Scope

CSCI 3136: Principles of Programming Languages

Agenda

- Announcements
  - Assignment 7 is out and due July 12.
- Readings: Read Chapter 3.3
- Lecture Contents
  - Introduction to Scheme
  - Introduction to Scope
  - Lexical Scope
  - Implementation of Lexical Scope
  - Dynamic Scope
  - Shallow vs Deep Binding

Introduction to Scheme

- Scheme is a functional programming language where programs comprise lists and lists comprise atoms
- Every list and atom is an expression
- Computation consists of evaluating the expressions
- An Atom is one of
  - Identifier, e.g., `counter`
  - Value, e.g., `42, “Hello”, #t`
  - List, e.g., `(+ 1 2 3 4)`
  - Quote, e.g., `‘ ( 1 2 3 4) or ‘()`
- Lists are evaluated by
  - Evaluating every atom in the list
  - Evaluating the list of evaluations by treating the first value as a function and applying it to the remaining values
- E.g. `(+ 1 (* 2 3) 4))` evaluates to `11`

Scheme Example

Scheme

```
(define fun (lambda (a b)
      (let ((x '())
            (y '())
            (z '()))
        (set! x (+ a b))
        (let ((a 3)
              (b 4))
          (let ((a 5)
                (b 6))
            (set! y (+ a b)))
          (set! z (+ a b)))
        (list x y z))))
```

```
(fun 1 2)
```

(3 11 7)

Pseudocode

```
def fun( a, b ) {
  var x = nil
  var y = nil
  var z = nil
  x = a + b
  {
    var a = 3
    var b = 4
    {
      var a = 5
      var b = 6
      y = a + b
    }
    z = a + b
  }
  return ( x y z )
}
```

Introduction

- Idea: How we use variables (local/global/etc) depends on the rules of scope!
- Definitions:
  - **Scope of a binding** is the region of a program or time interval(s) in the programs execution during which the binding is active.
  - **A scope** is a maximal region of the program where no bindings are destroyed (e.g., body of a procedure).
- Two generals types of scoping rules:
  - Lexical (static) scoping
  - Dynamic scoping

