Scoping rules

**Scope of a declaration**: Portion of program to which the declaration applies

**Program**

```plaintext
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
    call B      // occurrence of B
  end
Procedure C     // declaration of C
  x: real     // declaration of x
  begin
    ...
    call B     // occurrence of B
  end
begin           // of program
  ...
  call B       // occurrence of B
  ...
end
```

**Lexical scoping (de Bruijn notation)**

Symbol table matches declarations and occurrences.
⇒ Each name can be represented as a pair
  `(nesting_level, local_index)`.

**Program**

```plaintext
(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
    ...
    // occurrences of y, x, and z
    (2,1) = (1,1) + (2,2)
    call (1,3)         // occurrence of B
  end
Procedure (1,4)     // declaration of C
  (2,1): real       // declaration of x
  begin
    ...
    call (1,2)       // occurrence of B
  end
begin           // of program
  ...
  call (1,3)      // occurrence of B
  ...
end
```

---

**Space allocation for code and data**

Lexical scoping ⇒ each name can be represented as a pair `(level, offset)`.

```
<table>
<thead>
<tr>
<th>First try</th>
<th>Second try</th>
</tr>
</thead>
<tbody>
<tr>
<td>prog</td>
<td>code procs, B, C</td>
</tr>
<tr>
<td>x [1, 0]</td>
<td>x [1, 0]</td>
</tr>
<tr>
<td>y [1, 4]</td>
<td>y [1, 4]</td>
</tr>
<tr>
<td>call procB</td>
<td>call procB</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>y [2, 0]</td>
<td>y [2, 0]</td>
</tr>
<tr>
<td>z [2, 4]</td>
<td>z [2, 4]</td>
</tr>
<tr>
<td>(2, 0) = (1,0)+(2,2)</td>
<td>(2, 0) = (1,0)+(2,2)</td>
</tr>
<tr>
<td>call procB</td>
<td>call procB</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>x [2, 0]</td>
<td>y [2, 0]</td>
</tr>
<tr>
<td>call procB</td>
<td>z [2, 4]</td>
</tr>
</tbody>
</table>

control stack
```

How do we know what data object (2,0) represents at run time?

---

**Dynamic scoping — example**

**Program**

```plaintext
integer x = 0
Procedure B (func foo)
  integer x = 1
  begin
    ...
    call foo
    print “In Proc B x= ”, x
  end
Procedure C
  begin
    x = x + 1
  end
begin       // of program
  ...
  call B(C)
  print “In main program x= ”, x
end
```
Important problems

- How do declarations and occurrences of names match?
  - lexical scoping (uses symbol table)
  - dynamic scoping
- How to allocate space for variables and code at run time?
  - static
  - stack (run time stack, calling stack)
  - heap
- How to make sure that the mapping of names into memory locations at run time works correctly, i.e., the run-time environment is always valid?
  - access links
  - display

The procedure abstraction

The procedure abstraction

The essentials:
- on entry, establish p's environment
- at a call, preserve p's environment
- on exit, tear down p's environment
- in between, addressability and proper lifetimes

```
<table>
<thead>
<tr>
<th>procedure p</th>
</tr>
</thead>
<tbody>
<tr>
<td>prolog</td>
</tr>
<tr>
<td>pre-call</td>
</tr>
<tr>
<td>post-return</td>
</tr>
<tr>
<td>epilog</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>procedure q</th>
</tr>
</thead>
<tbody>
<tr>
<td>prolog</td>
</tr>
</tbody>
</table>
```

Each system has a a standard linkage

The procedure abstraction

Separate compilation:
- allows us to build large programs
- keeps compile times reasonable
- requires independent procedures

The linkage convention:
- a social contract
- machine dependent
- division of responsibility

The linkage convention ensures that procedures inherit a valid run-time environment and that they restore one for their parents.

Linkages execute at run time
Code to make the linkage is generated at compile time

Procedure linkages

Assume that each procedure activation has an associated activation record or frame (at run time)

```
<table>
<thead>
<tr>
<th>parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>return value</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>access link</td>
</tr>
<tr>
<td>caller FP</td>
</tr>
<tr>
<td>locals</td>
</tr>
</tbody>
</table>
```

Frame Pointer (FP)

Assumptions:
- call by reference parameter passing
- can always expand an allocated block
- locals stored in frame
Procedure linkages

The linkage divides responsibility between caller and callee

<table>
<thead>
<tr>
<th>Call</th>
<th>Caller</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocate basic frame</td>
<td>save registers, state extend basic frame (for local data)</td>
<td></td>
</tr>
<tr>
<td>evaluate &amp; store params</td>
<td>find static data area initialize locals</td>
<td></td>
</tr>
<tr>
<td>store return address</td>
<td>jump to child</td>
<td></td>
</tr>
<tr>
<td>store FP</td>
<td>set FP for child</td>
<td></td>
</tr>
<tr>
<td>set FP for child</td>
<td>initialize locals</td>
<td></td>
</tr>
<tr>
<td>jump to child</td>
<td>fall through to code</td>
<td></td>
</tr>
</tbody>
</table>

At compile time, we generate the code to do this

At run time, that code manipulates the frame & data areas

Run-time storage organization

To maintain procedure abstractions, the compiler must adopt some conventions to govern memory use.

Code space
- fixed size
- statically allocated  \((\text{link time})\)

Data space
- fixed size data may be statically allocated
- variable size data must be dynamically allocated
- dynamic allocation on stack or heap depending on lifetime of data item (e.g.: variable number of arguments to procedure)

Control stack
- dynamic slice of activation tree  \((\text{ASU p.391})\)
- usually supported in hardware

Run-time storage organization

Typical memory layout

<table>
<thead>
<tr>
<th>Logical Address Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>C 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>S t e h a c k</td>
</tr>
<tr>
<td>free memory</td>
</tr>
</tbody>
</table>

The classical scheme
- allows both stack and heap maximal freedom
- code and static may be separate or intermingled

Run-time storage organization

Where do local variables go?
When can we allocate them on a stack?

Key issue is lifetime of local names

Downward exposure:
- called procedures may reference my variables
- dynamic scoping
- lexical scoping

Upward exposure:
- can I return a reference to my variables?
- functions that return functions

With only downward exposure, the compiler can allocate the frames on the run-time stack
Run-time storage organization

Each variable must be assigned a storage class
(base address for static area, stack, heap)

Static or global variables
- addresses compiled into code (relocatable)
- allocated at compile-time
- limited to fixed size objects

Procedure local variables
Put them on the stack —
- if sizes are fixed, or known at procedure invocation time
- if lifetimes are limited, i.e., values are not preserved

Access to non-local data

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

Real globals
- visible everywhere
- naming convention gives an address

Lexically nested variables
- view variables as (level,offset) pairs (compile-time)
- map level to corresponding activation record or frame

Parameter passing

What about parameters?

Call-by-value
- store values, not addresses
- never restore on return
- arrays, structures, strings are a problem

Call-by-value-result
- store values, not addresses
- always restore on return
- arrays, structures, strings are a problem

Call-by-name
- build and pass thunk
- access to parameter invokes thunk
- all parameters are same size in frame

More about this next time
Parameter passing

What about variable length argument lists?

1. if caller knows that callee expects a variable number
   (a) caller can pass number as 0th parameter
   (b) callee can find the number directly
2. if caller doesn’t know anything about it
   (a) callee must be able to determine number
   (b) first parameter must be closest to FP

Consider printf:

- number of parameters determined by the format string
- it assumes the numbers match

Next lecture

Runtime Environments

- access to non-local variables — access links & displays
- intermediate representations

Please read ASU Chapter 7.4 & 8.1