



CS415 Compilers

Procedure Abstraction Part 3

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 #3 - Local Dead-Code Elimination
New due date: Wednesday May 4
- Homework #6 has been posted.
New deadline: Friday, April 29
- Midterm has been graded. Please see sample solution.
Need to ask for regrade by Wednesday, May 4
- Final exam on May 10 , 1:00pm (60 minutes in class)
- Grading Scheme
 - Exams: $2 \times 30\%$ (best two exams count)
 - Projects: $3 \times 10\%$
 - Homeworks: $5 \times 2\%$ (best five homeworks count)

0

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

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

```
1 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
```

}

0

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

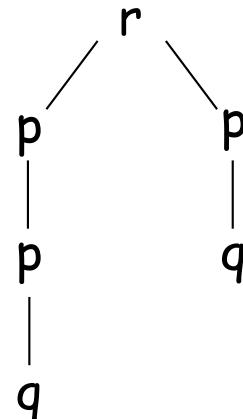
```
1 int q (x,y) // declaration
    int x,y;
{
    return x + y + d;
}
```

```
1 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
```

}

cs415, spring 22

(1) dynamic
activation tree



Lecture 25

0

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

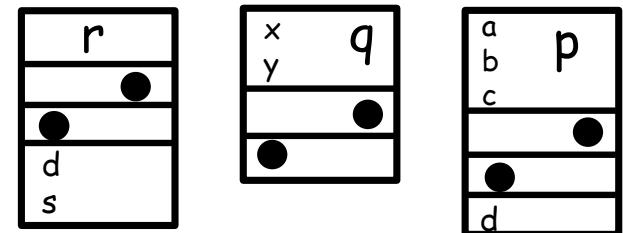
```
1 int q (x,y) // declaration
    int x,y;
{
    return x + y + d;
}
```

```
1 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
```

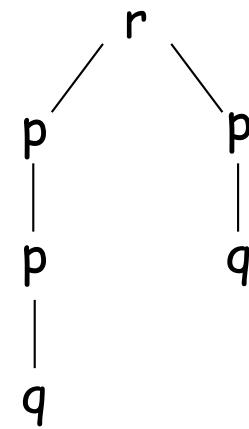
}

cs415, spring 22

(2) dynamic activation records in runtime stack



(1) dynamic activation tree



0

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

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

```
1 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
```

}

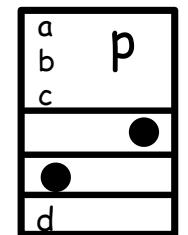
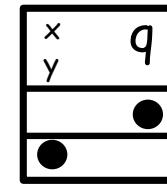
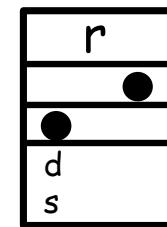
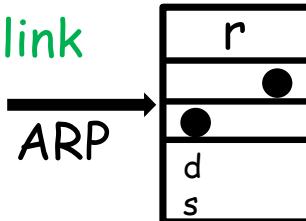
cs415, spring 22

control link

ARP

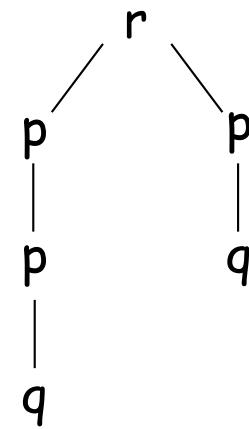
access link

(2) dynamic activation records in runtime stack



(1) dynamic activation tree

RUNTIME STACK



0

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

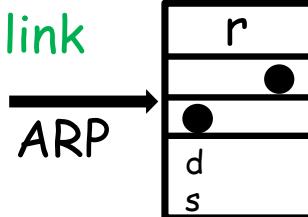
```
1 int q (x,y) // declaration
    int x,y;
{
    return x + y + d;
}
```

```
1 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
```

