CS 314 Principles of Programming Languages

Lecture 12

Zheng Zhang

Department of Computer Science
Rutgers University

Friday 14th October, 2016
Class Information

- Midterm exam changed to Wednesday, November 2, in class, closed book, closed notes.
- Homework 5 will be posted this weekend or early next week.
Review: How to Maintain Bindings

**Binding** – association of a name with an attribute (e.g., a name and a memory location, a function name and its “meaning”, a name and a value)

- symbol table: maintained by compiler during compilation  
  names ⇒ attributes
- environment: maintained by compiler generated code during program execution  
  names ⇒ memory locations
- memory: maps memory locations to values  
  memory locations ⇒ values

Key Questions:

- What initiates a binding?
- What ends a binding?
- How long do bindings for a name hold in a program?
Review: Lexical / Dynamic Scope

**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

**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
Procedure Activations

- Begins when control enters activation (call)
- Ends when control returns from activation

Example:
Stack Frame, Activation Record

- 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 maintenance 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
Lexical Scoping and Dynamic Scoping Example

How do we look for non-local variables?

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
  end
  . . .
  call C // occurrence of C
  call B // occurrence of B
end
Lexical Scoping and Dynamic Scoping Example

Calling chain: MAIN ⇒ C ⇒ B ⇒ B
Look up non-local variable reference

**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*

⇒ Non-local name binding is determined at *compile time*, and implemented at *run-time*

**Dynamic Scope:**

*Control link points to stack frame of caller*

⇒ Non-local name binding is determined and implemented at *run-time*
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)
- Use (level, offset) pair to get address by using chains of access links (at run-time)

Dynamic scoping

- variable names are preserved
- look-up of variable name uses chains of control links (at run-time)
Lexical scoping (de Bruijn notation)

Symbol table (generated at compile-time) matches declarations and occurrences.
⇒ Each name can be represented as a pair
   (nesting_level, local_index).

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

Program
  (1,1), (1,2): integer // declarations of x and y
  begin
  Procedure (1,3) // declaration of B
    (2,1), (2,2): real // declaration of y and z
    begin
      (2,1) = (1,1) + (2,2)
      if (...) call (1,3) // occurrence of B
    end
  Procedure (1,4) // declaration of C
    (2,1): real // declaration of x
    begin
      call (1,3) // occurrence of B
    end
  . . .
  call (1,4) // occurrence of C
  call (1,3) // occurrence of B
  end
Access to non-local data (lexical scoping)

What code 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** \( R_x \) \( R_y \) means \( R_x \leftarrow MEM(R_y) \)
   
   **STORE** \( R_x \) \( R_y \) means \( MEM(R_x) \leftarrow R_y \)
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 r1 #4  // offset of local variable
       // in frame (bytes)
    LOADI r2 #-4  // offset of access link
       // in frame (bytes)
    ADD r3 r0 r2  // address of access link in frame
    LOAD r4 r3    // get access link; r4 now
                   // contains ‘one-level-up’ FP
    ADD r5 r4 r1  // address of first local variable
                   // in frame
    LOAD r6 r5    // get content of variable

(2,2)  LOADI r7 #8  // offset of local variable in
       // frame (bytes)
    ADD r8 r0 r7  // address of second local variable
                   // in current frame
    LOAD r9 r8    // get content of variable

    +    ADD r10 r6 r9  // (1,1) + (2,2)

(2,1)  LOADI r11 #4  // offset of local variable in frame (bytes)
    ADD r12 r0 r11  // address of first local variable
                   // in current frame

    =    STORE r12 r10  // (2,1) = (1,1) + (2,2)
```
Access to non-local data (lexical scoping)

Two important steps

1. **Compile-time:** How do we map a name into a \((\text{level,offset})\) 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. **Runtime:** Given a \((\text{level,offset})\) pair, what’s the address? Two classic approaches
   - access links (**static links**)
   - displays
Access to non-local data (lexical scoping)

Runtime: To find the value specified by \((l, o)\)
Assume nested procedure has lower index than its parent procedure.

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

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

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

Zheng Zhang
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

Using a display

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

Some setup cost on each call

- Reference to \( p,16 \) looks up \( p \)’s ARP in display & adds 16
- Cost of access is constant \((\text{ARP} + \text{offset})\)
Next Lecture

Things to do:
Continue working on the project. Due Sunday October 23!
Read Scott: Chap. 9.3 ;

Next time:

- Parameter passing styles and their implementation.