CS 314 Principles of Programming Languages

Lecture 14: Names, Scopes and Bindings

Zheng (Eddy) Zhang
Rutgers University
March 5, 2018
Review: Program Memory Layout

• Static objects are given an absolute address that is retained throughout the execution of the program

• Stack objects are allocated and deallocated in last-in, first-out order, usually in conjunction with subroutine calls and returns

• Heap objects are allocated and deallocated at any arbitrary time
Review: Procedure Activations

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

Example:

```
procedure C:
    D
procedure B:
    if...then B else C
procedure A:
    B
main program:
    A
```

```
Calling chain: A ⇒ B ⇒ B ⇒ C ⇒ D
```

```
Direction of stack growth (usually towards lower addresses)
```

```
High
```

```
Low
```

```
procedure C:  
D
procedure B:  
    if...then B else C
procedure A:  
    B
main program:
    A
```

```
parameter
```

```
return value
```

```
return address
```

```
access link
```

```
caller FP
```

```
local variables
```

```
sp
```

Review: Procedure Activations

• Run-time stack contains frames from main program & 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
  5. Other temporaries including intermediate values & saved register

<table>
<thead>
<tr>
<th>Stack Frame</th>
<th>or Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter</td>
<td>return value</td>
</tr>
<tr>
<td>return address</td>
<td>access link</td>
</tr>
<tr>
<td>access link</td>
<td>caller FP</td>
</tr>
<tr>
<td>caller FP</td>
<td>local variables</td>
</tr>
<tr>
<td>Frame Pointer (FP)</td>
<td>or Activation Record</td>
</tr>
<tr>
<td>ARP</td>
<td>Usually stored in a register</td>
</tr>
</tbody>
</table>
Lexical Scope v.s. Dynamic Scope

**Lexical Scope**

- 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 Scope**

- 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
Lexical Scope

- 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

Example:
- The reference to \( n \) in \( W \) is associated with two different declarations at two different times
- The output is?

\[
\begin{align*}
\text{program } & L; \\
& \quad \text{var } n \text{: char; \{n declared in L\}} \\
& \quad \text{procedure } W; \\
& \quad \quad \begin{aligned}
& \text{begin} \\
& \quad \text{write (n); \{n referenced in W\}} \\
& \quad \text{end;}
\end{aligned} \\
& \quad \text{procedure } D; \\
& \quad \quad \begin{aligned}
& \text{var } n \text{: char; \{n declared in D\}} \\
& \quad \text{begin} \\
& \quad \quad \text{n := ‘D’; \{n referenced in D\}} \\
& \quad \quad W \\
& \quad \quad \text{end;}
\end{aligned} \\
& \quad \text{begin} \\
& \quad \quad \text{n := ‘L’; \{n referenced in L\}} \\
& \quad \quad W; \\
& \quad D
\end{align*}
\]
Lexical Scope

- 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

Example:
- The reference to n in W is associated with two different declarations at two different times
- The output is?

```
program L;
    var n: char; {n declared in L}
    procedure W;
        begin
            write (n); {n referenced in W}
        end;
    procedure D;
        var n: char; {n declared in D}
        begin
            n := 'D'; {n referenced in D}
            W
        end;
    begin
        n := 'L'; {n referenced in L}
        W;
        D
    end
```

Calling Chain:
```
L ⇒ W
L ⇒ D ⇒ W
```
Dynamic Scope

• 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

• Example:
  - The reference to n in W is associated with two different declarations at two different times
  - The output is ?

Calling Chain:

\[ L \Rightarrow W \]
\[ L \Rightarrow D \Rightarrow W \]
Dynamic Scope

- 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

Example:
- The reference to n in W is associated with two different declarations at two different times
- The output is ?

Calling Chain:
\[ L \Rightarrow W \]
\[ L \Rightarrow D \Rightarrow W \]
Lexical Scope v.s. Dynamic Scope

Lexical Scope

• 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
• Access link points to the most recently activated immediate lexical ancestor

Dynamic Scope

• 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
• Control link points to the caller
Lexical Scoping and Dynamic Scoping Example

Calling chain: MAIN ⇒ C ⇒ B ⇒ B

Program

x, y: integer // declarations of x and y
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
  begin
    ...
    call B // occurrence of B
    end
begin
  ...
  call C // occurrence of C
  call B // occurrence of B
end
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.