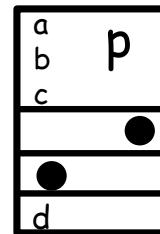
}

cs415, spring 22

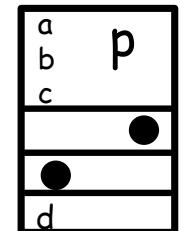
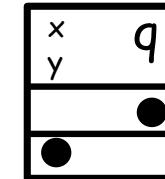
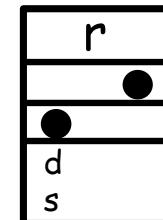
control link



access link



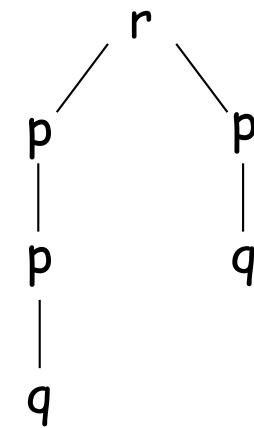
(2) dynamic activation records in runtime stack



(1) dynamic activation tree

RUNTIME STACK

Lecture 25



0

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

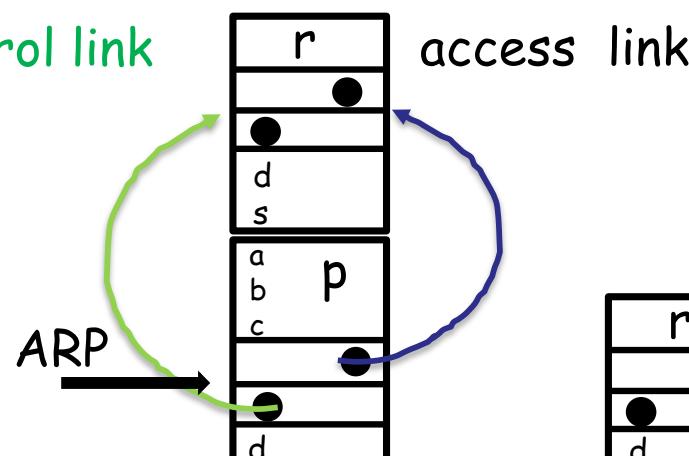
```
1 int q (x,y) // declaration
    int x,y;
{
    return x + y + d;
}
```

```
1 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
```

