

CSC 347 - Concepts of Programming Languages

Closures

Instructor: Stefan Mitsch



Learning Objectives

❓ What if functions have free variables?

- Identify when a function has free variables
- Identify nested methods and functions
- Describe the effect of closures: how free variables are bound to variables of the environment



Nested Functions

- Useful to limit the scope of functions and to hide arguments
- Allowed by GCC, but not C standard
- Allowed in Scala

```
1 int fact (int n) {  
2     int loop(int n, int result) {  
3         if (n <= 1) {  
4             return result;  
5         } else {  
6             return loop(n - 1, n * result);  
7         }  
8     }  
9     return loop(n, 1);  
10 }
```

```
1 $ gcc -c nested-fact.c  
2 $ gcc -pedantic -c nested-fact.c  
3 function.c: In function 'fact':  
4 function.c:2:3: warning: ISO C forbids nested functions [-pedantic]
```

```
1 def fact(n:Int) : Int =  
2     def loop(n:Int, result:Int) : Int =  
3         if n<=1 then result  
4         else loop(n-1, n*result)  
5     end loop  
6     loop(n, 1)  
7 end fact
```



Nested Functions: Java and Scala

- Closures: nested functions in Scala and Java store environment on function definition

Scala

```
1 def f(x:Int) : Int=>Unit =  
2   def g(y:Int) = println(s"x=$x, y=$y")  
3   g // return function g, which accesses x  
4 end f  
5  
6 val h1 = f(10)  
7 h1(2)           // prints x=10, y=2  
8 val h2 = f(20)  
9 h1(3)           // prints x=10, y=3  
10 h2(3)           // prints x=20, y=3
```

• Java

```
1 import java.util.function.Function;  
2  
3 public static Function<Integer,Void> f(int x) {  
4     Function<Integer,Void> g = y -> {  
5         System.out.format ("x=%d, y=%d\n", x, y);  
6         return null;  
7     }  
8     return g;  
9 }  
10 public static void main (String[] args) {  
11     Function<Integer,Void> h1 = f(10);  
12     h1.accept(2);           // prints x=10, y=2  
13     Function<Integer,Void> h2 = f(20);  
14     h1.accept(3);           // prints x=10, y=3  
15     h2.accept(3);           // prints x=20, y=3  
16 }
```



Closures: Problem Summary

```
1 def outer(x:A) : B=>C =  
2   def inner(y:B) : C =  
3     //...use x and y...  
4   end inner  
5   inner // return inner  
6 end outer  
7  
8 val f = outer(...)  
9 f(...)
```

1. Function `outer` is called
 - AR contains data `x`
2. Function `outer` returns nested function `inner`
 - Function `inner` has free variable `x`
 - $\Rightarrow$ resolve with `x` from `outer`'s AR
3. Lifetime of `outer`'s AR and `x` ends
4. Nested function `inner` is called through alias `f`
 - Function `inner` needs `x` from `outer`'s AR

❓ Mutable vs. immutable state?



Closures: Copy or Share

```
1 def outer(x:A) : B=>C =  
2   def inner(y:B) : C =  
3     // use x and y  
4     // free: x  
5     // bound: y  
6   end inner  
7   inner // return inner  
8 end outer
```

Closure contains

- Code for `inner`
- Copy of `x`



Closures: Copy or Share

```
1 def outer(x:A) : B=>C =  
2   var u:A = x  
3   def inner(y:B) : C =  
4     // use x, u, and y:  
5     // free: x, u  
6     // bound: y  
7   end inner  
8   u = u + 1  
9   inner // return inner  
10 end outer
```

Closure contains

- Code for `inner`
- Copy of `x`
- What to do with `u` ?
 - `inner` sees updated `u` ?
 - Scala
 - require `u` to be immutable?
 - Java



