REMINDERS

- There were three versions of the midterm. Exams will be returned in recitation. Sample solutions will be posted.

- Homework 5 has been posted. Due on Friday, November 6.
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 \((\text{level}, \text{offset})\) pairs
  (compile-time symbol table)
- look-up of \((\text{level}, \text{offset})\) 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)

Two important problems arise

1. *How do we map a name into a (level,offset) pair?*

   We use a block structured symbol table
   (compile-time)
   
   • when we look up a name, we want to get the most recent declaration for the name
   • the declaration may be found in the current procedure or in any nested procedure

2. *Given a (level,offset) pair, what’s the address?*

   Two classic approaches
   (run-time)
   
   ⇒ access links
   (static links)
   
   ⇒ displays
Managing non-local data (lexical scoping)

To find the value specified by \((l, o)\)

- need current procedure level, \(k\)
- if \(k = l\), is a local value
- if \(k > l\), must find \(l\)'s activation record
  \[ \Rightarrow \text{follow } k - l \text{ access links} \]
- \(k < l\) cannot occur

Maintaining access links:

If procedure \(p\) is nested immediately within procedure \(q\), the access link for \(p\) points to the activation record of the most recent activation of \(q\).

- calling level \(k + 1\) procedure
  1. pass my FP as access link
  2. my backward chain will work for lower levels
- calling procedure at level \(l \leq k\)
  1. find my link to level \(l - 1\) and pass it
  2. its access link will work for lower levels
The display

To improve run-time access costs, use a display.

- table of access links for lower levels
- lookup is index from known offset
- takes slight amount of time at call
- a single display or one per frame

Access with the display

assume a value described by \((l, o)\)

- find slot as \(DP[l]\) in display pointer array
- add offset to pointer from slot

“setting up the activation frame” now includes display manipulation.
Display management

Single global display:  

*simple method*

_on entry to a procedure at level l_

- save the level $l$ display value
- push FP into level $l$ display slot

_on return_

- restore the level $l$ display value
Review: Procedures

- Modularize program structure
  - **Argument**: information passed from caller to callee (actual parameter)
  - **Parameter**: *local* variable whose value (usually) is received from caller (formal parameter)

- Procedure declaration
  - procedure name, formal parameters, procedure body with local declarations and statement lists, optional result type
  example: `void translate(point *p, int dx)`
Parameters

Scott: Chapter 8.3

Parameter Association

- **Positional association**: Arguments associated with formals one-by-one; example: C, Pascal, Scheme, Java.

- **Keyword association**: formal/actual pairs; mix of positional and keyword possible; example: Ada
  
  ```
  procedure plot(x, y: in real; penup: in boolean)
  
  plot (0.0, 0.0, penup ⇒ true)
  
  plot (penup ⇒ true, x ⇒ 0.0, y ⇒ 0.0)
  ```

Parameter Passing Modes

- **pass-by-value**: C, Pascal, Ada (in parameter), Scheme, Algol 68
- **pass-by-result**: Ada (out parameter)
- **pass-by-value-result**: Ada (in out parameter)
- **pass-by-reference**: Fortran, Pascal (var parameter)
- **pass-by-name** (not really used any more): Algol60
Review: Stack Frames

- 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)
  2. Return address (within calling procedure)
  3. Mechanism to find non-local variables (access link)
  4. Storage for parameters
  5. Storage for local variables
  6. Storage for final values

```
(parameters)

(return value)

(return address)

(access link)

(caller FP)

(locals)
```

Frame Pointer (FP)
begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := k+1;
    j := j+2;
  end r;

... 
  m := 5;
  n := 3
  r(m,n);
  write m,n;
end

Output?:
Pass-by-value

begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := k+1;
    j := j+2;
  end r;
...
  m := 5;
  n := 3
  r(m,n);
  write m,n;
end

Output:
5 3

Advantage: Argument protected from changes in callee
Disadvantage: Copying of values takes execution time and space, especially for aggregate values (e.g.: arrays, structs).
Pass-by-reference

begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := k+1;
    j := j+2;
  end r;
...
  m := 5;
  n := 3
  r(m,n);
  r(m,n);
  write m,n;
end

Output:
6 5

Advantage: more efficient than copying
Disadvantage: leads to aliasing: there are two or more names for the same storage location; hard to track side effects
Next Lectures Roadmap

- More on parameter passing.
- Introduction to functional languages; read Scott Chapter 10
- Lambda calculus