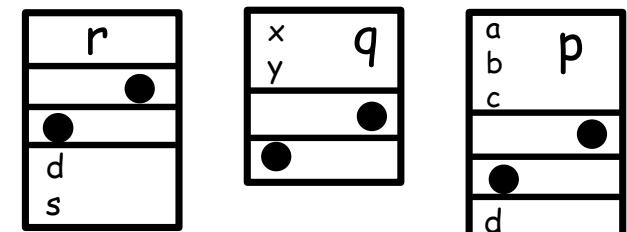
}

cs415, spring 22

control link

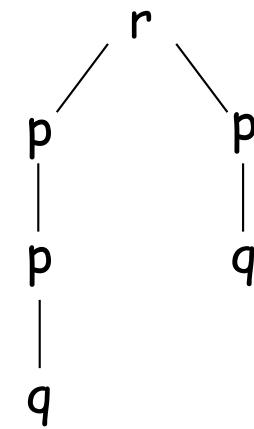


(2) dynamic activation records in runtime stack



(1) dynamic activation tree

RUNTIME STACK



Lecture 25

0

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

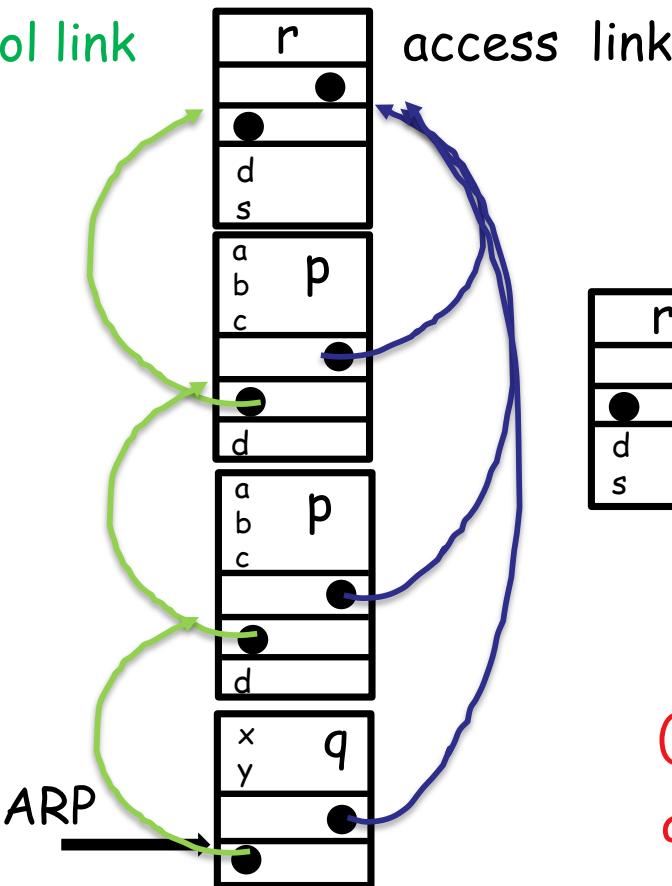
```
1 int q (x,y) // declaration
    int x,y;
{
    return x + y + d;
}
```

```
1 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
```

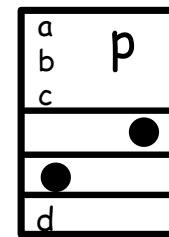
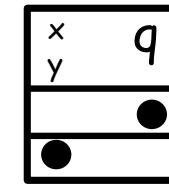
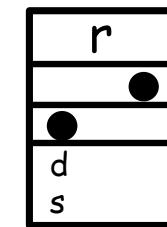
}

cs415, spring 22

control link



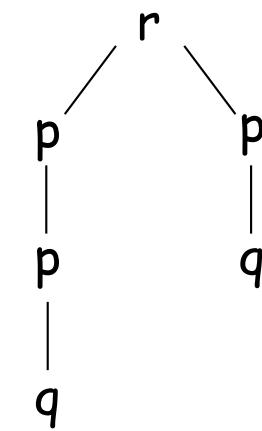
(2) dynamic activation records in runtime stack



(1) dynamic activation tree

RUNTIME STACK

Lecture 25



0

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

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

```
1 int p (a,b,c) // declaration
      int a, b, c;
{
    int d;
    if (...)

        d = q (c,b); // call
    else

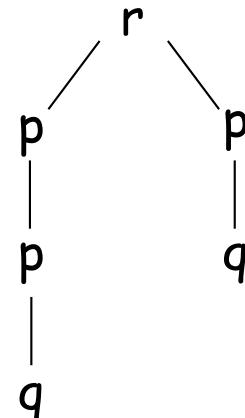
        d = p (a, d, c); // call
}
s = p(10, d, s); // call
s = p(11, s, d); // call
```

int

1

1

(1) dynamic activation tree



(3) static

symbol table symbol table
inside proc q 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

offset

(2) dynamic activation records
in runtime stack

lexical scoping

How does the compiler represent a specific instance of x ?

- Name is translated into a *static coordinate*
 - $\langle \text{level}, \text{offset} \rangle$ pair
 - "level" is lexical nesting level of the procedure
 - "offset" is *unique* within that scope
- Subsequent code will use the static coordinate to generate addresses and references
- "level" is a function of the table in which x is found
 - Known at compile time
 - Stored in the entry for each x
- "offset" must be assigned and stored in the symbol table
 - Assigned at compile time
 - Known at compile time
 - Used to generate code that executes at run-time

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

Variable-length data

$$\text{loadAI } r1, c1 \Rightarrow r2 : \text{MEM}(r1 + c1) \rightarrow r2$$

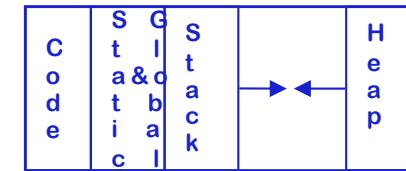
- If activation record (AR) can be extended, put it below local variables
- Leave a pointer at a known offset from ARP
- Otherwise, put variable-length data on the heap

