

# LECTURE 14

Names, Scopes, and Bindings:  
Scopes

# SCOPE

The *scope of a binding* is the textual region of a program in which a name-to-object binding is active.

- Nonspecifically, *scope* is a program region of maximal size in which no bindings change.
- Typically called a *block* – the body of a module, class, subroutine, etc.
- In C and C++, we delimit a block with {...}.

# SCOPE

- *Statically scoped* language: the scope of bindings is determined at compile time.
  - Used by almost all but a few programming languages.
  - More intuitive than dynamic scoping.
  - We can take a C program and know exactly which names refer to which objects at which points in the program solely by looking at the code.
- *Dynamically scoped* language: the scope of bindings is determined at run time.
  - Used in Lisp (early versions), APL, Snobol, and Perl (selectively).
  - Bindings depend on the flow of execution at runtime.

# SCOPE

The set of active bindings at any point in time is known as the *referencing environment*.

- Determined by *scope rules*.
- May also be determined by *binding rules*.
- There are two options for determining the reference environment:
  - *Deep binding*: choice is made when the reference is first created.
  - *Shallow binding*: choice is made when the reference is first used.
  - Relevant for dynamically-scoped languages.

# STATIC SCOPING

The bindings between names and objects can be determined by examination of the program text.

Scope rules of a program language define the scope of variables and subroutines, which is the region of program text in which a name-to-object binding is usable.

- Early Basic: all variables are global and visible everywhere
- Fortran 77: the scope of a local variable is limited to a subroutine; the scope of a global variable is the whole program text unless it is hidden by a local variable declaration with the same variable name.
- Algol 60, Pascal, and Ada: these languages allow nested subroutines definitions and adopt **the closest nested scope rule** – bindings introduced in some scope are valid in all internally nested scopes unless hidden by some other binding to the same name.

# CLOSEST NESTED SCOPE RULE

To find the object referenced by a given name:

- Look for a declaration in the current innermost scope.
- If there is none, look for a declaration in the immediately surrounding scope, etc.
- In each of the function bodies to the right, what bindings are visible?

```
def f1(a1):  
    x = 1  
    def f2(a2):  
        def f3(a3):  
            print "x in f3: ", x  
    def f4(a4):  
        def f5(a5):  
            x = 2
```

# CLOSEST NESTED SCOPE RULE

Some languages allow the outer binding to be accessed by using a scope resolution operator.

- In C++, `::x` refers to the global declaration of `x`.
- In Python, `global x` refers to the global declaration of `x` and `nonlocal x` refers to nested, but non-global declarations.
- In Ada, a name may be prefixed by the scope in which it is declared. `My_proc.x` refers to the declaration of `x` inside of procedure `My_proc`.

# STATIC SCOPING

You can think of built-in and pre-defined objects as being defined in an invisible outer scope which surrounds the scope of global objects.

The search for a binding will terminate at this outermost scope. Because of this, you can even create new global bindings for predefined objects, hiding their default binding (this creates a *hole* in the scope of the default binding).



# STATIC SCOPING

Using static scoping, what will this program print?

```
a:integer
procedure first
  a:=1
procedure second
  a:integer
  first()
procedure main
  a:=2
  second()
  write_integer(a)
```

# STATIC SCOPING

We'll get '1'. The following are the statements we execute in order:

```
a:integer
main()
  a:=2
  second()
    a:integer
    first()
      a:=1 ← Bound to global a
  write_integer(a)
```

```
a:integer
procedure first
  a:=1
procedure second
  a:integer
  first()
procedure main
  a:=2
  second()
  write_integer(a)
```

# STATIC LINKS

In the previous lecture, we saw how we can use offsets from the current frame pointer to access local objects in the current subroutine.

What if I'm referencing a local variable to an enclosing subroutine? How can I find the frame that holds this variable? The order of stack frames will not necessarily correspond to the lexical nesting.

But the enclosing subroutine must appear *somewhere* on the stack as I couldn't have called the current subroutine without first calling the enclosing subroutine.

