CS320

Immutability and Recursion

Jaemin Hong

September 2, 2020

Before We Start. . .

Today's lecture deals with

- defining variables and functions in Scala
- advantages of immutability
- recursion

There are many topics, but I have only 15 minutes.

- I will explain only important ideas.
- Please try example runs by yourself.

Defining Variables

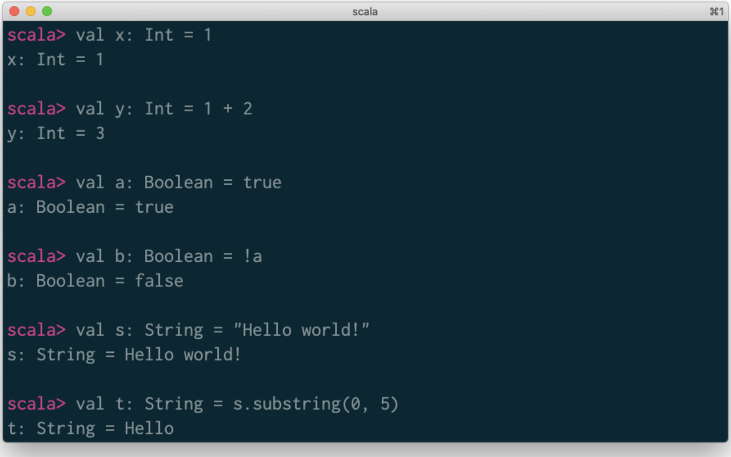
`val` *name*`:` *type* `=` *expr*

```
val x: Int = 1
val y: Int = 1 + 2

val a: Boolean = true
val b: Boolean = !a

val s: String = "Hello world!"
val t: String = s.substring(0, 5)
```

Scala types: `Int`, `Char`, `Float`, `Boolean`, `String`, `Unit`, `(Int, String)`, `List[Int]`, `Option[String]`, `Int => Int`, `...`

Defining Variables

Use the REPL for simple examples!



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

scala> val y: Int = 1 + 2
y: Int = 3

scala> val a: Boolean = true
a: Boolean = true

scala> val b: Boolean = !a
b: Boolean = false

scala> val s: String = "Hello world!"
s: String = Hello world!

scala> val t: String = s.substring(0, 5)
t: String = Hello
```

Defining Variables

Type annotations make code verbose.
You can omit type annotations.

`val` *name* `=` *expr*

```
val x = 1
val y = 1 + 2

val a = true
val b = !a

val s = "Hello world!"
val t = s.substring(0, 5)
```

Defining Variables

Variables defined by `val` are immutable.

```
val x = 1
x = 2
```

⇒ error: reassignment to val

To define mutable variables, you can use `var`.

```
var x = 1 (OR var x: Int = 1)
x = 2
```

However, **DO NOT use `var` in exercises and projects** except where we specify to allow. Also, if you are new to FP, try to write code as much as you can without `var`.

Defining Function

`def` *name*(*name*: *type*, ⋯): *type* = *expr*

```
def add(x: Int, y: Int): Int = x + y
```

```
def addSquared(x: Int, y: Int): Int =
  add(x * x, y * y)
```

You do not need to use `return`.

Defining Function

`def` *name*(*name*: *type*, ···): *type* = *expr*

```
def add(x: Int, y: Int): Int = x + y

def addSquared(x: Int, y: Int): Int = {
  val xSquared = x * x
  val ySquared = y * y
  add(xSquared, ySquared)
}
```

You can define variables inside functions.
To write multiple lines, use curly braces.

Defining Function

`def` *name*(*name*: *type*, ⋯): *type* = *expr*

```
def add(x: Int, y: Int): Int = x + y
```

```
def addSquared(x: Int, y: Int): Int = {
  def square(x: Int): Int = x * x
  add(square(x), square(y))
}
```

You can define functions inside functions.

Defining Function

You can omit retun type annotations.

def *name*(*name*: *type*, $\cdots$) = *expr*

```
def add(x: Int, y: Int) = x + y

def addSquared(x: Int, y: Int) = {
  def square(x: Int) = x * x
  add(square(x), square(y))
}
```

Defining Function

You CANNOT omit parameter type annotations.

```
def add(x, y) = x + y
```

⇒ error: ':' expected but ',' found.

Type annotations are useful for

- debugging
- documentation

Therefore,

- omit them only if they are too trivial (local variables...)
- keep them to make code contain more information (funtion return types...)

Immutability

Immutability is one of the key principles of functional programming. It has various advantages:

1. is easier to reason about
2. does not require defensive copies before passing
3. can be accessed concurrently by multiple threads
4. makes safe hash table keys

(from the book “Programming in Scala”)

Immutability

1. Immutable things are easier to reason about.

```
def f(y: Int) = {
  val x = y
  ...
  g(x, ...)
}
```

Yeah! x still equals y.

```
def f(y: Int) = {
  var x = y
  ...
  g(x, ...)
}
```

Does x still equal y?

Immutability

1. Immutable things are easier to reason about.

```
def f(y: Int) = {
  val x = List(y)
  ...
  g(x, ...)
```

Yeah! x still contains y.

```
  ...
  h(x, ...)
}
```

Yeah! x still contains y.

```
def f(y: Int) = {
  val x = ListBuffer(y)
  ...
  g(x, ...)
```

Does x still contain y?

```
  ...
  h(x, ...)
}
```

Does x still contain y?

Immutability

1. Immutable things are easier to reason about.