Initializing local variables

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

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



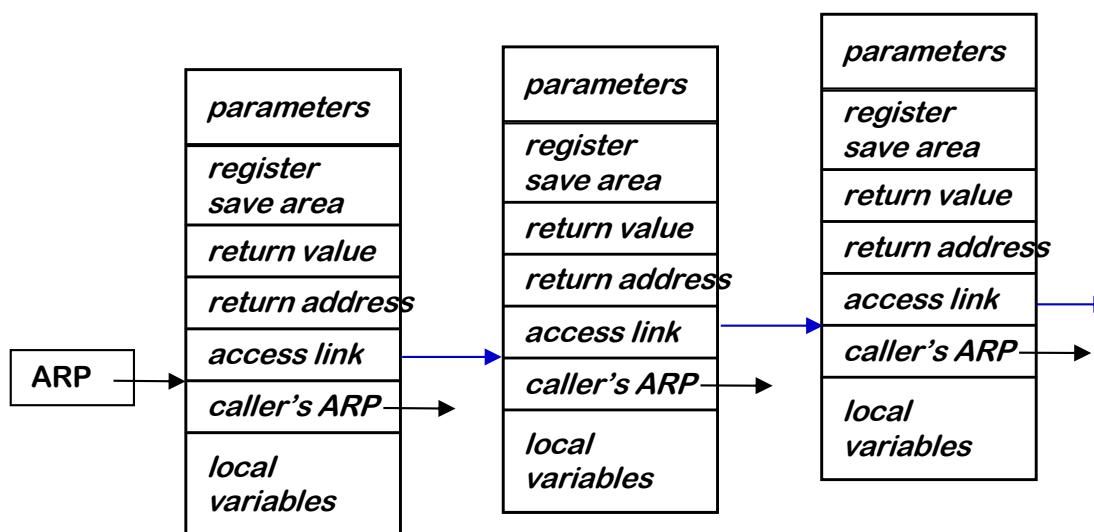
Efficiency prefers static, stack, then heap

Must create base addresses

- Global & static variables
 - Construct a label by mangling names (*i.e.*, &_fee)
 - Local variables
 - Convert to static data coordinate and use ARP + offset
 - Local variables of other procedures
 - Convert to static coordinates (level, offset)
 - Find appropriate ARP
 - Use that ARP + offset
- { Must find the right AR
Need links to nameable ARs**

Using access links (static links)

- Each AR has a pointer to most recent AR of immediate lexical ancestor (`mylevel - 1`)
- Lexical ancestor need not be the caller



Some setup cost
on each call

- Reference to `<p,16>` runs up access link chain to `p`
- Cost of access is proportional to lexical distance

Assume

- Current lexical level is 2
- Access link is at ARP - 4

Using access links

SC	Generated Code
<2,8>	loadAI r ₀ , 8 \Rightarrow r ₂
<1,12>	loadAI r ₀ , -4 \Rightarrow r ₁ loadAI r ₁ , 12 \Rightarrow r ₂
<0,16>	loadAI r ₀ , -4 \Rightarrow r ₁ loadAI r ₁ , -4 \Rightarrow r ₁ loadAI r ₁ , 16 \Rightarrow r ₂

