Programming Languages – Functional Languages

Mahmoud Abdelsalam

Functional Programming concepts

- Pure functions & side effects
- Referential transparency
- First class functions & higher order functions
- Immutability
- Recursion & tail-recursion
- Lambda functions
- Strict and lazy evaluation
- Pattern matching

Functional Programming concepts - Immutability

- Immutability: variables/objects that don’t change.

Functional Programming concepts - Immutability

- Immutability: variables/objects that don’t change.
  - You can create a NEW one but you can’t modify the existing ones.

Functional Programming concepts - Immutability

- Immutability: variables/objects that don’t change.
  - You can create a NEW one but you can’t modify the existing ones.
  - variable/reference immutability vs object immutability.

Functional Programming concepts - Immutability

- Variables mutability vs immutability (two types of variables in Scala):

Functional Programming concepts - Immutability

- Variables mutability vs immutability (two types of variables in Scala):
  - var: allows reassignment of objects.
  - val: doesn’t allow reassignment of objects.

Functional Programming concepts - Immutability

- Variables mutability vs immutability (two types of variables in Scala):
  - var: allows reassignment of objects.
  - val: doesn't allow reassignment of objects.

```
var x = 5
x = 10 //OK
```

Functional Programming concepts - Immutability

- Variables mutability vs immutability (two types of variables in Scala):
  - var: allows reassignment of objects.
  - val: doesn't allow reassignment of objects.

```
var x = 5
x = 10 //OK

val x = 5
x = 10 //Error
```

Functional Programming concepts - Immutability

- Objects mutability vs immutability:
  - You can have immutable variable pointing to mutable object:

Functional Programming concepts - Immutability

- Objects mutability vs immutability:
  - You can have immutable variable pointing to mutable object:

```
//Array is mutable object in Scala
val arr = Array(5, 6)
arr(0) = 10 //OK
arr = Array(1, 2) //Error
```

Functional Programming concepts - Immutability

- Objects mutability vs immutability:
  - You can have mutable variable pointing to mutable object:

Functional Programming concepts - Immutability

- Objects mutability vs immutability:
  - You can have mutable variable pointing to mutable object:

```
//Array is mutable object in Scala
var arr = Array(5, 6)
arr(0) = 10 //OK
arr = Array(1, 2) //OK
```

Functional Programming concepts - Immutability

- Objects mutability vs immutability:
  - You can have immutable variable pointing to immutable object:

Functional Programming concepts - Immutability

- Objects mutability vs immutability:
  - You can have immutable variable pointing to immutable object:

```
//List is immutable object in Scala
val arr = List(5, 6)
arr(0) = 10 //Error
arr = List(1, 2) //Error
```

Functional Programming concepts - Immutability

- Objects mutability vs immutability:
  - You can have mutable variable pointing to immutable object:

Functional Programming concepts - Immutability

- Objects mutability vs immutability:
  - You can have mutable variable pointing to immutable object:

```
//List is immutable object in Scala
var arr = List(5, 6)
arr(0) = 10 //Error
arr = List(1, 2) //OK
```

Functional Programming concepts - Immutability

- WHY do we need immutability???

Functional Programming concepts - Immutability

- WHY do we need immutability???

  1. One you create an object and give it a value, it’s guaranteed to stay the same.

Functional Programming concepts - Immutability

- WHY do we need immutability???

1. One you create an object and give it a value, it’s guaranteed to stay the same.
2. More mathematical like.

Functional Programming concepts - Immutability

- WHY do we need immutability???

1. One you create an object and give it a value, it's guaranteed to stay the same.
2. More mathematical like.
3. Think about concurrent programming
   - Multi-threading? Immutable objects are more thread-safe.

Functional Programming concepts - Immutability

HOW do we write programs using only immutable objects?

Functional Programming concepts - Immutability

- Factorial function:

```
def fact(n: Int): Int = {
    var i = n //Can't change input
    var f = 1
    while (i > 1){
        f = f * i
        i = i - 1
    }
    f
}
```

Functional Programming concepts - Immutability

- Factorial function – use recursion:

```
def fact(n: Int): Int = {
    if (n < 1) 1
    else n * fact(n - 1)
}
```

For loops, we use recursion.

Functional Programming concepts - Immutability

What about other situations?

For example:

I have string “Hello, world”

I want to change the first char to ‘#’, to get: “#ello, world”

Functional Programming concepts - Immutability

What about other situations?

For example:

I have string “Hello, world”

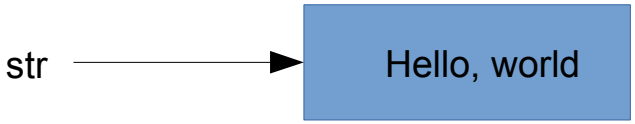
I want to change the first char to ‘#’, to get: “#ello, world”

For each new state/change:

we create a new object

Functional Programming concepts - Immutability

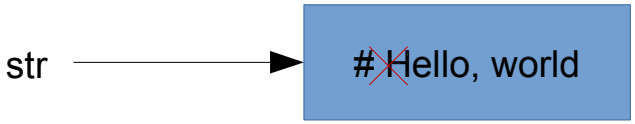
val str = "Hello, world"



Functional Programming concepts - Immutability

val str = "Hello, world"