Closures: Scala

- `inner` has free `x`
- Needs closure

```
1 def outer(x:Int) : Boolean=>Int =  
2   def inner(y:Boolean) : Int =  
3     x + (if y then 0 else 1)  
4   end inner  
5   inner // return inner  
6 end outer
```

- Object-oriented implementation in Java

```
1 public static Function1<Boolean, Integer> outer(int x) {  
2   return new Closure(x);  
3 }  
4  
5 public final class Closure  
6   extends AbstractFunction1<Boolean, Integer> {  
7  
8   private final int x;  
9  
10  public final Integer apply(Boolean y) {  
11    return x + (y ? 0 : 1);  
12  }  
13  
14  public Closure(int x) { this.x = x; }  
15 }
```



Closures: Scala

- `inner` has free `x`, `u`
- Needs closure
- Object-oriented impl. stores mutable `u` on heap

```
1 def outer (x:Int) : Boolean=>Int =  
2   var u:Int = x  
3   def inner (y:Boolean) : Int =  
4     x + u + (if y then 0 else 1)  
5   end inner  
6   u = u+1  
7   inner // return inner  
8 end outer
```

```
1 public static Function1<Boolean, Integer> outer(int x) {  
2   IntRef u = new IntRef(x);  
3   var c = new Closure(x, u);  
4   u.elem = u.elem+1;  
5   return c;  
6 }  
7  
8 public final class Closure  
9   extends AbstractFunction1<Boolean, Integer> {  
10  private final int x;  
11  private final IntRef u;  
12  public final Integer apply(Boolean y) {  
13    return x + u.elem + (y ? 0 : 1);  
14  }  
15  public Closure(int x, IntRef u) {  
16    this.x = x;  
17    this.u = u;  
18  }  
19 }
```



Example: Use Closures

How to create a "container" for data and functions?

Object-oriented programming: classes

```
1 public class Incrementor {
2     private int i;
3     public Incrementor(int i) {
4         this.i = i;
5     }
6     public int increment(int x) {
7         return x+i;
8     }
9 }
10 // use object
11 Incrementor inc = new Incrementor(2);
12 inc.increment(4); // returns 6
13 inc.increment(5); // returns 7
```

? Functional programming

```
1 def incrementor(i:Int) : Int=>Int =
2     (x:Int) => x+i
3 end incrementor
4
5
6
7
8
9
10 // use closure
11 val inc = incrementor(2)
12 inc(4) // returns 6
13 inc(5) // returns 7
```

- Useful for languages without string visibility modifiers (Python, JavaScript)



Example: Interpret Closures

- Closure for method

```
1 val f: ()=>Int =  
2   var x = -1  
3   def g() =  
4     x = x + 1  
5     x  
6   end g  
7   g
```

- Closure for function

```
1 val f: ()=>Int =  
2   var x = -1  
3   () => {  
4     x = x + 1  
5     x  
6   }
```

- Initializes:

- `x` to `-1` when initializing variable `f`

- Returns:

- incremented `x` on every call `f()`

```
1 scala> f()  
2 res0: Int = 0  
3 scala> f()  
4 res1: Int = 1
```



Example: Interpret Closures

- Closure for method

```
1 def g(y: Int) : ()=>Int =  
2   var z = y  
3   def h() =  
4     z = z + 1  
5     z  
6   end h  
7   h
```

- Closure for function

```
1 val g: Int=>()=>Int =  
2   y => {  
3     var z = y  
4     () => {  
5       z = z + 1  
6       z  
7     }  
8   }
```

- `g(i)` returns:

- function `()=>Int`, its own `z` set to `i`

```
1 scala> val h1=g(10)  
2 h1: () => Int = $$Lambda$1098/39661414@54d8c20d  
3 scala> val h2=g(20)  
4 h2: () => Int = $$Lambda$1098/39661414@5bc7e78e
```

- `h1()` and `h2()` return:

- their own incremented `z`

```
1 scala> h1()  
2 res3: Int = 11  
3 scala> h1()  
4 res4: Int = 12  
5 scala> h2()  
6 res5: Int = 21  
7 scala> h1()  
8 res6: Int = 13
```