# STATIC LINKS

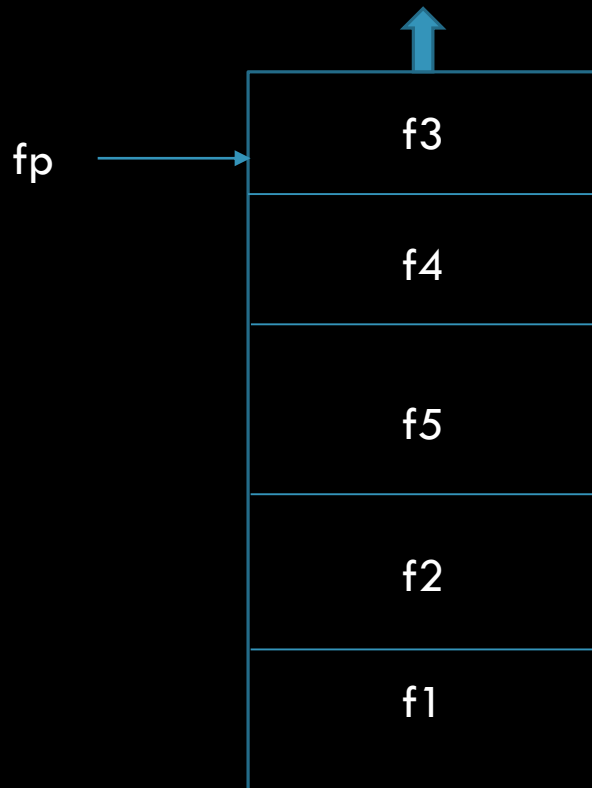
Consider this example. The sequence of function calls is:

1. f1
2. f2
3. f5
4. f4
5. f3

```
def f1():
    x = 1
    def f2():
        print x
        def f3():
            print x
        def f4():
            print x
            f3()
        def f5():
            print x
            f4()
        f5()
    f2()
f1() # executes first!
```

# STATIC LINKS

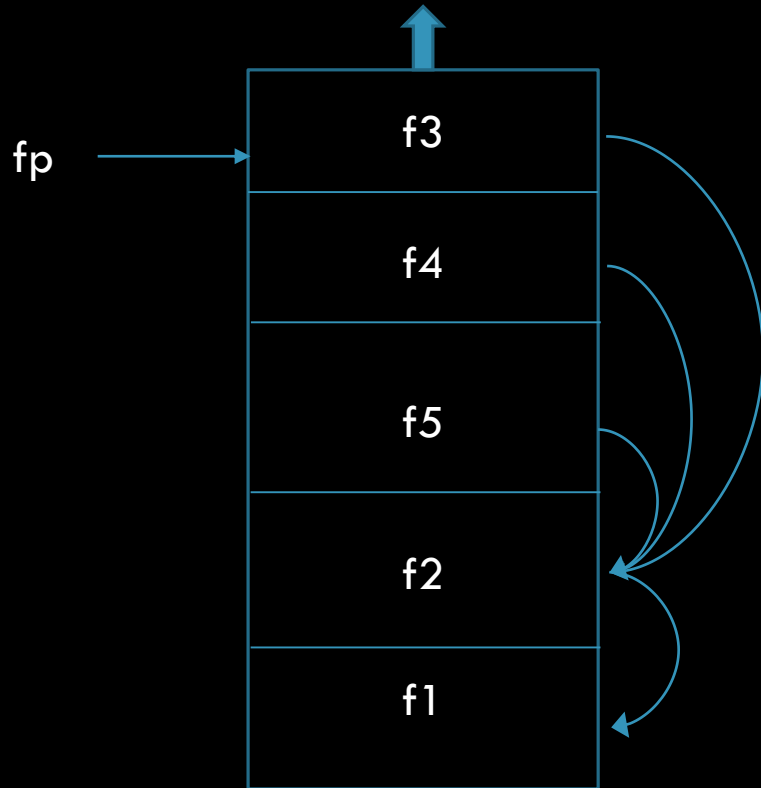
The stack will look like this:



```
def f1():  
    x = 1  
    def f2():  
        print x  
        def f3():  
            print x  
            def f4():  
                print x  
                f3()  
            def f5():  
                print x  
                f4()  
            f5()  
        f2()  
    f1()    # executes first!
```