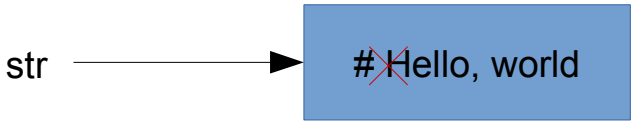
//str(0) = '#'



Functional Programming concepts - Immutability

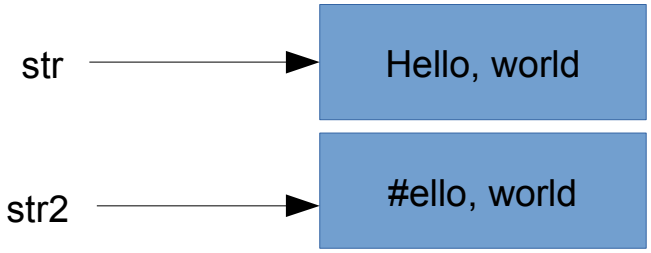
val str = "Hello, world"

//str(0) = '#' -> Error strings are immutable



Functional Programming concepts - Immutability

```
val str = "Hello, world"
//str(0) = '#' -> Error strings are immutable
val str2 = "#" + str.substring(1)
```



Functional Programming concepts - Immutability

- What if we use var?

  var str = "Hello, world"

  //str(0) = '#' -> Error strings are immutable

  str = "#" + str.substring(1)

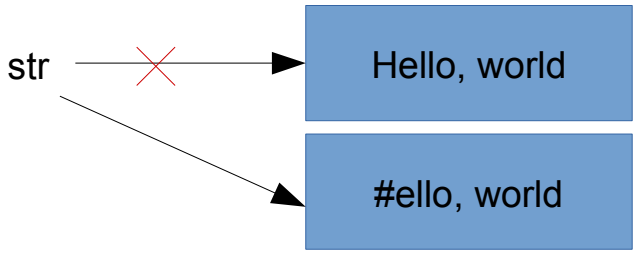
Functional Programming concepts - Immutability

- What if we use var?

  var str = "Hello, world"

  //str(0) = '#' -> Error strings are immutable

  str = "#" + str.substring(1)



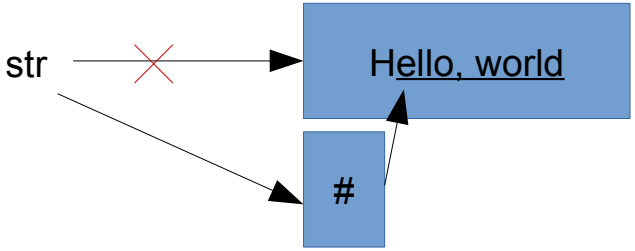
Functional Programming concepts - Immutability

- What about performance? Creating new object for each change seems excessive!

Functional Programming concepts - Immutability

- What about performance? Creating new object for each change seems excessive!

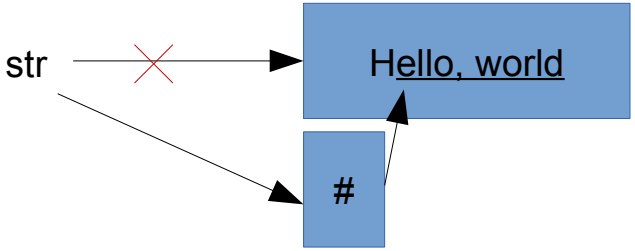
Sharing partial values



Functional Programming concepts - Immutability

- What about performance? Creating new object for each change seems excessive!

Sharing partial values & Persistent Data Structures



Functional Programming concepts

- Pure functions & side effects
- Referential transparency
- First class functions & higher order functions
- Immutability
- Recursion & tail-recursion
- Lambda functions
- Strict and lazy evaluation
- Pattern matching

Functional Programming concepts – Recursion & Tail Recursion

- Recursion is the concept of breaking down a problem into smaller pieces and solving each one incrementally.

Functional Programming concepts – Recursion & Tail Recursion

- Recursion is the concept of breaking down a problem into smaller pieces and solving each one incrementally.
- For example, sum of range from 1 to n:

```
def sum(n: Int): Int = {
    if (n == 1) 1
    else n + sum(n - 1)
}
```

Functional Programming concepts – Recursion & Tail Recursion

- Recursion can be very exhausting to the stack memory.

Functional Programming concepts – Recursion & Tail Recursion

- Recursion can be very exhausting to the stack memory.
- Loops are more efficient? YES

Functional Programming concepts – Recursion & Tail Recursion

- Recursion can be very exhausting to the stack memory.
- Loops are more efficient? YES

Solution? Tail recursion

Functional Programming concepts – Recursion & Tail Recursion

- Tail recursion:
  - Recursive call should be the last statement to be executed.
  - The compiler will substitute the recursive function by a loop.

Functional Programming concepts – Recursion & Tail Recursion

- Use tail recursion:

```
def sum(n: Int, res: Int): Int = {
    if (n == 1) res
    else sum(n - 1, res + n)
}
```

Functional Programming concepts

- Pure functions & side effects
- Referential transparency
- First class functions & higher order functions
- Immutability
- Recursion & tail-recursion
- Lambda functions
- Strict and lazy evaluation
- Pattern matching