# L10: Function Closure

***Rachata Ausavarungnirun***

*(rachata.a@tggs.kmutnb.ac.th)*

***February 6th, 2020***

***Architecture Research Group***

***Software System Engineering***

***Thai-German Graduate School, KMUTNB***

# Before We Begin

- Again, assignment 2 is up
- Assignment 1.1 is also up
  - Feels free to redo the nested list question
- Two more classes before the midterm
  - Then I will do one review session on Feb 13th

- Midterm: Tuesday Feb 18th

# Forall

- Recursive data types have another build-in utility
- X.forall(p) is the same as return Boolean
- def forAll[T](x: List[T], p: T => Boolean): Boolean =

```
    x match {
        case Nil => true
        case h::t => p(h) && forAll(t, p)
    }
```

# Forall: Example

- Consider the following expression type
- sealed trait Expr
  case class Constant(n: Double) extends Expr
  case class Negate(e: Expr) extends Expr
  case class Sum(e1: Expr, e2: Expr) extends Expr
  case class Prod(e1: Expr, e2: Expr) extends Expr
- def map(f: Double => Double, e: Expr): Expr = e match {
  case Constant(x) => Constant(f(x))
  case Negate(e) => Negate(map(f, e))
  case Sum(e1,e2) => Sum(map(f,e1), map(f,e2))
  case Prod(e1,e2) => Prod(map(f,e1), map(f,e2)) }

# Forall: Example

- Check if all expressions are positive
- def forAllConst(p: Double => Boolean, e: Expr): Boolean = e match
  {
  case Constant(x) => p(x) → (x: Double) => (x > 0)
  case Negate(e_) => forAllConst(p, e_)
  case Sum(e1,e2) => forAllCost(p, e1) && forAllConst(p, e2)
  case Prod(e1,e2) => forAllCost(p, e1) && forAllConst(p, e2)
  }

# Communicating Across Functions

- Let's say we want to convert a string
  - "True, False, True, False, true, False, True, Fales"
  - Into a list of Boolean
- What is the function signature?
  - def convertBoolList(st: String): List[Boolean]
- Step 1:
  - def convertBoolList(st: String): List[Boolean] = {
    val entries = st.split(",").map(_.toBoolean).toList
    entries (return)
    }

↑ Convert to Boolean (actual value True, False)

- What might be the problem?

# Communicating Across Functions

- Let's say we want to convert a string
  - "True, False, True, False, **true**, False, True, **Fales**"
  - Into a list of Boolean
- What is the function signature?
  - def convertBoolList(st: String): List[Boolean]
- Step 1:
  - def convertBoolList(st: String): List[Boolean] = {
    val entries = st.split(",").map(_.toBoolean).toList
    entries
    }
- What might be the problem?

# Communicating Across Functions

- You need to communicate this problem
- We can make the function more expressive
  - Option type (Put Nil) don't know what it means
- We can throw an exception

or

Pattern matching

- case none ———
- case h::t ———

# Communicating Across Functions

- def convertBoolList(st: String): Option[List[Boolean]] = {
  try {
    val entries = st.split(",").map(_.toBoolean).toList
   Some(entries)
  } catch {
   case e: IllegalArgumentException => None
  }
 }

# Function Closure - finish of something (End of function)

- What is the scope of a function's definition?
  - Now that functions are being passed around …

- Answer:
  - The body of a function is evaluated in the environment where the **function is defined**, not when it is called

- This is called ***the lexical scope***

- Let's do an example

# Lexical Scope Example

- Consider:

```
val x = 11                          // 1
def foo(y: Int) = x + y             // 2
def z = foo(14)                     // 3
val x = x + 1                       // 4
val y = 4                           // 5
val t = foo(x+y)                    // 6
```

x ⟶ 11 ⟶ 12

y ⟶ 4

foo(12+4) ⟶ foo(16)

foo(y→16) = 11+16 = 27

(old version of x)

- foo on line 6 is called-by-value
  - Also, on that line, x is 12 and y is 4
- Inside foo itself, x is 11
  - y is the input parameter to foo (which is 16 from line 6)

# Function Closure in Lexical Scope

- Using the last example
  - Somehow, foo takes the value x in the old environment
- Fundamentally, the execution will keep these old environment as needed
- A function definition has two parts
  - The code (the function you write)
  - The environment (at the point where you define the function)
- This part is called the function closure
  - You are teleported back to the old environment
- You cannot explicitly manipulate this environment

# More Example

- ```
  val x = 1
  def mkBar(y: Int) = {
    val t = x + 1
    (z: Int) => t+y+z
  }
  val x = 4
  val bar = mkBar(2)
  val y = 1
  val z = bar(3)
  ```

- What is z?

bar(3):

mkBar(2):

Int => (Int => Int)

(z: Int) => (2+2+z)

3

= (2+2+3) = 7
  ```

# Dynamic Scope

- Environment is used when the function is called
  - Instead of when it is defined
    ↳ Variable will be used when it is called
- PL researches shows more benefit for using lexical scoping

# Using Function Closure

- Functions can be evluated at multiple places
  - A function body: not evaluated until the function is called
  - A function body: evaluated every time the function is called
  - A variable binding: evaluates its expression when the binding is evaluated, not every time the variable is used
- To avoid repeating computation, you can store the evaluation inside function closures

# Example

- def longerThan(xs: List[String], s: String) =
  xs.filter(x => x.length > s.length)

  ↳ called many times

- def longerThan(xs: List[String], s: String) = {
  **val thresLen = s.length**
  **xs.filter(x => x.length > thresLen)**
  }
- s.length is called once and bounded
  - And you use the anonymous function to compare with x.length

# Example #2

- def fib(n:Int) : Long =
  if (n<=2) 1 else fib(n-1) + fib(n-2)
- def mkFibFoo(t:Int) = {
  (x:Long) => fib(t) + x
  }
- def mkBetterFibFoo(t: Int) = {
  val fibt = fib(t)
  (x: Long) => fibt + x
  }
- val f = mkFibFoo(45)
  // try f(1), f(2)
- val g = mkBetterFibFoo(45)
  // try g(1), g(2)

# Before We Leave Today

# In-class Exercise 10

- Work with Expr. Instead of asking is a predicate p true on all Constants, write a function def exists(p: Double => Boolean, e: Expr) that answers whether any of the constants satisfies p

- Modify our simple word count example (from Lecture 9) so that it counts the number of unique words (i.e., if you see apple 10 times, count this word only once)