# STATIC LINKS

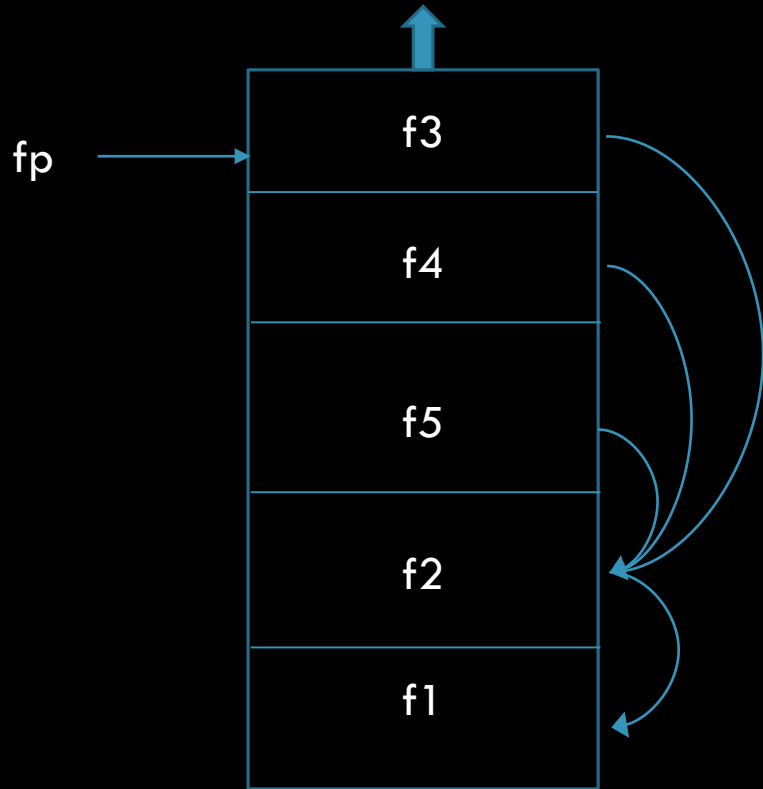
We will maintain information about the lexically surrounding subroutine by creating a static link between a frame and its “parent”.



```
def f1():  
    x = 1  
    def f2():  
        print x  
        def f3():  
            print x  
        def f4():  
            print x  
            f3()  
        def f5():  
            print x  
            f4()  
        f5()  
    f2()  
f1()      # executes first!
```

# STATIC LINKS

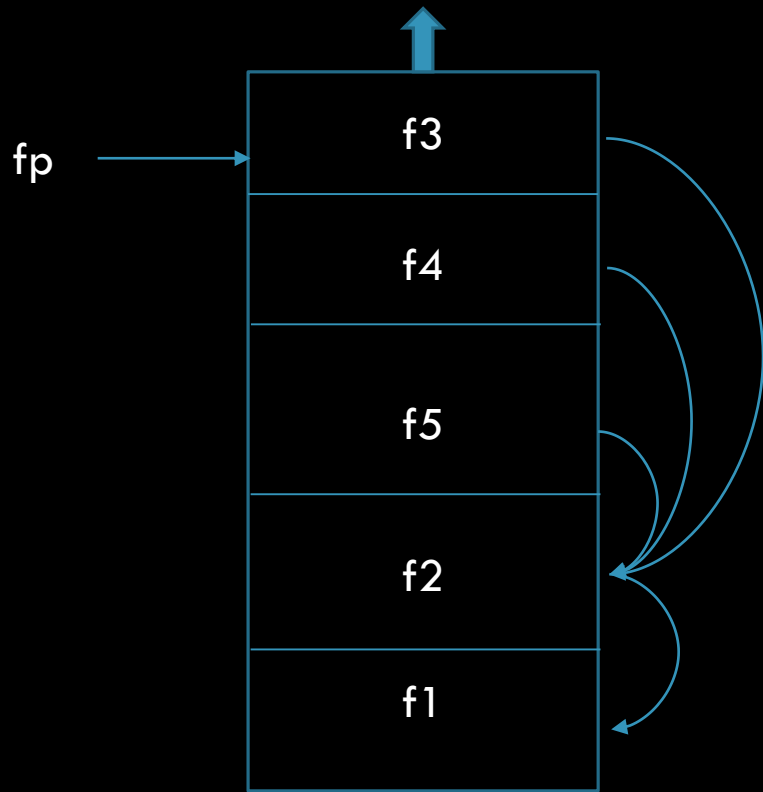
A “parent” is the most recent invocation of the lexically surrounding subroutine. The outermost subroutine has a null static link.



```
def f1():  
    x = 1  
    def f2():  
        print x  
        def f3():  
            print x  
        def f4():  
            print x  
            f3()  
        def f5():  
            print x  
            f4()  
        f5()  
    f2()  
f1()      # executes first!
```