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
Access to Non-Local Data (Lexical Scoping)

Using access link:

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

- Need current procedure level \(k\)
- If \(k = l\), it is a local variable
- If \(k > l\), must find \(l\)'s activation record
  \[\Rightarrow\] follow \(k - l\) access link
- \(k < l\) cannot occur
Access to Non-Local Data (Lexical Scoping): Access Link

Using access links (static links)

• Each AR has a pointer to most recent AR of immediate lexical ancestor
• Lexical ancestor does not need to be the caller

Example: reference to \( <p, 16> \) runs up access link to \( p \)

Cost of access link is proportional to lexical distance
Access to Non-Local Data (Lexical Scoping)

Using access link:

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

• Need current procedure level \(k\)
• If \(k = l\), it is a local variable
• If \(k > l\), must find \(l\)’s activation record
  ⇒ follow \(k - l\) access link
• \(k < l\) cannot occur
Maintaining Access Links

Setting up access link:

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 \( i \leq k \)
  1. Find my link to level \( i - 1 \) and pass it to callee
  2. Its access link will work for lower levels
Two important steps

1. **Compile-time**: 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 ancestor procedure

2. **Run-time**: Given a (level, offset) pair, what’s the address?
   - Two classical approaches:
     ⇒ access link (**static link**)
     ⇒ display
Lexical Scoping

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
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
  begin
    ...
    call B  // occurrence of B
  end
begin
  ...
  call C  // occurrence of C
  call B  // occurrence of B
end

Program
(1,1), (1,2): integer  // declarations of x and y
Procedure (1,3)  // declaration of B
(2,1), (2,2): real  // declaration of y and z
begin
  ...
  (2,1) = (1,1) + (2,2)  // occurrences of y, x, and z
  if (...) call (1,3)  // occurrence of B
end
Procedure (1,4)  // declaration of C
(2,1): real
begin
  ...
  call (1,3)  // occurrence of B
end
begin
  ...
  call (1,4)  // occurrence of C
  call (1,3)  // occurrence of B
end
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.
Access to Non-Local Data (Lexical Scoping): Display

Using a display

Cost of access link is constant (ARP + offset)

- Global arrays of pointers to nameable array
- Needed ARP is an array access away

Example: reference to \(<p, 16>\) looks up p’s APR in display and add 16
Display Management

Single global display:

On entry to a procedure at level i:

- Save the level i display value
- push FP into level i display slot

On return:

- Restore the level i display value

<table>
<thead>
<tr>
<th>parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>register</td>
</tr>
<tr>
<td>save area</td>
</tr>
<tr>
<td>return value</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>saved ptr</td>
</tr>
<tr>
<td>caller’s ARP</td>
</tr>
<tr>
<td>local variables</td>
</tr>
</tbody>
</table>
Procedures

• Modularize program structure
  - Actual parameter: information passed from caller to callee
    \((Argument)\)
  - Formal parameter: local variable whose value (usually) is received from caller

• Procedure declaration
  - Procedure names, formal parameters, procedure body with formal local declarations and statement lists, optional result type

Example: void translate(point *p, int dx)
Parameters

Parameter Association

• Positional association: Arguments associated with formals one-by-one;
  Example: C, Pascal, Java, Scheme
• Keyword association: formal/actual pairs; mix of positional and
  keyword possible;
  Example: Ada
  
  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/C++, Pascal, Java/C# (value types), Scheme
• Pass-by-result: Ada, Algol W
• Pass-by-value-result: Ada, Swift
• Pass-by-reference: Fortran, Pascal, C++, Ruby, ML
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:

Advantage: Argument protected from changes in callee.
Disadvantage: Copying of values takes execution time and space, especially for aggregate values (e.g.: structs, arrays)
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 storage location; hard to track side effects
Pass-by-result

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

Output:

  Program doesn’t compile or has runtime error
begin
  c: array[1...10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := 1; → HERE IS A PROGRAM THAT WORKS
    j := 2;
  end r;
  ...
  m := 5;
  n := 3;
  r(m, n);
  write m, n;
end

Output: ?
begin
  c: array[1...10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := 1;  \textit{\text{HERE IS ANOTHER PROGRAM THAT WORKS}}
    j := 2;
  end r;
...
  m := 5;
  n := 3;
  r(m, m);  \textit{\text{NOTE: CHANGE THE CALL}}
  write m, n;
end

Output: 1 or 2 for m?

\textit{Problem}: order of copy back makes a difference; implementation dependent.
Pass-by-value-result

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 makes a difference; implementation dependent.
Pass-by-value-result

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[3], ... c[10];
end
Output:
Pass-by-value-result

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[3], ... c[10];
end
Output:

*Problem*: When is the address computed for the copy-back operation?
At procedure call (procedure entry), just before procedure exit, or somewhere in between? (Example: ADA on entry)
Pass-by-value-result

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[3], ... 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, or somewhere in between? (Example: ADA on entry)
Pass-by-value-result

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]);   \rightarrow WHAT ELEMENT OF "c" IS ASSIGNED TO?
    write c[1], c[2], c[3], ... 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, or somewhere in between? (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 ←→ formal) and (formal ←→ formal) parameter passing.

```
begin
  j, k, m: integer;
  procedure r(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).

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 (expansive for aggregate values); does not have aliasing, but copy-back order dependence.
Things to do: