

CSC 347 - Concepts of Programming Languages

Scope and Lifetime

Instructor: Stefan Mitsch



Learning Objectives

❓ How should identifiers relate to memory locations?

- Interpret global, static, and dynamic scope
- Identify when shadowing occurs
- Identify bugs related to the difference between the scope of an identifier and the lifetime of a memory location



Scope

- *Scope* of an identifier: region of text in which it may be used

```
1 def f (x: Int) =  
2   val y = x+1  
3   if x>y then  
4     val z = y+1  
5     println(s"z = $x")  
6   end if  
7 end f
```

- `x` and `y` are in scope after their declaration until end of method `f`
- `z` is in scope after its declaration until end of `if`-block



Scope: Free Variables

- Where could `z` come from?

```
1 ...  
2 def g (x:Int) : Int =  
3   var y:Int = x * 2  
4   z * x * y           // x and y are bound; z is free
```



Occurrences of Identifiers

- *Free* occurrence has no matching binding

```
1 y = 5*x;    // Free occurrences of x and y
```

- *Binding* occurrence declares the identifier

```
1 val y : Int;    // binding occurrence of y
```

- *Bound* occurrence follows matching declaration

```
1 var y : Int;    // Binding occurrence of y
2 var x : Int;    // Binding occurrence of x
3
4 x = 6;          // Bound occurrence of x
5 y = 5*x;        // Bound occurrences of x and y
```



Binding and Circular Dependencies

? What to do with circular dependencies?

```
1 char f (int x) { return x>0 ? g (x-1) : 1; }
```

```
1 char g (int x) { return f (x) + f (x); }
```

- Most modern languages allow any order
- C, C++ require *forward declarations*

```
1 char f (int x);  
2 char g (int x);  
3 // f and g definitions can now be in any order
```



Formal Semantics for Functions Revisited

- Global store ξ for variables and their values
- Function store ϕ : maps function names to argument names and function bodies
- Local store ρ for arguments and local variables

Call-by-value

$$\begin{array}{c}
 \langle e_1, \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_1, \xi_1, \phi_1, \rho_1 \rangle \\
 \vdots \\
 \langle e_n, \xi_{n-1}, \phi, \rho_{n-1} \rangle \Downarrow \langle v_n, \xi_n, \phi_n, \rho_n \rangle
 \end{array}
 \left. \vphantom{\begin{array}{c} \langle e_1, \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_1, \xi_1, \phi_1, \rho_1 \rangle \\ \vdots \\ \langle e_n, \xi_{n-1}, \phi, \rho_{n-1} \rangle \Downarrow \langle v_n, \xi_n, \phi_n, \rho_n \rangle \end{array}} \right\} \text{evaluate arguments } e_n$$

$$\begin{array}{c}
 \phi(f) = (x_1, \dots, x_n, b) \\
 \text{look up function}
 \end{array}$$

$$\frac{
 \begin{array}{c}
 \langle b, \xi_n, \phi, \{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\} \rangle \Downarrow \langle v, \xi', \phi', \rho' \rangle \\
 \text{evaluate function body } b
 \end{array}
 }{
 \langle f(e_1, \dots, e_n), \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v, \xi', \phi, \rho_n \rangle
 } \text{(FunCBV)}$$

? Where to look up variables?

$$\overline{\langle x, \xi, \phi, \rho \rangle \Downarrow \langle \xi(x), \xi, \phi, \rho \rangle} \text{(Var)}$$



Global Scope: Python

- Python `global` accesses the global scope

```
1 def useX():  
2     print (x)  
3  
4 def defX():  
5     global x  
6     x = 1
```

```
1 >>> useX()  
2 Traceback (most recent call last):  
3   File "<stdin>", line 1, in <module>  
4   File "<stdin>", line 2, in useX  
5 NameError: name 'x' is not defined  
6  
7 >>> defX()  
8 >>> useX()  
9 1
```



Local-Then-Global Scope

? What does this program do, using Scala *syntax*, but various different *semantics*?

```
1 var x: Int = 1
2
3 def g() =
4   var x: Int = 2
5
6   def f() =
7     x = 3
8   end f
9
10  f(); println(s"(A) = $x")
11  x = 2
12
13  def h() =
14    var x: Int = 4
15    f(); println(s"(B) = $x")
16  end h
17
18  h(); println(s"(C) = $x")
19 end g
20
```



