Names, Scopes, and Binding

Read: Scott, Chapter 3.1, 3.2 and 3.3.1, 3.3.2 and 3.3.6
Lecture Outline

- Notion of binding time
- Object lifetime and storage management
- An aside: Stack Smashing 101

- Scoping
  - Static scoping
  - Dynamic scoping
Scoping

- In most languages the same variable name can be used to denote different memory locations

Scoping rules: map variable to location

Scope: region of program text where a declaration is visible

- Most languages use static scoping
  - Mapping from variables to locations is made at compile time

Block-structured programming languages
  - Nested subroutines (Pascal, ML, Scheme, etc.)
  - Nested blocks (C, C++ { … })
Static Scoping in Block Structured Programming Languages

- Also known as lexical scoping
- Block structure and nesting of blocks gives rise to the closest nested scope rule
  - There are local variable declaration within a block
  - A block inherits variable declarations from enclosing blocks
  - Local declarations take precedence over inherited ones
    - Hole in scope of inherited declaration
    - In other words, inherited declaration is hidden
- Lookup for non-local variables proceeds from inner to outer enclosing blocks
Example - Block Structured PL

```plaintext
main
  a, b, c: integer
  procedure P
    c: integer
    procedure S
      c, d: integer
    end R
    S()
  end P
end main
```

Nested block structure allows locally defined variables and subroutines
main
  a, b, c: integer
  procedure P
    c: integer
    procedure S
      c, d: integer
      procedure R
      ...
    end R
    R()
    end S
    S()
  end P
  procedure R
    a: integer
    ...
      = a, b, c
  end R
  P()
end main

Rule: a variable is visible if it is declared in its own block or in a textually surrounding block and it is not 'hidden' by a binding in a closer block (i.e., there is no hole in scope)
Example with Frames

main
  a, b, c: integer /*1*/
  procedure P /*3*/
    c: integer
    procedure S /*8*/
      c, d: integer
      procedure R /*10*/
      ...
      end R /*11*/
    R() /*9*/
  S() /*7*/
end P /*13*/
procedure R /*5*/
  a: integer
  ... = a, b, c
end R /*6*/
P() /*2*/ ...
end main /*14*/

---

at /*1*/
main
---
---

at /*2*/, main calls
main.P

fp - currently active frame

sp
Example

main

a, b, c: integer /*1*/
procedure P /*3*/
c: integer
procedure S /*8*/
c, d: integer
procedure R /*10*/
...
end R /*11*/
R() /*9*/
end S /*12*/
R() /*4*/
S() /*7*/
end P /*13*/
procedure R /*5*/
a: integer
... = a, b, c
end R /*6*/
P() /*2*/ ...
end main /*14*/
Example

```
main
  a, b, c: integer /*1*/
  procedure P /*3*/
    c: integer
    procedure S /*8*/
      c, d: integer
      procedure R /*10*/
        ...
        end R /*11*/
      R() /*9*/
    end S /*12*/
  end P /*13*/
  procedure R /*5*/
    a: integer
    ... = a, b, c
  end R /*6*/
P() /*2*/ ...
end main /*14*/
```
Example

```r
main
  a, b, c: integer /*1*/
  procedure P /*3*/
    c: integer
    procedure S /*8*/
      c, d: integer
      procedure R /*10*/
      ...
      end R /*11*/
      R() /*9*/
    end S /*12*/
  end P /*13*/
  procedure R /*5*/
    a: integer
    ... = a, b, c
  end R /*6*/
P() /*2*/ ...
end main /*14*/
```

at /*6*/ main.R exits
sp ← fp
fp ← old fp
in main.R’ s frame
Example

main

a, b, c: integer /*1*/
procedure P /*3*/
c: integer
procedure S /*8*/
c, d: integer
procedure R /*10*/
...
end R /*11*/
R() /*9*/
end S /*12*/
R() /*4*/
S() /*7*/
end P /*13*/
procedure R /*5*/
a: integer
... = a, b, c
end R /*6*/
P() /*2*/ ...
end main /*14*/

at /*7*/,
P calls P.S;
at /*8*/:
Example

```plaintext
main
  a, b, c: integer /*1*/
  procedure P /*3*/
    c: integer
    procedure S /*8*/
      c, d: integer
      procedure R /*10*/
        ...
        end R /*11*/
      R() /*9*/
    end S /*12*/
  R() /*4*/
  S() /*7*/
end P /*13*/
procedure R /*5*/
  a: integer
  ... = a, b, c
end R /*6*/
P() /*2*/ ...
end main /*14*/
```

