

Functional Programming in Typed Racket

Florian Florian
fbie@itu.dk

IT University of Copenhagen & UCAS

2016-05-26



IT University
of Copenhagen

Who am I?

Just call me Florian!

Who am I?

Just call me Florian!

- ▶ I am from Germany and live in Denmark.

Who am I?

Just call me Florian!

- ▶ I am from Germany and live in Denmark.
- ▶ Second year PhD student at IT University of Copenhagen and UCAS.

Who am I?

Just call me Florian!

- ▶ I am from Germany and live in Denmark.
- ▶ Second year PhD student at IT University of Copenhagen and UCAS.
- ▶ My Chinese is really bad (but I am trying to learn!)

Who am I?

Just call me Florian!

- ▶ I am from Germany and live in Denmark.
- ▶ Second year PhD student at IT University of Copenhagen and UCAS.
- ▶ My Chinese is really bad (but I am trying to learn!)
- ▶ If you do not understand what I say or if I speak too fast, **please tell me right away!**

All slides and course materials are available on:

<https://github.com/fbie/parallel-functional-lectures>

Why Functional Programming?

Functional programming is old but recently it has become very popular!

Why Functional Programming?

Functional programming is old but recently it has become very popular!

- ▶ Makes it easier to think about what a program does.

Why Functional Programming?

Functional programming is old but recently it has become very popular!

- ▶ Makes it easier to think about what a program does.
- ▶ Parallel in its nature → easy to parallelize (next lecture).

Why Functional Programming?

Functional programming is old but recently it has become very popular!

- ▶ Makes it easier to think about what a program does.
- ▶ Parallel in its nature → easy to parallelize (next lecture).
- ▶ Used in financial industry, modern compilers and more.

Why Functional Programming?

Functional programming is old but recently it has become very popular!

- ▶ Makes it easier to think about what a program does.
- ▶ Parallel in its nature → easy to parallelize (next lecture).
- ▶ Used in financial industry, modern compilers and more.
- ▶ Jobs in functional programming pay better!



Racket and its sister-language **Typed Racket** are Scheme-dialects.



Racket and its sister-language **Typed Racket** are Scheme-dialects.

- ▶ Racket is dynamically typed, Typed Racket has a static, strong type system.



Racket and its sister-language **Typed Racket** are Scheme-dialects.

- ▶ Racket is dynamically typed, Typed Racket has a static, strong type system.
- ▶ Functional language, no side-effects (mostly).



Racket and its sister-language **Typed Racket** are Scheme-dialects.

- ▶ Racket is dynamically typed, Typed Racket has a static, strong type system.
- ▶ Functional language, no side-effects (mostly).
- ▶ Good for meta-programming (we won't look at that).

Hello World in Typed Racket

```
;; This is a comment!  
;; Tell the run-time, which language to use.  
#lang typed/racket  
  
;; Now, print something.  
(print "Nihao!")
```

A More Interesting Program

```
#lang typed/racket
```

```
;; Define x as the result of the computation.  
(define x (* (+ 2 4) (+ 42 9)))
```

A More Interesting Program

```
#lang typed/racket
```

```
;; Define x as the result of the computation.  
(define x (* (+ 2 4) (+ 42 9)))
```

We can check its value in the interactive mode:

```
> x  
59
```

Do You Think This Looks Strange?

In Racket, all expressions are written like this:

```
(function arg1 arg2 ... argN)
```

Do You Think This Looks Strange?

In Racket, all expressions are written like this:

```
(function arg1 arg2 ... argN)
```

Operators are also just functions:

<code>(+ x y)</code>	\Rightarrow	$x + y$
<code>(> x y)</code>	\Rightarrow	$x > y$
<code>(/ x y)</code>	\Rightarrow	$\frac{x}{y}$
<code>(f x y)</code>	\Rightarrow	$f(x, y)$

Do You Think This Looks Strange?

In Racket, all expressions are written like this:

```
(function arg1 arg2 ... argN)
```

Operators are also just functions:

<code>(+ x y)</code>	\Rightarrow	$x + y$
<code>(> x y)</code>	\Rightarrow	$x > y$
<code>(/ x y)</code>	\Rightarrow	$\frac{x}{y}$
<code>(f x y)</code>	\Rightarrow	$f(x, y)$

Some functions are *variadic*:

<code>(+ x y z)</code>	\Rightarrow	$x + y + z$
------------------------	---------------	-------------

Local Bindings

Just like local variables in Java, but you can never change them!

```
#lang typed/racket
```

```
(let ([x (* 2 16)]  
      [y (* 3 17)])  
  (print (+ x y)))
```

What does this program do?

Note: You cannot reference `x` and `y` after the last closing parenthesis of the `let` expression!