Dynamic Scope

? What does this program do, using Scala *syntax*, but various different *semantics*?

```
1 var x: Int = 1
2
3 def g() =
4   var x: Int = 2
5
6   def f() =
7     x = 3
8   end f
9
10  f(); println(s"(A) = $x")
11  x = 2
12
13  def h() =
14    var x: Int = 4
15    f(); println(s"(B) = $x")
16  end h
17
18  h(); println(s"(C) = $x")
19 end g
20
```



Static Scope

? What does this program do, using Scala *syntax*, but various different *semantics*?

```
1 var x: Int = 1
2
3 def g() =
4   var x: Int = 2
5
6   def f() =
7     x = 3
8   end f
9
10  f(); println(s"(A) = $x")
11  x = 2
12
13  def h() =
14    var x: Int = 4
15    f(); println(s"(B) = $x")
16  end h
17
18  h(); println(s"(C) = $x")
19 end g
20
```



Static and Dynamic Scope

Static Scope

- Compile time scoping, also known as **lexical scope**
- Identifiers are bound to the closest binding occurrence in an **enclosing block** of the program code
- Renaming any identifier to a fresh name consistent throughout its scope does not change program behavior

Dynamic Scope

- Runtime scoping
- Identifiers are bound to the binding occurrence in the **closest activation record**
- Consistent renaming may break a working program!



Static Scope: Scala

- Scala uses static scope
- Most languages do use static scope: Java, C, C++, Python (unless `global`) etc.

```
1 var x:Int = 10      // x0
2
3 def f () : Unit =
4   x = 20             // x is bound statically to x0
5
6 def g () : Unit =
7   var x:Int = 30.    // x1
8   f ()
9
10 g ()
11 println (x) // prints 20
```



Dynamic Scope: Bash

- Bash uses dynamic scope

```
1 x=10          # x0
2
3 function f() {
4     x=20      # x is dynamically bound to x1
5 }
6
7 function g() {
8     local x=30 # x1
9     f
10 }
11
12 g
13 echo $x # prints 10
```



Dynamic Scope to Static Scope

```
1 (let ((x 10))
2   (defun f ()
3     (setq x 20))
4   (defun g ()
5     (let ((x 30))
6       (f)))
7   (g)
8   x
9 )
```

- Dynamic Scope in Emacs Lisp: `x=10` at the end
- Static Scope in Common Lisp: `x=20` at the end



Static vs. Dynamic Scope: Perl

- Dynamic Scope in Perl: `local`

```
1 local $x = 10;
2 sub f {
3     $x = 20;
4 }
5 sub g {
6     local $x = 30;
7     f ();
8 }
9 g ();
10 print ($x); # prints 10
```

- Static Scope in Perl: `my`

```
1 my $x = 10;
2 sub f {
3     $x = 20;
4 }
```



Shadowing in Statically Scoped Languages

- Static scope at the level of blocks
 - ❓ Should reusing names be allowed?

```
1 static void f () {  
2     int x = 1;  
3     {  
4         int y = x + 1;  
5         {  
6             int x = y + 1;  
7             System.out.println ("x = " + x);  
8         }  
9     }  
10 }
```

- In Java, shadowing of local variables is not allowed, see [Java Language Specification](#)

```
1 $ javac C.java  
2 C.java:7: error: variable x is already defined in method f()  
3     int x = y + 1;
```



Shadowing

- In Java, local variables are allowed to shadow fields

```
1 public class C {  
2     static int x = 1;  
3  
4     static void f () {  
5         int y = x + 1;  
6         {  
7             int x = y + 1;  
8             System.out.println ("x = " + x);  
9         }  
10    }  
11  
12    public static void main (String[] args) {  
13        f ();  
14    }  
15 }
```

```
1 $ javac C.java
```



Shadowing

- C allows shadowing of variables

```
1 int main () {  
2     int x = 1;  
3     {  
4         int y = x + 1;  
5         {  
6             int x = y + 1;  
7             printf ("x = %d\n", x);  
8         }  
9     }  
10 }
```

```
1 $ gcc -o scope scope.c  
2 $ ./scope  
3 x = 3
```

- Scala allows shadowing of variables



Shadowing and Recursion

? Is `x` in scope?