# STATIC LINKS

A subroutine  $k$  levels deep can trace a *static chain* of length  $k$ .

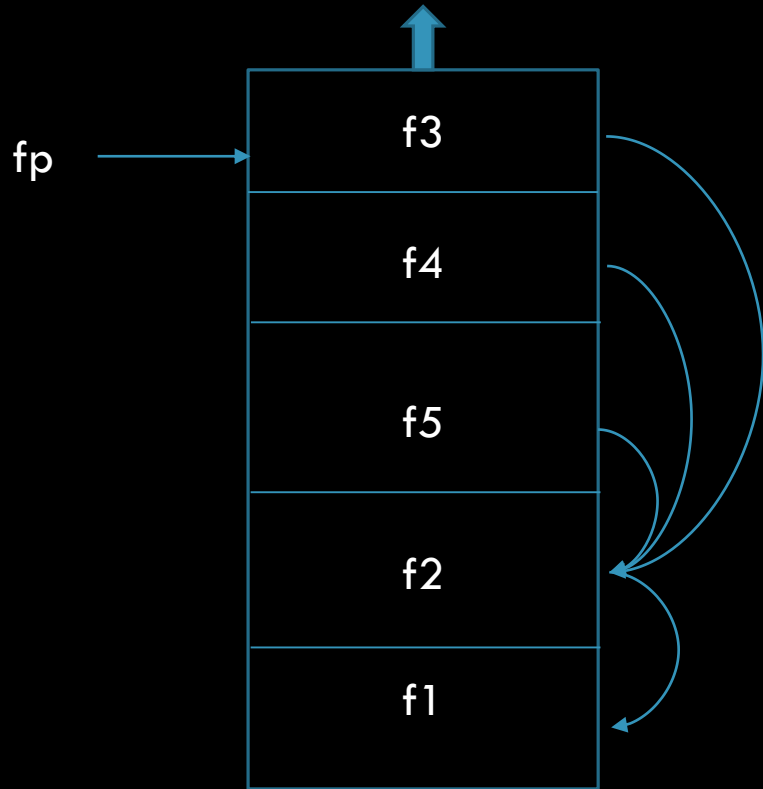


```
def f1():  
    x = 1  
    def f2():  
        print x  
        def f3():  
            print x  
        def f4():  
            print x  
            f3()  
        def f5():  
            print x  
            f4()  
        f5()  
    f2()  
f1()      # executes first!
```



# STATIC LINKS

So, to reference a variable in an enclosing subroutine, we can traverse the static chain until we find its definition.



```
def f1():  
    x = 1  
    def f2():  
        print x  
        def f3():  
            print x  
        def f4():  
            print x  
            f3()  
        def f5():  
            print x  
            f4()  
        f5()  
    f2()  
f1()      # executes first!
```

# DYNAMIC SCOPING

Scope rule: the "current" binding for a given name is the one encountered most recently during execution.

- Typically adopted in (early) functional languages that are interpreted.
- With dynamic scope:
  - Name-to-object bindings cannot be determined by a compiler in general.
  - Easy for interpreter to look up name-to-object binding in a stack of declarations.
- Generally considered to be “a bad programming language feature”.
  - Hard to keep track of active bindings when reading a program text.
  - Most languages are now compiled, or a compiler/interpreter mix.