at /*9*/ S calls in S.R; at /*10*/
Example

main

    a, b, c: integer /*1*/
    procedure P /*3*/
        c: integer
        procedure S /*8*/
            c, d: integer
            procedure R /*10*/
                ...
            end R /*11*/
        R() /*9*/
    end S /*12*/
end P /*13*/
procedure R /*5*/
    a: integer
    ... = a, b, c
end R /*6*/
P() /*2*/ ...
end main /*14*/
Example

main
  a, b, c: integer /*1*/
procedure P /*3*/
  c: integer
procedure S /*8*/
  c, d: integer
  procedure R /*10*/
    ...
  end R /*11*/
  R() /*9*/
end S /*12*/
  end S /*12*/
  R() /*4*/
  S() /*7*/
end P /*13*/
  procedure R /*5*/
  a: integer
  ... = a, b, c
end R /*6*/
  P() /*2*/ ...
end main /*14*/

/*12*/pop S’s frame
Example

main
  a, b, c: integer /*1*/
  procedure P /*3*/
    c: integer
    procedure S /*8*/
      c, d: integer
      procedure R /*10*/
        ...
        end R /*11*/
      R() /*9*/
    end S /*12*/
  R() /*4*/
  S() /*7*/
end P /*13*/
procedure R /*5*/
  a: integer
  ... = a, b, c
end R /*6*/
P() /*2*/ ...
end main /*14*/

at /*13*/

/13*/ pop P’ s frame
/14*/ pop main’ s frame
so that sp ← fp
Static Link vs. Dynamic Link

- **Static link** for a frame of subroutine $P$ points to the most recent frame of $P$’s lexically enclosing subroutine
  - Bookkeeping required to maintain the static link
  - If subroutine $P$ is enclosed $k$-levels deep from main, then the length of the static chain that begins at a frame for $P$, is $k$
  - To find non-local variables, follow static chain
- **Dynamic link** points to the caller frame, this is essentially old fp stored on frame
Observations

- Static link of a subroutine P points to the frame of the most recent invocation of subroutine Q, where Q is the lexically enclosing subroutine of P
  - Used to implement static scoping using a display

- Dynamic link may point to a different subroutine’s frame, depending on where the subroutine is called from
For now, we assume languages that do not allow subroutines to be passed as arguments or returned from other subroutines, i.e., subroutines (functions) are third-class values.

- When subroutines (functions) are third-class values, it is guaranteed the static reference environment is on the stack.
- I.e., a subroutine cannot outlive its reference environment.
An Important Note!

- Static scoping rules become more involved in languages that allow subroutines to be passed as arguments and returned from other subroutines, i.e., subroutines (functions) are first class values.

- We will return to scoping later during our discussion of functional programming languages.
Dynamic Scoping

- Allows for local variable declaration
- Inherits non-local variables from subroutines that are live when current subroutine is invoked
  - Use of variable is resolved to the declaration of that variable in the most recently invoked and not yet terminated frame. I.e., lookup proceeds from closest predecessor on stack to furthest
- (old) Lisp, APL, Snobol, Perl
Example

main
procedure Z
    a: integer
    a := 1
    Y()
    output a
end Z
procedure W
    a: integer
    a := 2
    Y()
    output a
end W
procedure Y
    a := 0 /*1*/
end Y
Z()
W()
end main

Which a is modified at /*1*/ under dynamic scoping? Z.a or W.a or both?
main calls Z, Z calls Y, Y sets Z.a to 0.
Example

main

procedure Z
  a: integer
  a := 1
  Y()
  output a
end Z

procedure W
  a: integer
  a := 2
  Y()
  output a
end W

procedure Y
  a := 0; /*1*/
end Y

Z()
W()
end main

main calls W, W calls Y, Y sets W.a to 0.
Static vs. Dynamic Scoping

Static Scoping:
- a bound to R.a,
- b to main.b,
- c to main.c

Dynamic Scoping:
- a bound to R.a,
- b to main.b,
- c to P.c
Dynamic Scoping

- Dynamic scoping is considered a bad idea. Why?

- More on static and dynamic scoping to come!
The End