- C warns about uninitialized variables

```
1 int main (void) {  
2     int x = 10;  
3     {  
4         int x = x + 1;  
5         printf ("x = %08x\n", x);  
6     }  
7     return 0;  
8 }
```

```
1 $ gcc -o scope scope.c  
2  
3 $ gcc -Wall -o scope scope.c  
4 scope.c: In function 'main':  
5 scope.c:5:7: warning: unused variable 'x' [-Wunused-variable]  
6 scope.c:7:9: warning: 'x' is used uninitialized in this function [-Wuninitialized]  
7
```



Shadowing and Recursion

- Scala variables and fields are set to default values (e.g., 0) before the initialization code is run
- Recursion is allowed when initializing fields

Scala

```
1 scala> val x:Int = 1 + x
2 x: Int = 1
```

Expressed in Java

```
1 public class C {
2     // default-initialized to 0
3     private final int x;
4     public int x() { return x; }
5     public C() { x = 1 + x; }
6 }
```



Shadowing and Recursion

? Does that work with complex datatypes?

```
1 val xs:List[Int] = 1 :: xs
2 // java.lang.NullPointerException
```

- `xs` default-initialized to `null`
- `null != Nil` : exception occurs because `1 :: null` is `null.::(1)`



Scala Lazy Lists

- Lazy lists `#::` are non-strict

```
1 val ones: LazyList[Int] = 1 #:: ones
2 // ones: LazyList[Int] = LazyList(<not computed>)
```

```
1 scala> ones.take (5)
2 res0: scala.collection.immutable.LazyList[Int] = LazyList(<not computed>)
3
4 scala> ones.take (5).toList
5 res1: List[Int] = List(1, 1, 1, 1, 1)
```



Scala Lazy Lists

- *Lazy* evaluation of list elements

```
1 def f (x:Int) : LazyList[Int] =  
2   println (s"Called f($x)")  
3   x #:: f(x+1)  
4 end f  
5 // f: (x: Int): LazyList[Int]
```

- Create a lazy list

```
1 scala> val xs:LazyList[Int] = f(10)  
2 Called f(10)  
3 xs: LazyList[Int] = LazyList(<not computed>)
```

- Access elements of lazy list

```
1 scala> xs.take(4).toList  
2 Called f(11)  
3 Called f(12)
```



Recursion and Corecursion

Recursion

- Works **analytically**: breaks computation into smaller pieces until it reaches a base case
- Operates on arbitrarily complex data as long as they can be reduced to simple base cases

Corecursion

- Works **synthetically**: builds up data from a base case
- Generates arbitrarily complex data as long as it can be produced from simple base cases
- Often implemented with lazy evaluation



Example: Recursive and Corecursive Factorial

Recursive

```
1 def fact(n: Int) : BigInt =  
2   require(n>=0)  
3   if n == 0 then 1  
4   else n * fact(n-1)
```

Tail-Recursive

```
1 def fact(n: Int, f: Int=1) : BigInt =  
2   require(n>=0)  
3   if n == 0 then f  
4   else fact(n-1, n*f)
```

Corecursive

```
1 def fact(n: Int=0,  
2         f: Int=1) : LazyList[BigInt] =
```



Scope and Lifetime

Scope

- Of an identifier in source code
- Where it is bound

Lifetime

- Of an area of memory
- Duration during which it is allocated



Storage Options

Global

- Static storage
- Available for lifetime of program

Call Stack

- *In activation records in call stack* (stack-allocated, recall Systems 1)
- Available whilst function active (called but not returned)

Heap



Lifetime Issues

- ✖ Lifetime too short: common with explicit memory de-allocation
 - reads return other value
 - writes overwrite other value
 - resource state incorrect, e.g., file handle closed
 - can cause security problems
- ✖ Lifetime too long: common with garbage collection
 - uses too much memory (*memory leak*)
 - too late in freeing other resources / finalization
 - can cause vulnerability to denial of service attacks



Dangling Pointers: Stack

🔪 What is wrong with this program?