# DYNAMIC SCOPING


Using dynamic scoping, what will this program print?

```
a:integer
procedure first
  a:=1
procedure second
  a:integer
  first()
procedure main
  a:=2
  second()
  write_integer(a)
```

# DYNAMIC SCOPING

Using dynamic scoping, what will this program print?

```
a:integer
main()
  a:=2
  second()
    a:integer
    first()
      a:=1
    write_integer(a)
```



Bound to most recent a.

Output will be '2'

```
a:integer
procedure first
  a:=1
procedure second
  a:integer
  first()
procedure main
  a:=2
  second()
  write_integer(a)
```

# DYNAMIC SCOPING IMPLEMENTATION

Each time a subroutine is called, its local variables are pushed onto the stack with their name-to-object binding.

When a reference to a variable is made, the stack is searched top-down for the variable's name-to-object binding.

After the subroutine returns, the bindings of the local variables are popped.

Different implementations of a binding stack are used in programming languages with dynamic scope, each with advantages and disadvantages.

Difficulty: when to create the referencing environment for subroutines that can be treated as objects?

# BINDING RULES FOR SUBROUTINES

*Deep binding:* choice is made when the reference is first created.

*Shallow binding:* choice is made when the call is made.

What is the output if we use deep binding?

```
int x = 10;
function f(int a) {
    x = x + a;
}

function g(function h) {
    int x = 30;
    h(100);
    print(x);
}

function main() {
    g(f);
    print(x);
}
```

# BINDING RULES

*Deep binding:* choice is made when the reference is first created.

*Shallow binding:* choice is made when the reference is first used.

What is the output if we use deep binding?

30

110

What is the output if we use shallow binding?

```
int x = 10;
function f(int a) {
    x = x + a;
}

function g(function h) {
    int x = 30;
    h(100);
    print(x);
}

function main() {
    g(f);
    print(x);
}
```

# BINDING RULES

*Deep binding:* choice is made when the reference is first created.

*Shallow binding:* choice is made when the reference is first used.

What is the output if we use deep binding?

30

110

What is the output if we use shallow binding?

130

10

```
int x = 10;
function f(int a) {
    x = x + a;
}

function g(function h) {
    int x = 30;
    h(100);
    print(x);
}

function main() {
    g(f);
    print(x);
}
```



# STATIC AND DYNAMIC SCOPING EXAMPLE

```
a:integer;
procedure foo
  a = 10
  goo()
  hoo()
  write(a)
procedure goo
  a = 20
procedure hoo
  write(a)
procedure main
  a:integer
  a = 30
  foo()
  write(a)
```

What is the output with static scoping? Dynamic scoping with deep bindings? Dynamic scoping with shallow bindings?

# STATIC AND DYNAMIC SCOPING EXAMPLE

```
a:integer;
procedure foo
  a = 10
  goo()
  hoo()
  write(a)
procedure goo
  a = 20
procedure hoo
  write(a)
procedure main
  a:integer
  a = 30
  foo()
  write(a)
```

Static

20

20

30

Dynamic Scoping

20

20

20

# DYNAMIC SCOPING

*Deep binding:* reference environment of `older` is established with the first reference to `older`, which is when it is passed as an argument to `show`.

```
main(p)
  thres:=35
  show(p, older)
    thres:integer
    thres:=20
    older(p)
      return p.age>thres
    if <return value is true>
      write(p)
```

```
thres:integer
function older(p:person):Boolean
  return p.age>thres

procedure show(p:person, c:function)
  thres:integer
  thres:=20
  if c(p)
    write(p)

procedure main(p)
  thres:=35
  show(p, older)
```

# DYNAMIC SCOPING

*Shallow binding:* reference environment of `older` is established with the call to `older` in `show`.

```
main(p)
  thres:=35
  show(p, older)
    thres:integer
    thres:=20
    older(p)
      return p.age>thres
    if <return value is true>
      write(p)
```

```
thres:integer
function older(p:person):Boolean
  return p.age>thres

procedure show(p:person, c:function)
  thres:integer
  thres:=20
  if c(p)
    write(p)

procedure main(p)
  thres:=35
  show(p, older)
```