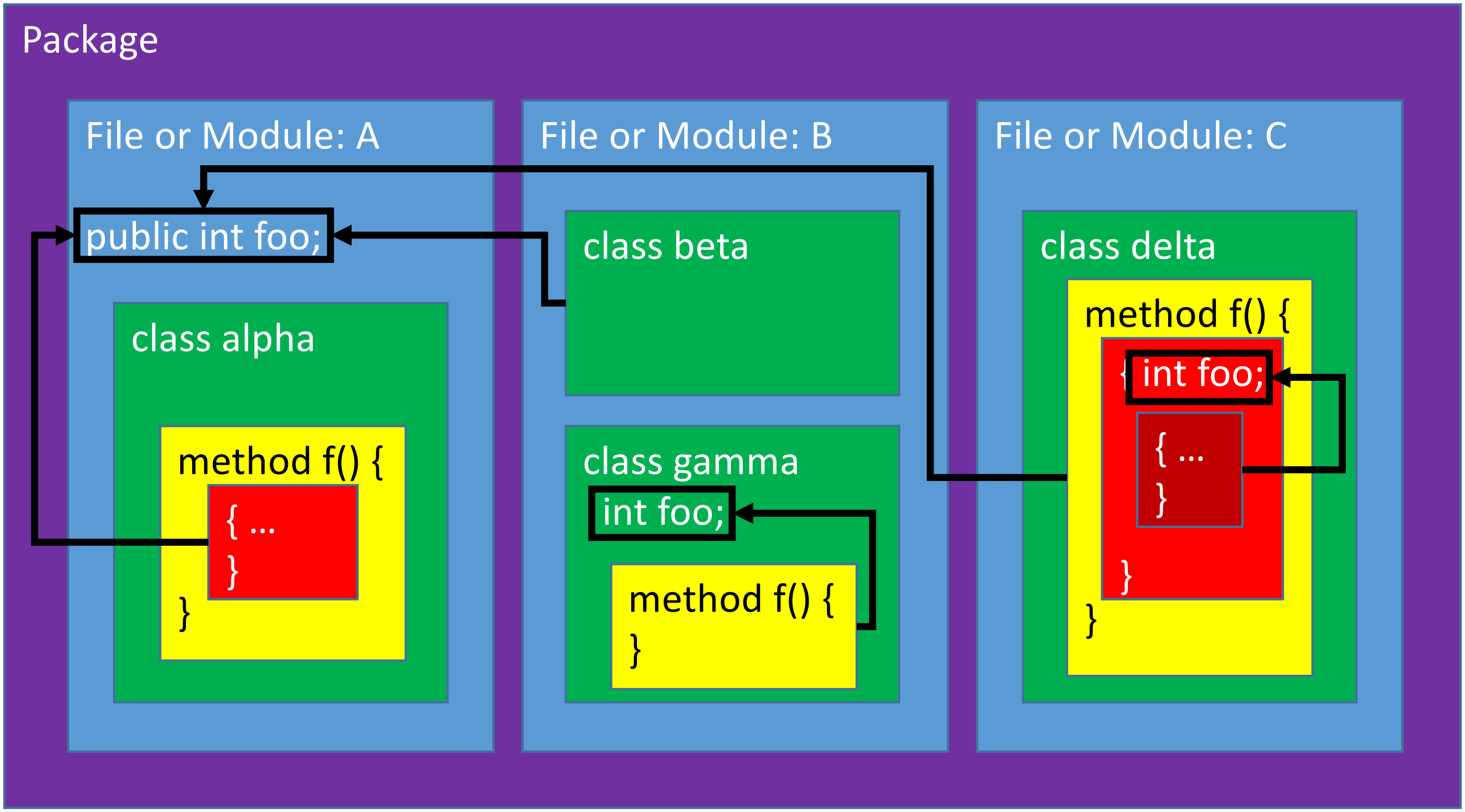
Types of Scoping Rules

- Lexical (static) scoping
  - Binding based on nesting of blocks
  - Can be determined at compile time
  - Is the default for most languages
- Dynamic scoping
  - Binding depends on flow of execution at run time
  - Can be determined only at run time
  - Typically a bad idea
- Questions
  - How do these work?
  - When should we use them?
  - What are the costs?

Lexical Scope

- Idea: Current binding for a name is the one encountered in the smallest enclosing lexical unit.
- Lexical units
  - Packages, modules, source files
  - Classes and nested classes
  - Procedures/methods/subroutines and nested subroutines
  - Blocks
  - Records and structures
- Common Variant: Current binding for a name is
  - The one encountered in the smallest enclosing lexical unit and
  - Preceding the current point in the program text

Lexical Units



Languages that Use Lexical Scope

- **C** requires names to be declared or defined before use
- **Java** requires local variables to be declared before use
- **Prolog** definitions need to be in scope
- **Scheme** has a variety of constructs
  - `(... (define x ...) fun1 fun2 ... funk )`
  - `(lambda (x ...) fun1 fun2 ... funk )`
  - `(let ((x exp1) (y exp2) (z exp3))`
    `fun1 fun2 ... funk )`
  - `(let* ((x exp1) (y exp2) (z exp3))`
    `fun1 fun2 ... funk )`
  - `(letrec ((x exp1) (y exp2) (z exp3))`
    `fun1 fun2 ... funk )`

Scheme: `define`

The **`define`** operator is used to define a variable

```
(define answer 42)
```

- Define a new variable **`answer`** with value 42
- The general format of a define operation is

  (define *identifier* *expression*)

- The variable is visible from the point of definition until the end of the current scope

Scheme: `let`

The **`let`** expression is used to create a new scope and define variables local to that scope

```
(let ((name “Alice”) (quest “Holy Grail”) (weight 42))
   (output (list name quest weight))
)
```

- Defines a new scope with three local variables
- The general format of a **`let`** expression is

```
(let ((id1 expr) (id2 expr) …)
  exprA
  …
  exprX
)
```

- The variables are visible inside `(let …)`
- The result of a **`let`** expression is the result of the last expression
- We will look at lambda, `let*` and `letrec` later.