```
def f(y: Int) = {
  val x = g(y)
```

Yeah! I know `y`.

```
  h(x)
}
```

Yeah! I know `x`.

```
def f(y: Int) = {
  val x = g(y, a)
```

What is `a`?

```
  h(x, b)
}
```

What is `b`?

Code with mutable **global** variables are especially difficult.

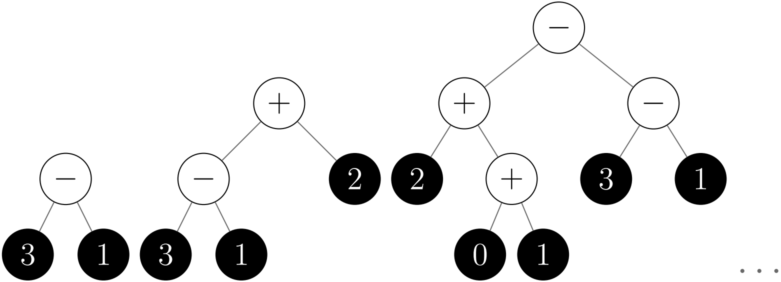
Immutability

2. Mutable objects require defensive copies before passing.

```
def f(y: Int) = {      def f(y: Int) = {
  val x = List(y)        val x = ListBuffer(y)

  I do not want to allow the function g to change x.

                         val x2 = x.clone
  g(x, ...)              g(x2, ...)

x remains the same.    x remains the same.
  ...                    ...
}                      }
```

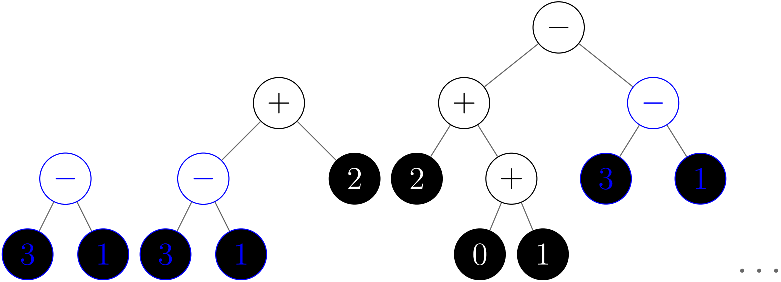
Immutability

This semester, you will treat lots of abstract syntax trees.



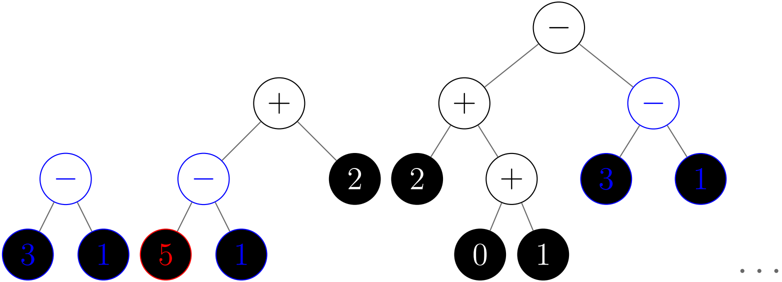
Immutability

Some of them may share the same subtree.



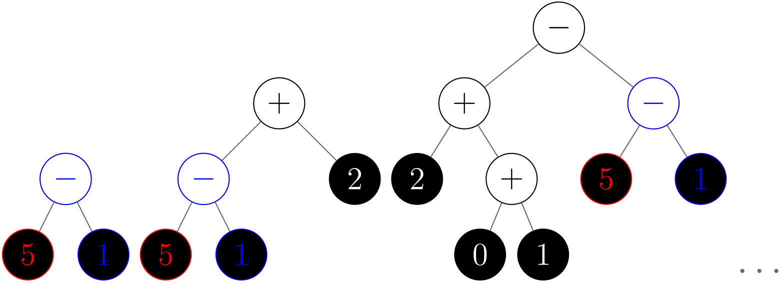
Immutability

If you mutate one leaf, . . .



Immutability

If you mutate one leaf, there can be unintended changes.



However, if you mutate nothing, no problem!

Recursion

Consider the factorial function.
(For brevity, ignore negative integers and integer overflow.)

```
def factorial(n: Int): Int = {
  var m = n
  var res = 1
  while (m > 1) {
    res *= m
    m -= 1
  }
  res
}
```

Recursion

Loops are essential in programming. However, if everything is immutable, will loops work?

`while` ($expr_1$) $expr_2$

The value of $expr_1$ never changes. Therefore, we have 2 possibilites:

- $expr_1$ is `false`. The loop does nothing.
- $expr_1$ is `true`. The program runs forever.

Loops are mostly useless. How can we implement the factorial function?

Recursion

- The solution is **recursion**.
- A recursive function is a function calling itself.
- As loops mutate things, recursive functions change arguments for recursive calls.
- In Scala, the return types of recursive functions cannot be ommited.

Recursion

```
def factorial(n: Int): Int =
  if (n <= 1) 1
  else n * factorial(n - 1)
```

- It reflects the mathematical definition of the factorial function: $n! = \left\{ \begin{array}{ll} 1 & \text{if } n \leq 1 \\ n \times (n-1)! & \text{if } n > 1 \end{array} \right\}$
- It is easier to reason about than the loop version.

Jaemin Hong
jaemin.hong@kaist.ac.kr
https://hjaem.info