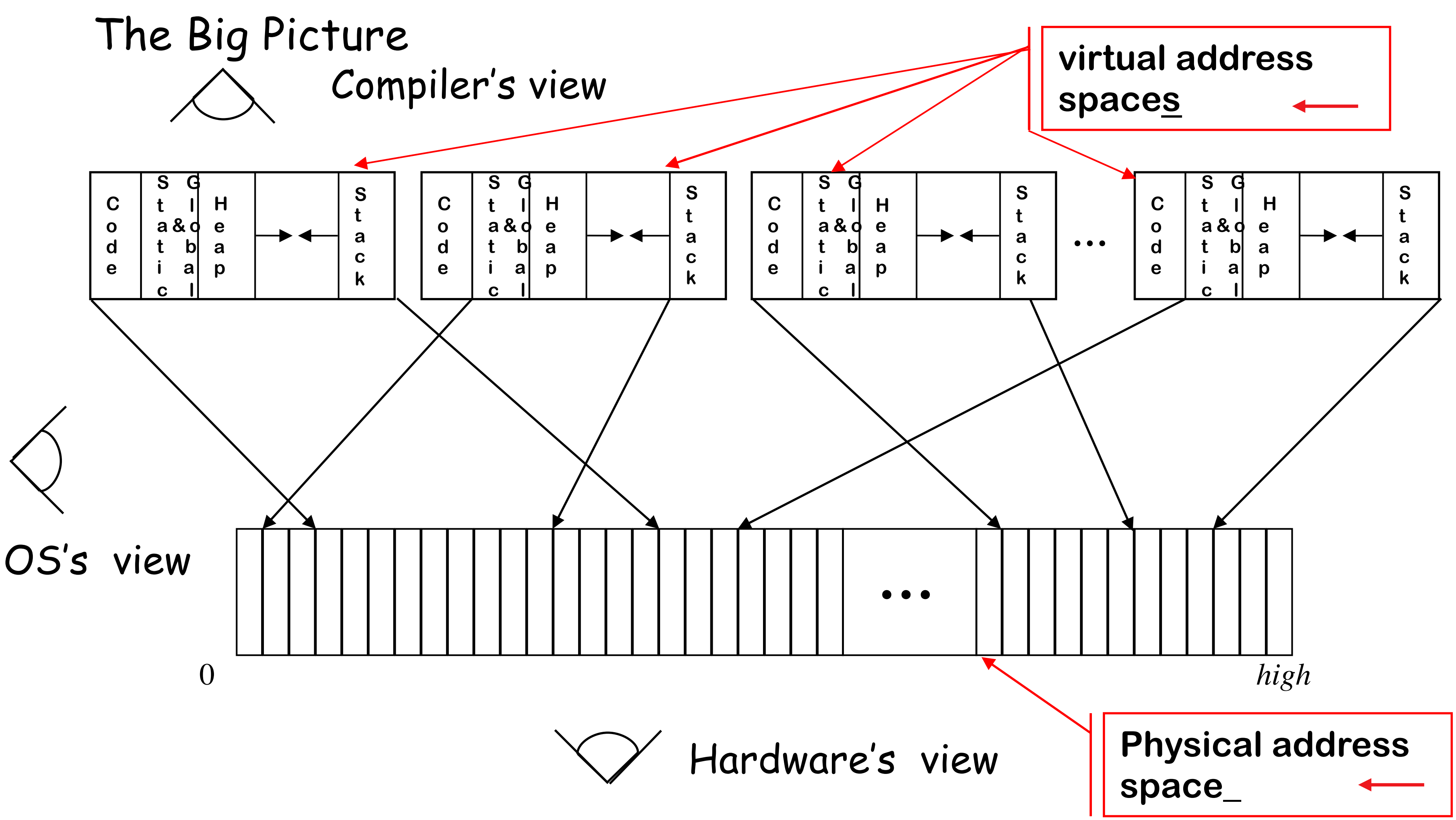


Single Logical Address Space

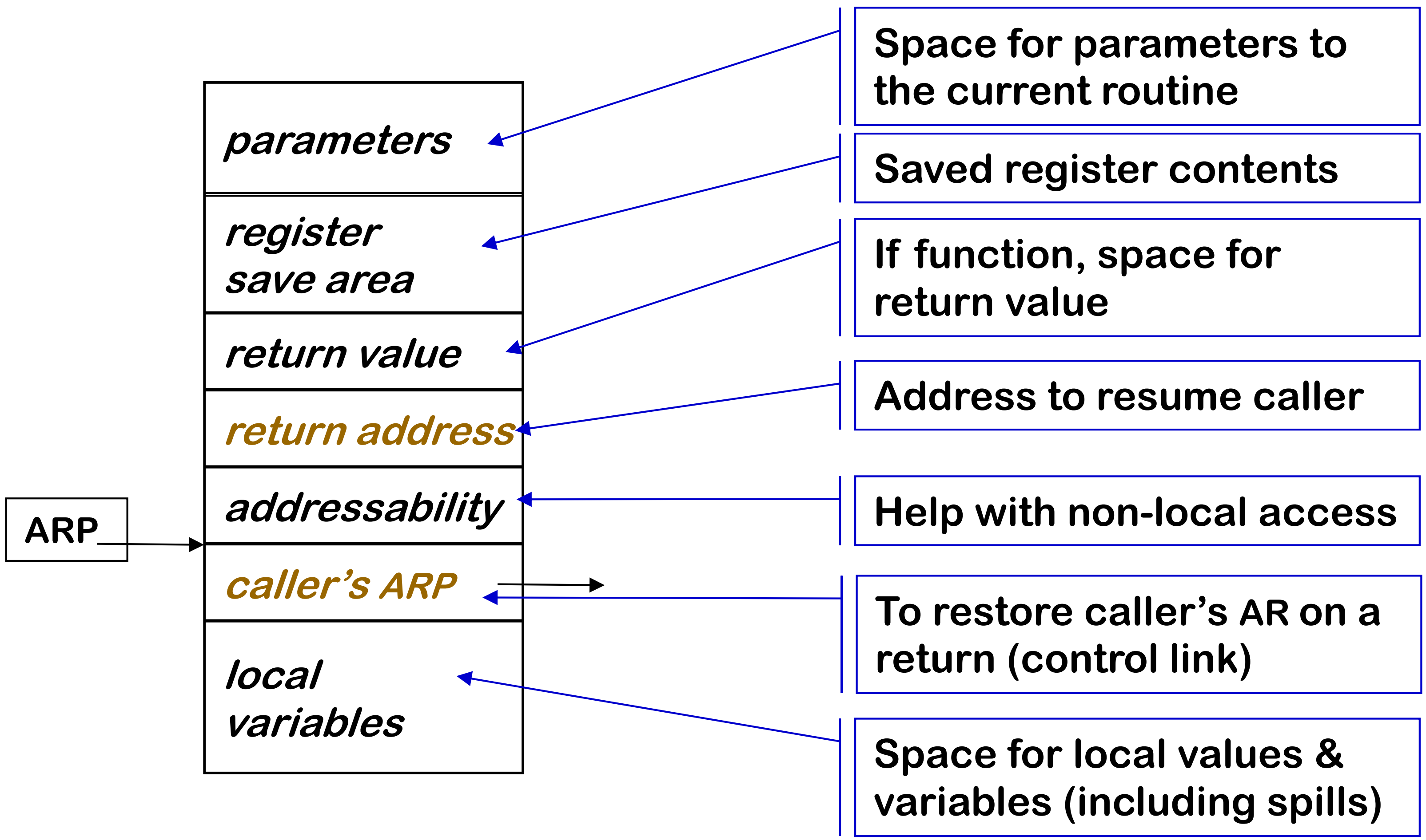
- Better utilization if stack & heap grow toward each other
- Very old result (Knuth)
- Code & data separate or interleaved

- Code, static, & global data have known size
- Heap & stack both grow & shrink over time
- This is a virtual address space

How Does This Really Work?



Activation Record Basics



One AR for each invocation of a procedure

Things to do and next class

Work on the project!

More procedure abstraction

Wrap-up parsing: SLR(1) and LALR(1)

Read EaC: Chapter 3.4