Scheme Example

Scheme

```
(define fun (lambda (a b)
      (let ((x '())
            (y '())
            (z '()))
        (set! x (+ a b))
        (let ((a 3)
              (b 4))
          (let ((a 5)
                (b 6))
            (set! y (+ a b)))
          (set! z (+ a b)))
        (list x y z))))

(fun 1 2)
```

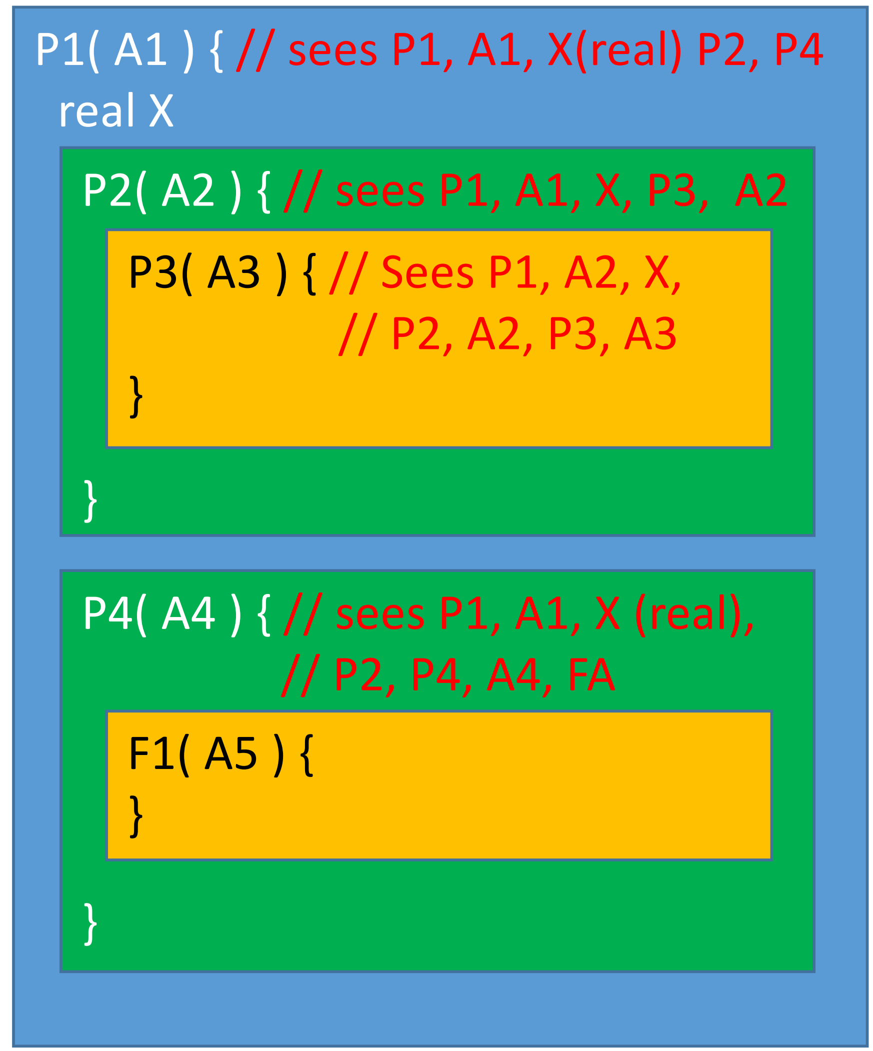
(3 11 7)

Pseudocode

```
def fun( a, b ) {
  var x = ()
  var y = ()
  var z = ()
  x = a + b
  {
    var a = 3
    var b = 4
    {
      var a = 5
      var b = 6
      y = a + b
    }
    z = a + b
  }
  return ( x y z )
}
```

Pascal Example

```
procedure P1( A1 : T1 );
     var X : real;
     procedure P2( A2 : T2 );
       procedure P3( A3 : T3 );
       begin
         ...
       end;
     begin
       ...
     end;
     procedure P4( A4 : T4 );
       function F1( A5: T5) : T6;
       var X : integer;
       begin
       ... end;
     begin
     ... end;
begin
   ...
end;
```



Nested Classes and Functions

- Idea: Many languages support nested classes and functions
- **Nested Classes:** supported in languages like Java
  - Class definitions can contains class definitions, e.g.

    ```
    class Apple {
      ...
      class Seed {
      ...
    ```
  - Inner classes have access to outer classes methods and fields
- **Nested Functions:** supported in languages like Pascal
  - See example in previous slide
  - Inner function has access to everything the outer function does, plus it’s own parameters.

