CS 314 Principles of Programming Languages

Lecture 13

Zheng Zhang

Department of Computer Science
Rutgers University

Wednesday 19th October, 2016
Class Information

- Reminder: HW5 posted, due this Saturday October 22, 11:55pm EDT.
- Reminder: Project 1 due in five days, October 23, 11:55pm EDT.
- Midterm: November 2, in class, closed-book, closed-notes.
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
Review: 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
Review: Lexical and Dynamic 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
```
Review: Lexical and Dynamic Scoping Example

Calling chain: MAIN ⇒ C ⇒ B ⇒ B
Symbol Table

- Is a compile time data structure.
- Maps variable to their declarations.
- Stores attributes of variables needed, for instance, for type checking and code generation, e.g., (nesting-level, offset) pairs.

There are different implementation choices for symbol tables. One uses a stack of local scopes (block structured symbol table).
Symbol table matches declarations and occurrences.
⇒ Each variable name can be represented as a pair (nesting_level, 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
Review: 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*)

**Dynamic scoping**
- variable names must be preserved
- look-up of variable name uses chains of control links (*at run-time*)
Review: Access to non-local data (lexical scoping)

What code (ILOC) do we need to generate for statement (\(\ast\))?

\[(2,1) = (1,1) + (2,2)\]

What do we know?

1. The nesting level of the statement is \textbf{level 2}.
2. Register \(r_0\) contains the current FP (frame pointer).
3. \((2,1)\) and \((2,2)\) are \textbf{local variables}, so they are allocated in the activation record that current FP points to; \((1,1)\) is a \textbf{non-local variable}.
4. Two new instructions:
   - \textbf{LOAD} \(R_x \Rightarrow R_y\) means \(R_y \leftarrow \text{MEM}(R_x)\)
   - \textbf{STORE} \(R_x \Rightarrow R_y\) means \(\text{MEM}(R_y) \leftarrow R_x\)
Access to non-local data (lexical scoping)

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

$$(2,1) = (1,1) + (2,2)$$

\[
\begin{align*}
(1,1) & \mid \text{ LOADI } #4 \Rightarrow r1 & \quad \text{ offset of local variable} \\
& \quad \text{ in frame (bytes)} \\
& \quad \text{ LOADI } #-4 \Rightarrow r2 & \quad \text{ offset of access link} \\
& \quad \text{ in frame (bytes)} \\
& \quad \text{ ADD } r0 \ r2 \Rightarrow r3 & \quad \text{ address of access link in frame} \\
& \quad \text{ LOAD } r3 \ r4 & \quad \text{ get access link; } r4 \text{ now} \\
& \quad \quad \text{ contains ‘one-level-up’ FP} \\
& \quad \text{ ADD } r4 \ r1 \Rightarrow r5 & \quad \text{ address of first local variable} \\
& \quad \quad \text{ in frame} \\
& \quad \text{ LOAD } r5 \Rightarrow r6 & \quad \text{ get content of variable} \\
\end{align*}
\[
\begin{align*}
(2,2) & \mid \text{ LOADI } #8 \Rightarrow r7 & \quad \text{ offset of local variable in} \\
& \quad \text{ frame (bytes)} \\
& \quad \text{ ADD } r0 \ r7 \Rightarrow r8 & \quad \text{ address of second local variable} \\
& \quad \quad \text{ in current frame} \\
& \quad \text{ LOAD } r8 \Rightarrow r9 & \quad \text{ get content of variable} \\
& + & \text{ ADD } r6 \ r9 \Rightarrow r10 & \quad (1,1) + (2,2) \\
\end{align*}
\[
\begin{align*}
(2,1) & \mid \text{ LOADI } #4 \Rightarrow r11 & \quad \text{ offset of local variable in frame (bytes)} \\
& \quad \text{ ADD } r0 \ r11 \Rightarrow r12 & \quad \text{ address of first local variable} \\
& \quad \quad \text{ in current frame} \\
& = & \text{ STORE } r10 \Rightarrow r12 & \quad (2,1) = (1,1) + (2,2)
\end{align*}
\]
Access to non-local data (lexical scoping)

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

Two classic approaches

(***run-time**)

⇒ access links
⇒ displays

(***static links**
Access to 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
  \(\Rightarrow\) 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
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
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

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

Zheng Zhang
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 9.3

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

```plaintext
procedure plot(x, y: in real; z: in boolean)
... plot (0.0, 0.0, z ⇒ true)
... plot (z ⇒ 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 Frame

Scott: Chap. 9.1 - 9.3; 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 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
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);
  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
begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := k+1; ==> ERROR: CANNOT USE PARAMETERS
    j := j+2; WHICH ARE UNINITIALIZED
    end r;
...
  m := 5;
  n := 3
  r(m,n);
  write m,n;
end

Output: program doesn’t compile or has runtime error
Pass-by-result

begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := 1; ==> HERE IS ANOTHER PROGRAM
    j := 2; THAT WORKS
  end r;
...
  m := 5;
  n := 3
  r(m,m); ==> NOTE: CHANGED THE CALL
  write m,n;
end

Output: 1 or 2?
Problem: order of copy-back makes a difference; implementation dependent.
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:
6 5

Problem: order of copy-back can make a difference; implementation dependent.
begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := k+1;
    j := j+2;
  end r;
...
  /* set c[m] = m */
  m := 2;
  r(m,c[m]); ==> WHAT ELEMENT OF ‘‘c’’ IS ASSIGNED TO?
  write c[1], c[2], ... c[10];
end

Output:
1 4 3 4 5 ... 10 on entry
1 2 4 4 5 ... 10 on exit

Problem: When is the address computed for the copy-back operation? At procedure call (procedure entry), just before procedure exit, somewhere inbetween? (Example: ADA on entry)
Aliasing:

More than two ways to name the same object within a scope

Even without pointers, you can have aliasing through (global $\leftrightarrow$ formal) and (formal $\leftrightarrow$ formal) parameter passing.

```
begin
  j, k, m: integer;
  procedure q(a,b: integer);
  begin
    b := 3;
    m := m*a;
  end
  ...
  q(m,k);  ==> global/formal <m,a> ALIAS PAIR
  q(j,j);  ==> formal/formal <a,b> ALIAS PAIR
  write y;
end
```
Comparison: by-value-result vs. by-reference

Actual parameters need to evaluate to L-values (addresses).

```pascal
begin
  y: integer;
  procedure p(x: integer);
  begin
    x := x+1; ==> ref: x and y are ALIASED
    x := x+y; ==> val-res: x and y are NOT ALIASED
  end
  ...
  y := 2;
p(y);
write y;
end
```

Output:
- pass-by-reference: 6
- pass-by-value-result: 5

Note: by-value-result: Requires copying of parameter values
(expensive for aggregate values); does not have aliasing, but
copy-back order dependence;
Next Lectures Roadmap

▶ Introduction to functional languages; read Scott Chapter 10
▶ Lambda calculus