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int *f (int x) {
5     int y = x;
6     return &y;
7 }
8
9 int main (void) {
10     int *p = f (1);
11     printf ("*p = %d\n", *p);
12     return 0;
13 }
```

- Compile warning

```
1 $ gcc -o ar ar.c
2 ar.c: In function 'f':
```



Dangling Pointers

- Static analysis tools can help

```
1 $ valgrind ./ar
2 ==5505== Memcheck, a memory error detector
3 ==5505== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
4 ==5505== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
5 ==5505== Command: ./ar
6 ==5505==
7 ==5505== Conditional jump or move depends on uninitialised value(s)
8 ==5505==    at 0x4E7C1A1: vfprintf (vfprintf.c:1596)
9 ==5505==    by 0x4E85298: printf (printf.c:35)
10 ==5505==    by 0x400536: main (in /tmp/ar)
11 ==5505==
12 ==5505== Use of uninitialised value of size 8
13 ==5505==    at 0x4E7A49B: _itoa_word (_itoa.c:195)
14 ==5505==    by 0x4E7C4E7: vfprintf (vfprintf.c:1596)
15 ==5505==    by 0x4E85298: printf (printf.c:35)
16 ==5505==    by 0x400536: main (in /tmp/ar)
17 ==5505==
18 ==5505== Conditional jump or move depends on uninitialised value(s)
19 ==5505==    at 0x4E7A4A5: _itoa_word (_itoa.c:195)
20 ==5505==    by 0x4E7C4E7: vfprintf (vfprintf.c:1596)
21 ==5505==    by 0x4E85298: printf (printf.c:35)
22 ==5505==    by 0x400536: main (in /tmp/ar)
23 ==5505==
```



Dangling Pointers: Heap

✂ What is wrong with this program?

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int *f (int x) {
5     int *result = (int *) malloc (sizeof (int));
6     *result = x;
7     return result;
8 }
9
10 int main (void) {
11     int *p = f (1);
12     printf ("*p = %d\n", *p);
13     return 0;
14 }
```

- Program compiles



Dangling Pointers: Heap

```
1 $ valgrind ./ar
2 ==10962== Memcheck, a memory error detector
3 ==10962== Copyright (C) 2002-2011, and GNU GPL, by Julian Seward et al.
4 ==10962== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
5 ==10962== Command: ./ar
6 ==10962==
7 *p = 1
8 ==10962==
9 ==10962== HEAP SUMMARY:
10 ==10962==      in use at exit: 4 bytes in 1 blocks
11 ==10962==    total heap usage: 1 allocs, 0 frees, 4 bytes allocated
12 ==10962==
13 ==10962== LEAK SUMMARY:
14 ==10962==    definitely lost: 4 bytes in 1 blocks
15 ==10962==    indirectly lost: 0 bytes in 0 blocks
16 ==10962==    possibly lost: 0 bytes in 0 blocks
17 ==10962==    still reachable: 0 bytes in 0 blocks
18 ==10962==           suppressed: 0 bytes in 0 blocks
19 ==10962== Rerun with --leak-check=full to see details of leaked memory
20 ==10962==
21 ==10962== For counts of detected and suppressed errors, rerun with: -v
22 ==10962== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```



Dangling Pointers: Heap

🔪 What is wrong with this program?

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int *f (int x) {
5     int *result = (int *) malloc (sizeof (int));
6     *result = x;
7     return result;
8 }
9
10 int main (void) {
11     int *p = f (1);
12     free (p);
13     printf ("*p = %d\n", *p);
14     return 0;
15 }
```

- Program compiles



Dangling Pointers: Heap

```
1 $ valgrind ./ar
2 ==13594== Memcheck, a memory error detector
3 ==13594== Copyright (C) 2002-2011, and GNU GPLd, by Julian Seward et al.
4 ==13594== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
5 ==13594== Command: ./ar
6 ==13594==
7 ==13594== Invalid read of size 4
8 ==13594==    at 0x4005D2: main (in /tmp/ar)
9 ==13594==    Address 0x51f0040 is 0 bytes inside a block of size 4 freed
10 ==13594==    at 0x4C2A82E: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
11 ==13594==    by 0x4005CD: main (in /tmp/ar)
12 ==13594==
13 *p = 1
14 ==13594==
15 ==13594== HEAP SUMMARY:
16 ==13594==    in use at exit: 0 bytes in 0 blocks
17 ==13594==    total heap usage: 1 allocs, 1 frees, 4 bytes allocated
18 ==13594==
19 ==13594== All heap blocks were freed -- no leaks are possible
20 ==13594==
21 ==13594== For counts of detected and suppressed errors, rerun with: -v
22 ==13594== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
```



Summary

Scope

- Where an identifier is visible
- Global scope: variables are accessible from anywhere
- Static scope: closest lexical appearance in source code
- Dynamic scope: closest activation record

Lifetime

- How long a memory location is available
-  Dangling pointers: point to freed memory
-  Memory leaks: allocated memory that does not get freed