Maintaining access link

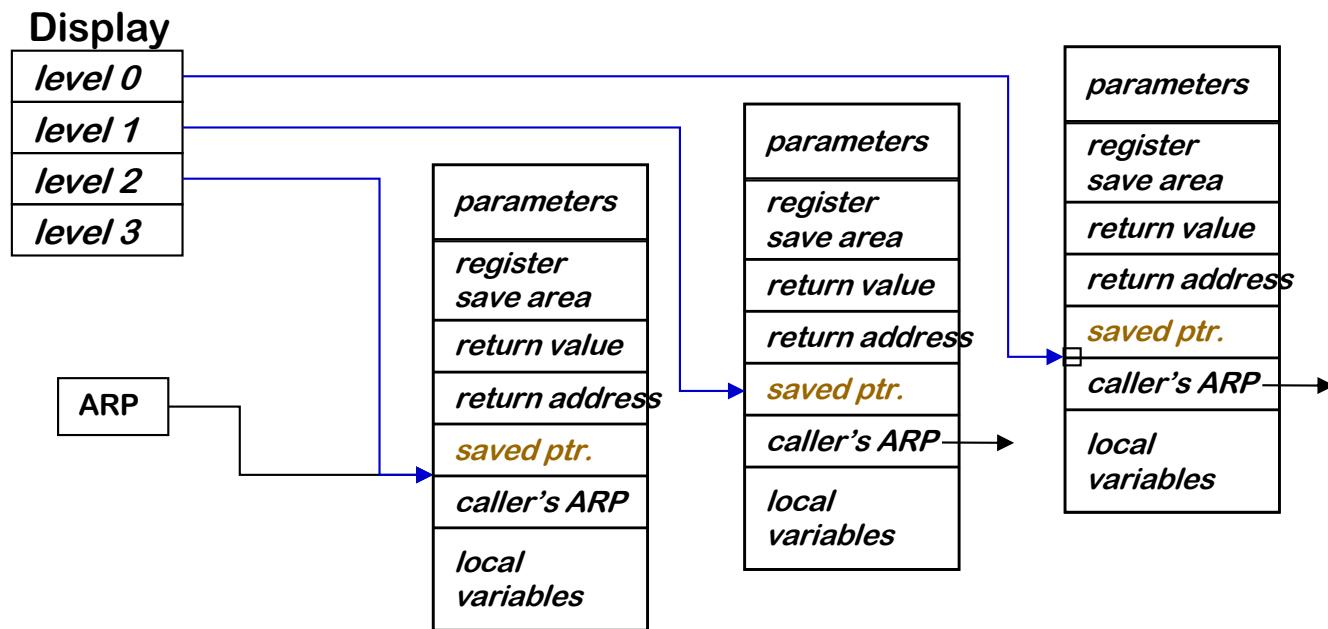
- Calling level $k+1$ (k is current level)
 - Use current ARP as link in new AR
- Calling level $j < k$
 - Find ARP for $j-1$
 - Use that ARP as link in new AR

*Access & maintenance cost varies with level
All accesses are relative to ARP (r₀)*

Using a display

- Global array of pointer to nameable ARPs
- Needed ARP is an array access away

Some setup cost
on each call



- Reference to $\langle p, 16 \rangle$ looks up p 's ARP in display & adds 16
- Cost of access is constant (ARP + offset)

Using a display

SC	Generated Code
<2,8>	loadAI r ₀ , 8 \Rightarrow r ₂
<1,12>	loadI _disp \Rightarrow r ₁ loadAI r ₁ , 4 \Rightarrow r ₁ loadAI r ₁ , 12 \Rightarrow r ₂
<0,16>	loadI _disp \Rightarrow r ₁ loadAI r ₁ , 16 \Rightarrow r ₂

Desired AR is at $_disp + 4 \times level$

Assume

- Current lexical level is 2
- Display is at label _disp

Maintaining access link

- On entry to level j
 - Save level j entry into AR
(Saved Ptr field)
 - Store ARP in level j slot
- On exit from level j
 - Restore level j entry

*Access & maintenance costs are fixed
Address of display may consume a register*

Access links versus Display

- Each adds some overhead to each call
- Access links costs vary with level of reference
 - Overhead only incurred on references & calls
- Display costs are fixed for all references
 - References & calls must load display address
 - Typically, this requires a register

Your mileage will vary

- Depends on ratio of non-local accesses to calls
- Extra register can make a difference in overall speed

For either scheme to work, the compiler must insert code into each procedure call & return

Work on the project!

Procedure abstractions wrap up

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

Read EaC: Chapter 3.4