Class Information

- Midterm exam: Friday, March 14, in class, closed book, closed notes.
- Homework problem set 5 has been posted. Due Tuesday, March 11. Sample solutions will be posted after class.
- Project submission via sakai is now open.
- Project extension until Monday, March 10?
- Final exam: Thursday, May 8, noon to 3:00pm

Block structured programming languages

Binding: variable **x** to memory locations

```
program main;
   var x: int
   procedure foo;
       procedure bar;
           var x: int;
           begin ...
              if (...) foo() else bar();
           end;
       procedure blah;
           var z, y, x: float;
           begin ...
              if (...) bar() else foo();
              x = x + 1; (*)
           end
       begin ...
          if (...) bar() else blah();
          x = x + 1; (**)
       end;
begin
  foo();
end.
```

How to generate code for statements (*) and (**)?

How much do we know about the binding at compile time?

Review: Lexical / Dynamic Scope

$\boxed{\text{lexical}}$

- Non-local variables are associated with declarations at *compile* time
- Find the smallest block syntactically enclosing the reference and containing a declaration of the variable

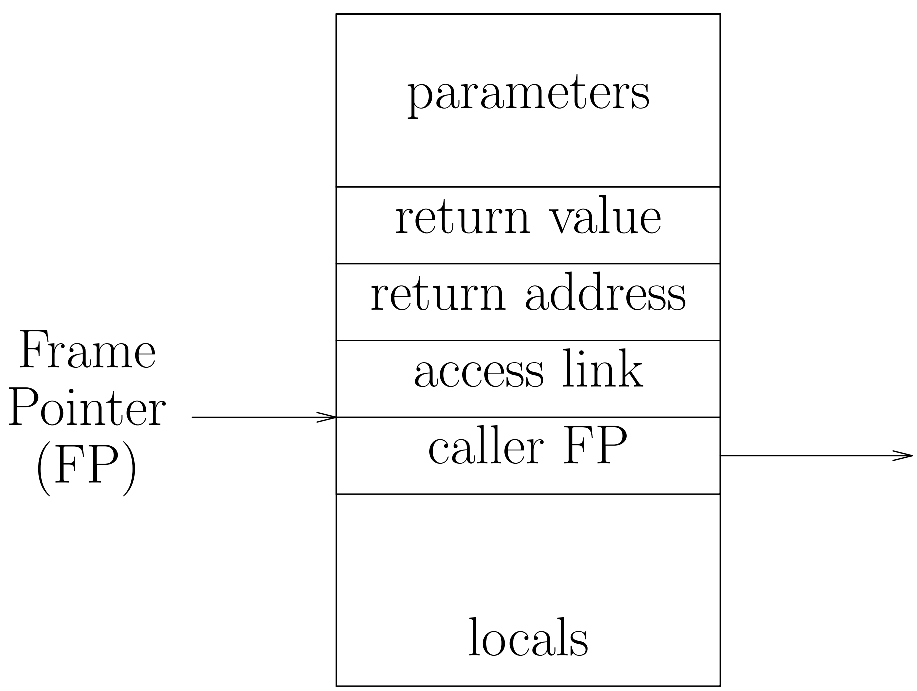
$\boxed{\text{dynamic}}$

- Non-local variables are associated with declarations at *run* time
- Find the most recent, currently active run-time stack frame containing a declaration of the variable

Stack Frame, Activation Record

Scott: Chap. 8.1 - 8.2; ALSU Chap. 7.1 - 7.3

- Run-time stack contains frames for main program and each active procedure.
- Each stack frame includes:
  1. Pointer to stack frame of caller (control link for stack maintainance and dynamic scoping)
  2. Return address (within calling procedure)
  3. Mechanism to find non-local variables (access link for lexical scoping)
  4. Storage for parameters, local variables, and final values



Context of Procedures

Two contexts:

- *static* placement in source code (same for each invocation)
- *dynamic* run-time stack context (different for each invocation)

Scope Rules

Each variable reference must be associated with a single declaration (ie, an offset within a stack frame).