The Same Program in Java

```
public static void main(String[] args) {  
    int x = 2 * 16;  
    int y = 3 * 17;  
    System.out.println(x + y);  
}
```


A First Function

```
#lang typed/racket

(: times-two (-> Number Number))
(define (times-two x)
  (* 2 x))
```

A First Function

```
#lang typed/racket

(: times-two (-> Number Number))
(define (times-two x)
  (* 2 x))
```

Several new things on this slide:

A First Function

```
#lang typed/racket

(: times-two (-> Number Number))
(define (times-two x)
  (* 2 x))
```

Several new things on this slide:

- Functions need no return statement. Their return value is the last executed statement!

A First Function

```
#lang typed/racket

(: times-two (-> Number Number))
(define (times-two x)
  (* 2 x))
```

Several new things on this slide:

- ▶ Functions need no return statement. Their return value is the last executed statement!
- ▶ Type annotations start with `:` and describe the type of a symbol.

A First Function

```
#lang typed/racket

(: times-two (-> Number Number))
(define (times-two x)
  (* 2 x))
```

Several new things on this slide:

- ▶ Functions need no return statement. Their return value is the last executed statement!
- ▶ Type annotations start with `:` and describe the type of a symbol.
- ▶ The type of `times-two` is *Number* \rightarrow *Number*.

Types in Typed Racket

We write the function type $A \rightarrow B$ as:

`(-> A B)`

where `A` is the parameter type and `B` is the return type.

Types in Typed Racket

We write the function type $A \rightarrow B$ as:

`(-> A B)`

where `A` is the parameter type and `B` is the return type.

`(-> A B C)`

Which types are parameter types, which ones are return types?

A Second Function

```
(: funny (-> Number Number)
(define (funny x)
  (if (< x 0)
      (- x)
      x))
```


A Second Function

```
(: funny (-> Number Number)
(define (funny x)
  (if (< x 0)
      (- x)
      x))
```

Conditionals have the form (if B E1 E2).

Polymorphic Types

Polymorphic types in Typed Racket are very explicit:

```
(: twice (All (A) (-> A (Pairof A A))))  
(define (twice a)  
  (cons a a))
```

Polymorphic Types

Polymorphic types in Typed Racket are very explicit:

```
(: twice (All (A) (-> A (Pairof A A))))  
(define (twice a)  
  (cons a a))
```

“For all types A , the type of `twice` is such that iff you pass it a value of some type A it will return a pair of type $A \times A$.”

Polymorphic Types

Polymorphic types in Typed Racket are very explicit:

```
(: twice (All (A) (-> A (Pairof A A))))  
(define (twice a)  
  (cons a a))
```

“For all types A , the type of `twice` is such that iff you pass it a value of some type A it will return a pair of type $A \times A$.”

It's just like generic types in Java!

The Same Program in Java

```
public static Pairof<A, A> twice(A a) {  
    return new Pairof<A, A>(a);  
}
```

Note: There is no build-in pair type in Java :(

State is Immutable!

You cannot change a variable's value. Instead, use **recursion**:

```
(: is-even? (-> Integer Boolean))  
(define (is-even? n)  
  (if (< 0 n)  
      (is-even? (- n 2))  
      (= n 0)))
```

State is Immutable!

You cannot change a variable's value. Instead, use **recursion**:

```
(: is-even? (-> Integer Boolean))  
(define (is-even? n)  
  (if (< 0 n)  
      (is-even? (- n 2))  
      (= n 0)))
```

Now, we can call the function:

```
> (is-even? 1)  
#f
```

State is Immutable!

You cannot change a variable's value. Instead, use **recursion**:

```
(: is-even? (-> Integer Boolean))  
(define (is-even? n)  
  (if (< 0 n)  
      (is-even? (- n 2))  
      (= n 0)))
```

Now, we can call the function:

```
> (is-even? 1)  
#f  
  
> (is-even? 2048)  
#t
```


The Same Program in Java

```
public static bool isEven(int n) {  
    while (0 < n)  
        n = n - 2;  
    return n == 0;  
}
```

Anonymous Functions

You can define functions without giving them a name. Such functions are called *lambda expressions*:

```
> ((lambda (x) x) 2)  
2
```

Anonymous Functions

You can define functions without giving them a name. Such functions are called *lambda expressions*:

```
> ((lambda (x) x) 2)  
2
```

Here, we have defined a lambda expression and applied it to the value 2. We call this function the *identity function*.

Anonymous Functions

You can define functions without giving them a name. Such functions are called *lambda expressions*:

```
> ((lambda (x) x) 2)  
2
```

Here, we have defined a lambda expression and applied it to the value 2. We call this function the *identity function*.

We will often come across lambda expressions in functional programming!

