# Functional Programming in Typed Racket

Florian BIERMANN
`fbie@itu.dk`

IT University of Copenhagen & UCAS

2016-05-26

## 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!**

## 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.
- Good for parallel programming, performance (see 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.
- Good for parallel programming, performance (see 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.
- Good for parallel programming, performance (see next lecture).
- Used in financial industry, modern compilers and more.
- Jobs in functional programming pay better!

## Typed Racket

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

## Typed Racket

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

- As the names say, Racket is untyped, Typed Racket has types.

## Typed Racket

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

- As the names say, Racket is untyped, Typed Racket has types.
- Functional language but with support for objects.

## Typed Racket

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

- As the names say, Racket is untyped, Typed Racket has types.
- Functional language but with support for objects.
- Great meta-programming (we won’t look at that).

## Hello World in Typed Racket

```
;; Tell 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)))
```

## Do You Think This Looks Strange?

In Racket, all expressions are written like this:

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

Operators are also just functions:

$$\begin{array}{lcc}
\texttt{(+ x y)} & \Rightarrow & x + y \\
\texttt{(> x y)} & \Rightarrow & x > y \\
\texttt{(/ x y)} & \Rightarrow & \frac{x}{y} \\
\texttt{(f x y)} & \Rightarrow & f(x, y)
\end{array}$$

## Local Bindings

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

```
#lang typed/racket

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

What does this program do?

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

## 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*.