Two choices:

1. Use static and dynamic context: *lexical scope*
2. Use dynamic context: *dynamic scope*

- Easy for variables declared locally, and same for *lexical* and *dynamic* scoping
- Harder for variables not declared locally, and not same for *lexical* and *dynamic* scoping

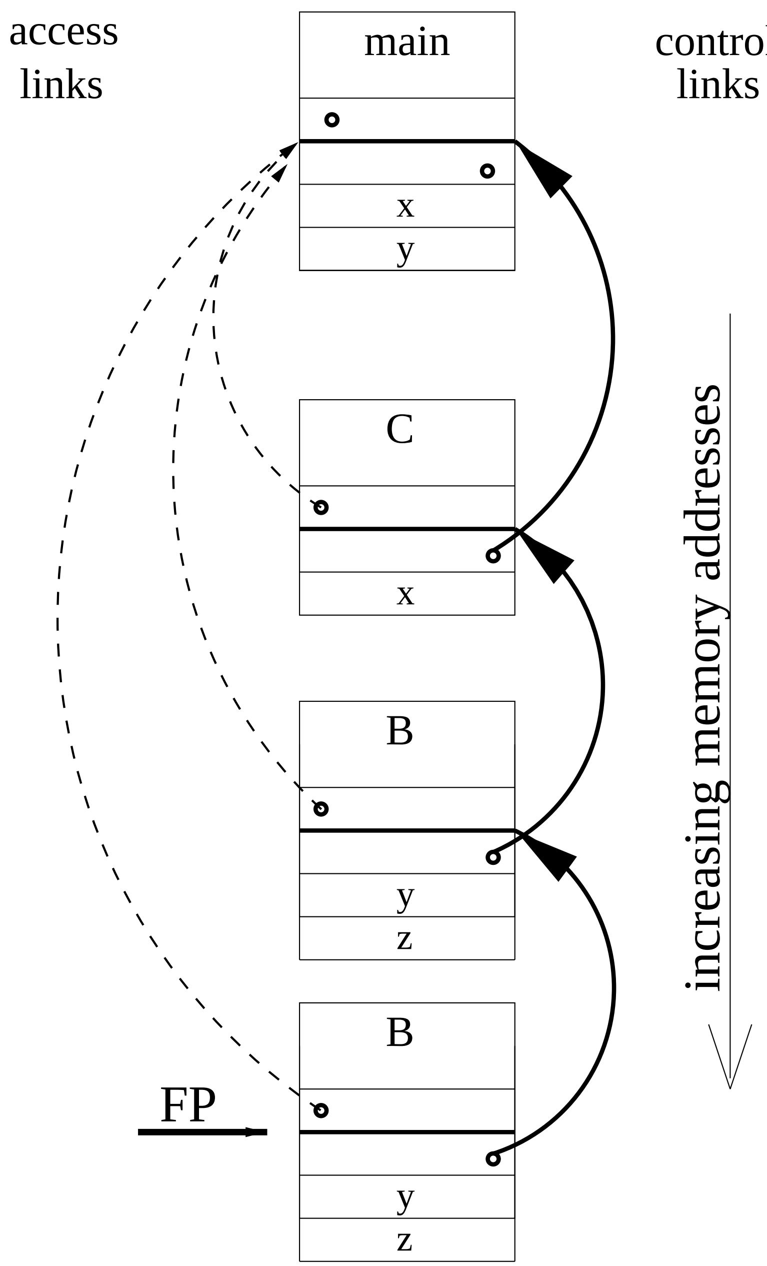
Lexical Scoping Example

scope of a declaration: Portion of program to which the declaration applies

```
Program
   x, y:  integer    // declarations of x and y
   begin
      Procedure B    // declaration of B
         y, z:  real    // declaration of y and z
         begin
            ...
            y = x + z    // occurrences of y, x, and z
            if (...)  call B         // occurrence of B
         end
      Procedure C    // declaration of C
         x:  real       // declaration of x
         begin
            ...
            call B    // occurrence of B
         end
      ...
      call C    // occurrence of C
      call B    // occurrence of B
   end
```

Lexical Scoping Example

Calling chain: MAIN $\Rightarrow$ C $\Rightarrow$ B $\Rightarrow$ B



Scoping and the Run-time Stack

Access links and **control links** may be used to look for non-local variable references.

