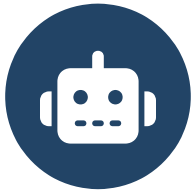
CSC 347 - Concepts of Programming Languages

Tail Recursion

Instructor: Stefan Mitsch



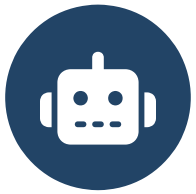
Fibonacci Numbers

>_ In Scala 3 new syntax, implement a recursive function to compute the Fibonacci numbers

💬

```scala
def fibonacci(n: Int): Int = n match {
  case 0 => 0
  case 1 => 1
  case _ => fibonacci(n - 1) + fibonacci(n - 2)
}
```

⚠ Try `fibonacci(30)` and `fibonacci(45)`



Fibonacci Numbers

Make it fast

```scala
def fibonacci(n: Int, memo: mutable.Map[Int, Int] = mutable.Map()): Int = n match {
  if n <= 0 then return 0
  if n == 1 then return 1
  if memo.contains(n) then return memo(n)
  // Calculate Fibonacci and store in the map
  val result = fibonacci(n - 1, memo) + fibonacci(n - 2, memo)
  memo(n) = result
  result
}
```

Do we need to keep all the numbers ever computed?

Learning Objectives

❓ How to get recursive iteration without stack penalty and without recomputing intermediate results?

- Identify and express tail recursion

Recursion and Stack Limitations

```
1 def countDown (x:Int) : Int = if x == 0 then 0 else 1 + countDown (x - 1)
```

- Each `(1 + ...)` represents a new AR

```
1 countDown (5)
2 --> 1 + countDown (4)
3 --> 1 + (1 + countDown (3))
4 --> 1 + (1 + (1 + countDown (2)))
5 --> 1 + (1 + (1 + (1 + countDown (1))))
6 --> 1 + (1 + (1 + (1 + (1 + countDown (0)))))
7 --> 1 + (1 + (1 + (1 + (1 + 0))))
```

- Summing up left to after the last recursive call returns
- ❓ How does the stack look like?

Call Stack

- Contains *activation records* (AR) for active calls, also known as *stack frames*
- Changes to call stack
  - AR pushed when a function/method call is made
  - AR popped when a function/method returns
- ❓ Runtime environments limit size of call stacks?
- Can cause problems with deep recursion
  - Java, Scala: `StackOverflowError`
  - C: stack limits set by operating system

Tail Recursive Calls

- All recursive calls are in tail-position

```
1 def countDown (x:Int, result:Int = 0) : Int =
2   if x == 0 then result
3   else countDown(x-1, 1+result) // tail-recursive call
```

- Expressions

```
1 countDown (5,0)
2 --> countDown (4,1)
3 --> countDown (3,2)
4 --> countDown (2,3)
5 --> countDown (1,4)
6 --> countDown (0,5)
```

- Result sum computed before recursive call is made, no work left
- ❓ How is the stack now different?

Tail Call Optimization

- Many compilers implement *tail-call optimization*
  - overwrite existing activation record instead of creating new
- Recursive calls must be *tail-recursive*
- Includes *mutual recursion*
  - `f` calls to `g`, which calls back to `f`

Exercise: Recursive vs. Tail-Recursive Fibonacci

```
1 def fib(n:Int) : Long =
2   if n <= 1 then n
3   else fib(n-1) + fib(n-2)
```

- Time complexity
  - $O(2^n)$
- ❓ How to improve?
-

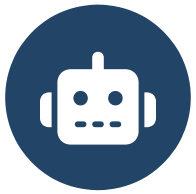
fib(0)	fib(1)	fib(2)	fib(3
0	1	1	2

- Represent sliding window in result (not tail-recursive!)

```
1 def fib(n:Int) : (Long, Long) =
2   if n <= 1 then (0, n)
3   else
4     val (a, b) = fib(n-1)
5     (b, a+b)
```

- Represent sliding window in arguments (tail-recursive)

```
1 def fib(n:Int, a:Long=0, b:Long=1) : Long =
2   if n == 0 then a
3   else if n == 1 then b
4   else fib(n-1, b, a+b)
```



Tail-recursive Fibonacci Numbers

>_ In Scala 3, implement a *tail-recursive* function to compute Fibonacci numbers

💬

```scala
def fibonacci(n: Int): Int =
  @tailrec
  def fibHelper(n: Int, a: Int, b: Int): Int = n match
    case 0 => a
    case _ => fibHelper(n - 1, b, a + b)
  end fibHelper
  fibHelper(n, 0, 1)
end fibonacci
```

- `tailrec` annotation: compiler error if not tail-recursive
- 💡 Specific instructions help generate good code

Translate Loop to Recursion

Loop (mutable data)

```
1 def factorial (n:Int) : Int =
2   val result = 1
3   var m = n
4   while m > 1 do
5     result = result * m
6     m = m - 1
7   result
```

- **Recursive (mutable)**

```
1 def factorial (n:Int) : Int =
2   var result = 1
3   var m = n
4   def loop () : Unit =
5     if m > 1 then
6       result = result*m
7       m = m-1
8       loop()
9   loop()
10  result
```

- **Recursive (mutable)**

```
1 def factorial (n:Int) : Int =
2   var result = 1
3   def loop (m:Int) : Unit =
4     if m > 1 then
5       result = result*m
6       loop(m-1)
7   loop(n)
8   result
```

- **Tail-recursive**

```
1 def factorial (n:Int) : Int =
2   @tailrec
3   def loop (m:Int, result:Int) : Int =
4     if m > 1 then loop(m-1, m*result)
5     else result
6   loop(n,1)
```

- **Recursive (immutable)**

```
1 // @tailrec // fails to compile
2 def factorial (n:Int) : Int =
3   def loop (m:Int) : Int =
4     if m > 1 then m * loop(m-1)
5     else 1
6   loop(n)
```

Summary

- Tail-call optimization
  - avoids the performance penalty of creating activation records
  - overwrites an existing activation record
  - all recursive calls must be in *tail position* (last operation)