Structs in Typed Racket

Structs are containers for values.

```
(struct (myNumberBox  
        [v : Number]))
```

```
(struct (myNumberStringBox  
        ([n : Number] [s : String])))
```

Structs in Typed Racket

Structs are containers for values.

```
(struct (myNumberBox  
        [v : Number]))
```

```
(struct (myNumberStringBox  
        ([n : Number] [s : String])))
```

We can also use type polymorphism here:

```
(struct (A) (myPolyBox [value : A]))
```

That Was a Quick Intro

Next, we will do a live coding session!

That Was a Quick Intro

Next, we will do a live coding session!

- ▶ I code on the big screen and show you how to do functional programming in Racket.

That Was a Quick Intro

Next, we will do a live coding session!

- ▶ I code on the big screen and show you how to do functional programming in Racket.
- ▶ You help with ideas and suggestions and experiment yourselves!

That Was a Quick Intro

Next, we will do a live coding session!

- ▶ I code on the big screen and show you how to do functional programming in Racket.
- ▶ You help with ideas and suggestions and experiment yourselves!
- ▶ All code we write will be available on
 - ▶ github.com/fbie/parallel-functional-programming

That Was a Quick Intro

Next, we will do a live coding session!

- ▶ I code on the big screen and show you how to do functional programming in Racket.
- ▶ You help with ideas and suggestions and experiment yourselves!
- ▶ All code we write will be available on
 - ▶ github.com/fbie/parallel-functional-programming
- ▶ You can download Racket from
 - ▶ racket-lang.org

Java Equivalent to Maybe

```
public abstract class Maybe<A> {}

public class None<A> extends Maybe<A> {
    public None() {}
}

public class Some<A> extends Maybe<A> {
    public final A a;
    public Some(A a) {
        this.a = a;
    }
}
```

Singly-Linked Cons List 1/2

```
(define xs (cons 'a (cons 'b  
    (cons 'c (cons 'd (cons 'e '()))))))
```

Singly-Linked Cons List 1/2

```
(define xs (cons 'a (cons 'b  
  (cons 'c (cons 'd (cons 'e '()))))))
```

Get the first element of `xs`, the “head”:

```
> (first xs)  
'a  
> (cdr xs)  
'a
```

Singly-Linked Cons List 1/2

```
(define xs (cons 'a (cons 'b  
  (cons 'c (cons 'd (cons 'e '()))))))
```

Get the first element of `xs`, the “head”:

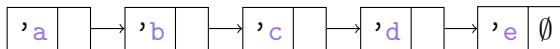
```
> (first xs)  
'a  
> (cdr xs)  
'a
```

Get the remaining part of `xs`, the “tail”:

```
> (rest xs)  
'('b 'c 'd 'e)  
> (cdr xs)  
'('b 'c 'd 'e)
```

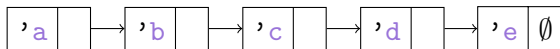
Singly-Linked Cons List 2/2

References to tail **cannot be changed**, because all **references are immutable**:



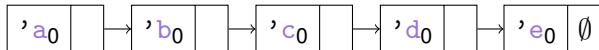
Singly-Linked Cons List 2/2

References to tail **cannot be changed**, because all **references are immutable**:



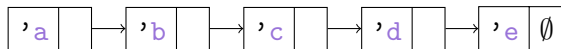
If we want to remove an element, we must build a new list, but we can re-use the tail of the element that we have deleted:

```
(remove xs 'c)
```



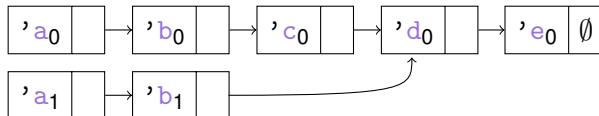
Singly-Linked Cons List 2/2

References to tail **cannot be changed**, because all **references are immutable**:



If we want to remove an element, we must build a new list, but we can re-use the tail of the element that we have deleted:

```
(remove xs 'c)
```



Java Equivalent to Cons List

```
public abstract class LinkedList<A> {}

public class Nil<A> extends LinkedList<A> {
    public Nil() {}
}

public class Cons<A> extends LinkedList<A> {
    public final A a;
    public final LinkedList<A> tail;

    public Cons(A a, LinkedList<A> tail) {
        this.a = a;
        this.tail = tail;
    }
}
```

Java Equivalent to Binary Tree

```
public abstract class BinaryTree<A> {}

public class Leaf<A> extends BinaryTree<A> {
    public final A a;
    public Leaf(A a) {
        this.a = a;
    }
}

public class Node<A> extends BinaryTree<A> {
    public final BinaryTree<A> left, right;
    public Node(BinaryTree<A> left,
                BinaryTree<A> right) {
        this.left = left; this.right = right;
    }
}
```