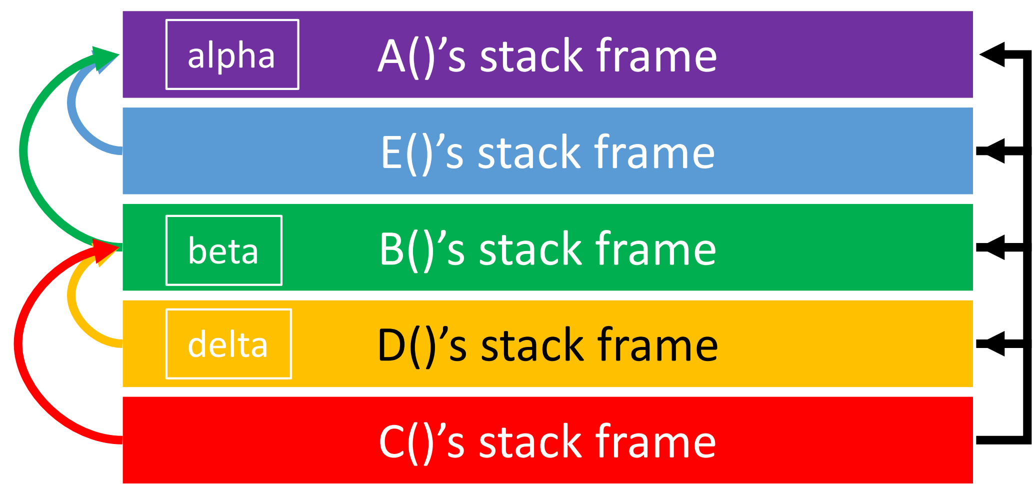
Implementation of Nested Functions in Lexical Scope

- Challenge, inner functions must be able to access the local variables of all outer functions
- Need reference to stack frame of outer functions
- Idea: Store references to outer functions' stack frame in inner functions' stack frame.
  - Why the stack frame? Recursion!

Stack Frame Links Example

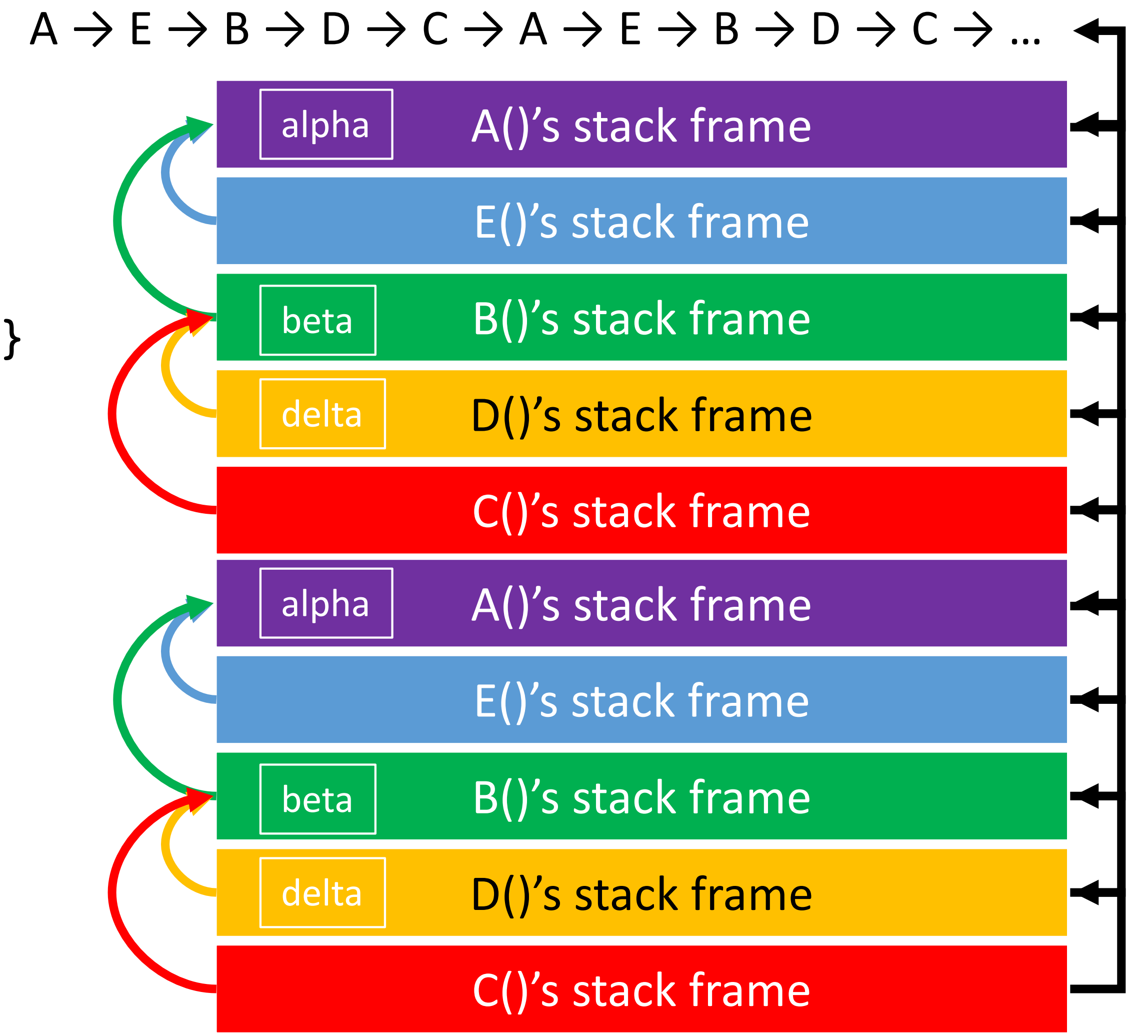
```
func A() {
  int alpha
  func B() {
    int beta
    func C() { … }
    func D() {
      int delta
      … C() …
    }
    … D() …
  }
  func E() {
    … B() …
  }
  … E() …
}
```

