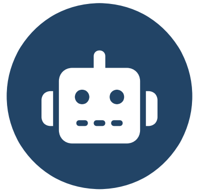
CSC 347 - Concepts of Programming Languages

Tail Recursion

Instructor: James Riely

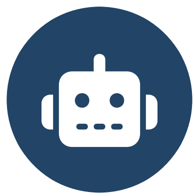


Fibonacci Numbers

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

```
1 def fibonacci(n: Int): Int = n match
2   case 0 => 0
3   case 1 => 1
4   case _ => fibonacci(n - 1) + fibonacci(n - 2)
```

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



Fibonacci Numbers

Make it fast

- 💬

```scala
1 import scala.collection.mutable
2 def fibonacci(n: Int, memo: mutable.Map[Int, Int] = mutable.Map.empty): Int =
3   if n <= 0 then return 0
4   if n == 1 then return 1
5   if memo.contains(n) then return memo(n)
6   // Calculate Fibonacci and store in the map
7   val result = fibonacci(n - 1, memo) + fibonacci(n - 2, memo)
8   memo(n) = result
9   result
```

- Applies dynamic programming
- ⚠ Do we need to keep all the numbers ever computed?
- ⚠ Does it truly solve the problem?

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 sum (xs:List[Int]) : Int =  xs match
2   case Nil      => 0
3   case x::rest => x + sum (rest)
4
5 val xs = List(11,21,31)
6 sum (xs)
```

```
sum(11::21::31::Nil)
--> sum(11::21::31::Nil)
--> 11 + sum(21::31::Nil)
--> 11 + (21 + sum(31::Nil))
--> 11 + (21 + (31 + sum(Nil)))
--> 11 + (21 + (31 + 0))
--> 11 + (21 + 31)
--> 11 + 52
--> 63 = (11 + (21 + (31 + 0)))
```

- Same as `xs.foldLeft(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

Sum of elements in a list computing forward

```
1 def sum (xs:List[Int], z:Int = 0) : Int =  xs match
2   case Nil      => z
3   case x::rest => sum (rest, z + x)
```

```
sum(11::21::31::Nil)
--> sum(11::21::31::Nil, 0)
--> sum(21::31::Nil, 11)
--> sum(31::Nil, 32)
--> sum(Nil, 63)
-->
-->
-->
--> 63 = (((0 + 11) + 21) + 31)
```

- **All recursive calls are in tail-position**
- Result sum computed before recursive call is made, no work left
- ❓ How is the stack now different?

Tail-Call Optimization: Rewrite to Loop

Tail-recursive

```
1 def sum (xs:List[Int], z:Int = 0) : Int =
2   xs match
3     case Nil      => z
4     case x::rest => sum (rest, z+x)
```

Recursive (mutable)

```
1  def sum (xs: List[Int]) : Int =
2    var z = 0
3    def loop (xs: List[Int]) : Int =
4      xs match
5        case Nil      => z
6        case x::rest =>
7          z = z + x
8          loop(rest)
9    end loop
10   loop(xs)
```

```
1  def sum (xs: List[Int]) : Int =
2    var z = 0
3    var l = xs
4    def loop () : Int =
5      l match
6        case Nil      => z
7        case x::rest =>
8          z = z + x
9          l = rest
10         loop()
11   end loop
12   loop()
```

Remove pattern matching

```
1  def sum (xs: List[Int]) : Int =
2    var z = 0
3    var l = xs
4    def loop () : Int =
5      if l == Nil then z
6      else
7        z = z + l.head
8        l = l.tail
9        loop()
10   end loop
11   loop()
```

Loop (mutable data)

```
1 def sum (xs: List[Int]) : Int =
2   var z = 0
3   var l = xs
4   while l != Nil do
5     z = z + l.head
6     l = l.tail
7   end while
8   z
```

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?
-

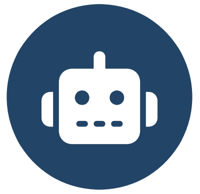
`fib(0)`	`fib(1)`	`fib(2)`	`fib(3`
0	1	1	2

- Represent sliding window in arguments (tail-recursive)

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

- Represent sliding window in `LazyList`

```
1 val fib: LazyList[BigInt] =
2   BigInt(0) #:: BigInt(1) #::
3     fib.zip(fib.tail).map {
4       case (a, b) => a + b
5     }
```

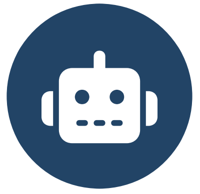


Tail-recursive Fibonacci Numbers

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

```scala
1 def fibonacci(n: Int): Int =
2   @tailrec
3   def fibHelper(n: Int, a: Int, b: Int): Int = n match
4     case 0 => a
5     case _ => fibHelper(n - 1, b, a + b)
6   end fibHelper
7   fibHelper(n, 0, 1)
8 end fibonacci
```

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



Continuation Passing

- Can all functions be made tail-recursive?
- Continuation passing: add an extra argument for a function to take the "return" value

Tail-recursive accumulator

```
1 def sum (xs:List[Int], z:Int = 0) : Int =
2   xs match
3     case Nil     => z
4     case x::rest => sum (rest, z+x)
5 end sum
6
7 var s = sum(List(1,2,3))
```

Tail-recursive continuation passing

```
1 def sum (xs:List[Int], c:Int=>Unit) : Unit =
2   xs match
3     case Nil     => c(0)
4     case x::rest => sum (rest, y=>c(x+y))
5 end sum
6
7 var s = -1
8 sum(List(1,2,3), s=_)
```

- Continuation passing is used by compilers of functional languages as an intermediate form (similar to static single assignment in imperative languages)

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)
  - continuation passing replaces return value with a function that is called upon "return"

Tail-Call Optimization: Rewrite to Loop

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 (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
```

```
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
```

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
```