Example: Use Closures for Partial Functions

>_ Implement a function that prints a greeting

Scala

```
1 val greet = greeter("Hello")
2 println(greet("Alice"))
3 println(greet("Bob"))
```

- Return a function for printing that accesses the outer greeting argument

```
1 def greeter(greeting: String) : String=>String =
2   (name: String) => s"$greeting $name!"
```

JavaScript

- Event handlers with preserved state

```
1 const button1 = document.getElementById("button1");
2 const button2 = document.getElementById("button2");
3
4 const greet = createGreeter("Hello");
5 button1.addEventListener("click", greet("Alice"));
6 button2.addEventListener("click", greet("Bob"));
```

```
1 function createGreeter(greeting) {
2   function greet(name) {
3     return function() {
4       console.log(greeting + " " + name + "!");
5     }
6   }
7   return greet;
8 }
```



Example: Use Closures for Stateful Functions

➤ Implement a function that collects numbers and computes their average:

```
1 val cavg = cumulativeAvg()  
2 cavg(2) // returns 2  
3 cavg(4) // returns 3  
4 cavg(6) // returns 4
```

- Initialize a list buffer, return a function that appends and then computes the average

```
1 def cumulativeAvg(): Int=>Int =  
2   val data = scala.collection.mutable.ListBuffer.empty[Int]  
3   x =>  
4     data.append(x)  
5     data.sum / data.length  
6 end cumulativeAvg
```



Example: Use Closures for Decorators

➤ Implement a function to cache the results of another function:

Scala

```
1 def square(x: Int) = { println(s"$x^2"); x*x }
2 val cachedSquare = memoize(square)
3 cachedSquare(5) // prints 5^2 and returns 25
4 cachedSquare(5) // returns 25 without printing
```

- Initialize cache, return function that consults and updates cache

```
1 def memoize[X,Y](fn: X=>Y) : X=>Y =
2   val cache = scala.collection.mutable.Map.empty[X,Y]
3   x => cache.getOrElseUpdate(x, fn(x))
```

Python

```
1 def memoize(function):
2     cache = {}
3     def closure(number):
4         if number not in cache:
5             cache[number] = function(number)
6         return cache[number]
7     return closure
```

- Use decorator

```
1 @memoize
2 def square(x):
3     print(f"{x}^2")
4     return x*x
```



Example: Use Closures for Data Encapsulation

➤ Implement a map data structure from arrays and index access; wanted use:

```
1 val (get, put) = map(10)
2 put(3, "Hello")
3 put(6, "world!")
4 println(get(3))
```

- Return a tuple of functions that access a shared array

```
1 def map(size: Int) : (Int=>String, (Int, String)=>Unit) =
2   val elems: Array[String] = Array.ofDim(size)
3
4   def get(k: Int)           = elems(k)
5   def put(k: Int, v: String) = elems(k) = v
6
7   (get, put)
8 end map
```



Example: Debouncing and Throttling

- Debouncing: delayed execution of a function (e.g., search after typing finished)
- Throttling: limit the number of executions in a time interval

```
1 const searchInput = document.getElementById("searchInput");
2
3 const debouncedSearch = debounce(function(searchTerm) {
4   console.log("Search for:", searchTerm);
5 }, 500); // Delay execution by 500ms after last keystroke
6
7 searchInput.addEventListener("keyup", function(event) {
8   debouncedSearch(event.target.value);
9 });
```

```
1 function debounce(func, delay) {
2   let timeout;
3
4   return function(...args) {
5     clearTimeout(timeout);
6     timeout = setTimeout(() => {
7       func.apply(this, args);
8     }, delay);
9   };
10 }
```



Summary

- Closures are necessary when functions have free variables
- Closures bind free variables of a function to variables from the environment
- Closures align the lifetime of functions and their accessed environment
- [Closures in Javascript](#)