- The call chain is:
  - A → E → B → D → C



Why Stack Frames? Recursion!

```
func A() {
  int alpha
  func B() {
    int beta
    func C() { … A() …}
    func D() {
      int delta
      … C() …
    }
    … D() …
  }
  func E() {
    … B() …
  }
  … E() …
}
```



Dynamic Scope

Current binding for a given name is the one

- Encountered most recently during execution
- Not hidden by another binding for the same name
- Not yet destroyed by exiting its scope

- This happens when a language implementation uses a single global reference environment instead of linked environments

Perl Example

```
# Static scoping
sub f {
  my $a = 1;
  print "f:$a\n";
  &printa();
}

sub printa {
  print "p:$a\n";
}

$a = 2;
&f();
```

Output
f:1
p:2

```
# Dynamic scoping
sub g {
  local $a = 1;
  print ″g:$a\n";
  &printa();
}

sub printa {
  print "p:$a\n";
}

$a = 2;
&g();
```

Most recently seen **a**

Output
g:1
p:1

Static vs Dynamic Example

```
a : integer -- global declaration

procedure first
    a := 1

procedure second
    a : integer -- local declaration
    first()

a := 2
second()
write_integer(a)
```

In dynamic scoping, first() will use this variable

Dynamic Scoping is generally a bad idea.

Static	Dynamic
Output is 1	Output is 2

Passing Functions

- Idea: In many languages we can
  - Pass a subroutine/function as a parameter
  - Return a subroutine/function

  I.e. functions are first class objects
- Example:

```
int add( int a, int b ) {
  return a + b;
}

int do_op( (int)(*op)(int,int), int o1, int o2 ) {
  return (*op)( o1, o2 ); /* call op() */
}

...
do_op( &add, 1, 2 );
...
```

Passing Functions (cont)

- Languages that allow passing functions:
  - Scheme
  - C, C++
  - Java (as of Java 8)
  - Many more
- What value does do_op() return? 10 or 16?

```
int add( int a, int b ) {
  return a + b + offset;
}

int do_op( (int)(*op)(int,int), int o1, int o2 ) {
  int offset = 7;
  return (*op)( o1, o2 ); /* call op() */
}

...
int offset = 13;
do_op( &add, 1, 2 );
```

Aside: Free Variables

- Definition: A **free variable** is any variable that is not local or a parameter.

```
int add( int a, int b ) {
  return a + b + offset;
}
```

Shallow vs Deep Binding

- When a subroutine is passed as a parameter, when are its free variables bound?
  - **Shallow binding** occurs when the routine is called
  - **Deep binding** occurs when the routine is first passed as a parameter
- Can happen in both static and dynamic scoping
- Known as the *funarg* problem

Shallow vs Deep Binding Example

```
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 );
}
```

What is the output?

Shallow	Deep
Output is 130 10	Output is 30 110

Shallow or Deep Binding in Scheme?

```
(define increase_x
   (lambda ()
     (set! x (+ x 1))))

(define execute
   (lambda (f)
    (let ((x 20))
     (display (list "inner x before:" x))
     (f)
     (display (list "inner x after:  " x)))))

(define x 1)
(display (list "outer x before:" x))
(execute increase_x)
(display (list "outer x after:  " x))
```

What is the output?

```
(outer x before: 1)
(inner x before: 20)
(inner x after:  20)
(outer x after:  2)
```