Static Scope:

> Access link points to stack frame of the most recently activated lexically enclosing procedure

$\Rightarrow$ Non-local name binding is determined at *compile time*, and implemented at *run-time*

Dynamic Scope:

> Control link points to stack frame of caller

$\Rightarrow$ Non-local name binding is determined and implemented at *run-time*

Lexical scoping (de Bruijn notation)

Symbol table matches declarations and occurrences.

$\Rightarrow$ Each $\boxed{\text{variable name}}$ can be represented as a pair

$$(\mathsf{nesting\_level},\ \mathsf{local\_index}).$$

```
Program
   (1,1), (1,2):  integer    // declarations of x and y
   begin
      Procedure B         // declaration of B
         (2,1), (2,2):  real   // declaration of y and z
         begin
            ...    // occurrences of y, x, and z
(*)         (2,1) = (1,1) + (2,2)
            if (...)  call B      // occurrence of B
         end
      Procedure C    // declaration of C
         (2,1):  real      // declaration of x
         begin
            ...
            call B    // occurrence of B
         end
      ...
      call C    // occurrence of C
      call B    // occurrence of B
   end
```

Access to non-local data

How does the code find non-local data at *run-time*?

Real globals

- visible *everywhere*
- translated into an address at compile time

Lexical scoping

- view variables as (*level*,*offset*) pairs (**compile-time symbol table**)
- look-up of (*level*,*offset*) pair uses chains of access links (**at run-time**)
- optimization to reduce access cost: display

Dynamic scoping

- variable names must be preserved
- look-up of variable name uses chains of control links (**at run-time**)
- optimization to reduce access cost: reference table

Access to non-local data (lexical scoping)

What code (ILOC) do we need to generate for statement $(*)$?

```
(2,1) = (1,1) + (2,2)
```

What do we know?

1. The nesting level of the statement is **level 2**.
2. Register r_0 contains the current FP (frame pointer).
3. **(2,1) and (2,2) are local variables**, so they are allocated in the activation record that current FP points to; **(1,1) is a non-local variable**.
4. Two new instructions:

   `LOAD` $R_x \Rightarrow R_y$ means $R_y \leftarrow MEM(R_x)$

   `STORE` $R_x \Rightarrow R_y$ means $MEM(R_y) \leftarrow R_x$

Access to non-local data (lexical scoping)

What code do we need to generate for statement (∗)?

```
(2,1) = (1,1) + (2,2)
```

```
(1,1) | LOADI #4 => r1    // offset of local variable
      |                   // in frame (bytes)
      | LOADI #-4 => r2   // offset of access link
      |                   // in frame (bytes)
      | ADD r0 r2 => r3   // address of access link in frame
      | LOAD r3 => r4     // get access link; r4 now
      |                   // contains ``one-level-up'' FP
      | ADD r4 r1 => r5   // address of first local variable
      |                   // in frame
      | LOAD r5 => r6     // get content of variable
      |
(2,2) | LOADI #8 => r7    // offset of local variable in
      |                   // frame (bytes)
      | ADD r0 r7 => r8   // address of second local variable
      |                   // in current frame
      | LOAD r8 => r9     // get content of variable
      |
  +   | ADD r6 r9 => r10  // (1,1) + (2,2)
      |
(2,1) | LOADI #4 => r11   // offset of local variable in frame (bytes)
      | ADD r0 r11 => r12 // address of first local variable
      |                   // in current frame
      |
  =   | STORE r10 => r12  // (2,1) = (1,1) + (2,2)
```