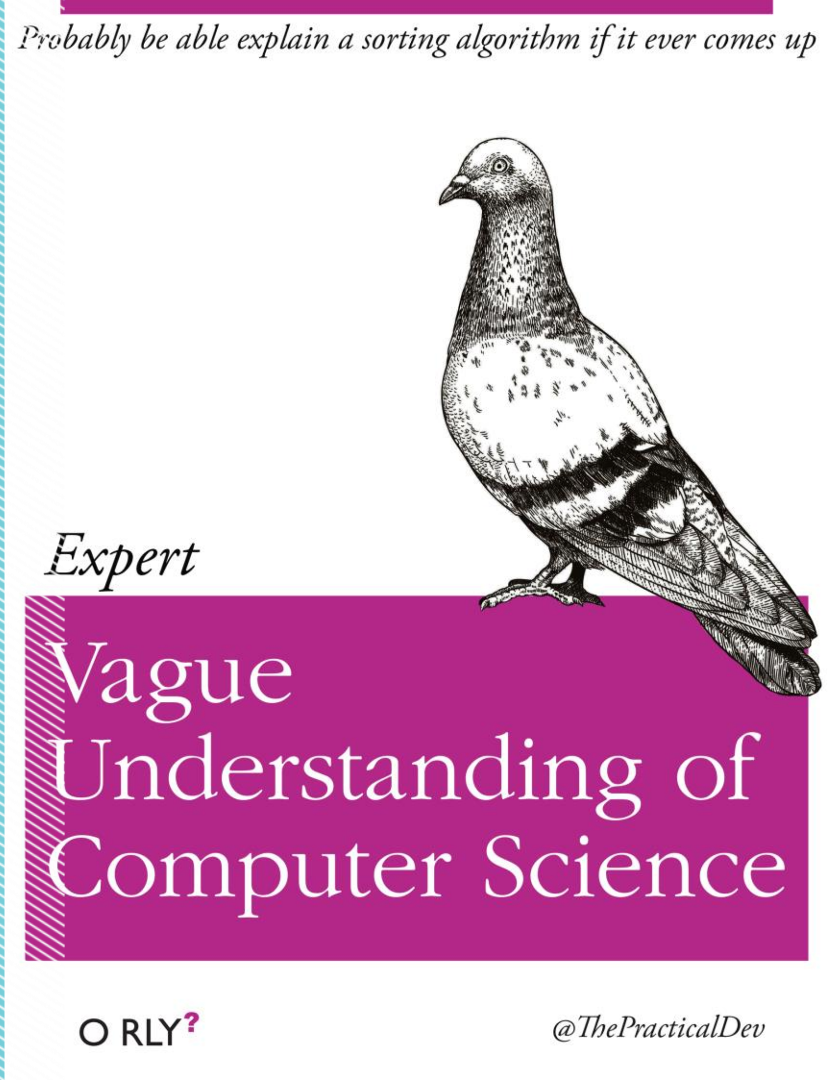
Functional Programming

(or what is *lambda* anyway)


Probably be able explain a sorting algorithm if it ever comes up
Expert
Vague Understanding of Computer Science
O RLY?
@ThePracticalDev

Frontend Junior Program - 2022

CONFIDENTIAL | © 2022 EPAM Systems, Inc.

Agenda

1. Intro
2. Turing Machine
3. Lambda Calculus
4. Functional Programming

Introduction

Our story begins in 1936 – yes, that was well before any electronic computers exist in the world

Probably it is not a surprise that there is science behind advanced technologies. Programming is not an exception (see also: *Computer Science*), and while *Software Engineering* relies on that heavily – especially in some areas – usually we don't need to have a deep dive into math.

Algorithmic thinking is vastly different from *mathematical thinking* (you don't have to be good in math to became a good Software Engineer). While we have complex problems here, solving these requires different approaches.

* * *

There is a field, however, where the science behind a code wonderfully reveals itself and it is very hard to understand what the functional programming is without having a bit of time travel.

"Sometimes it is the people no one imagines anything of who do the things that no one can imagine."

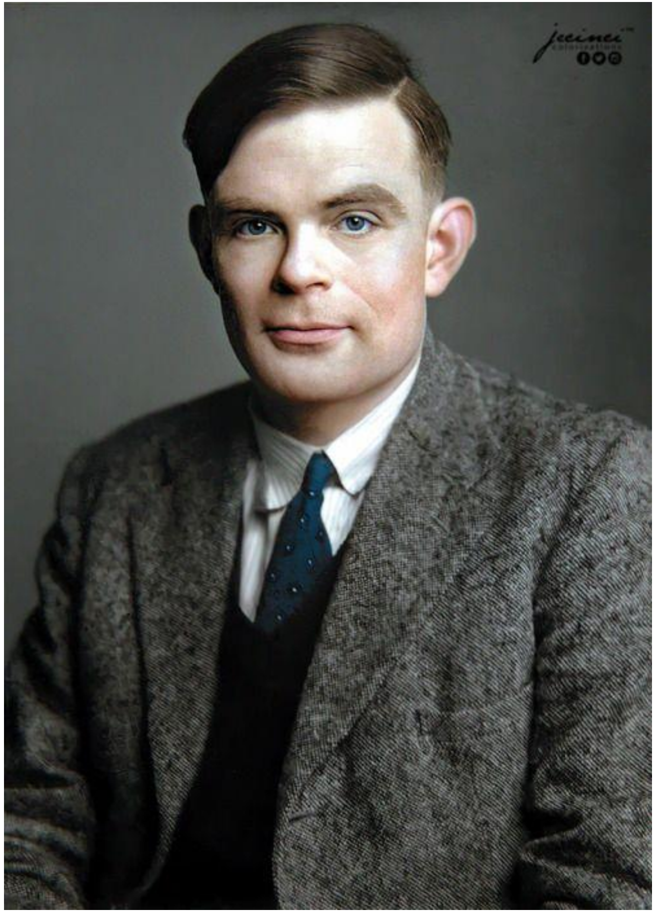
Turing machine

In 1936 a young English mathematician, Alan Turing, proposed an answer for the *Entscheidungsproblem** (decidability)

The answer was basically ~~42~~ “no” and he used a theoretical machine: the Turing Machine to prove that.

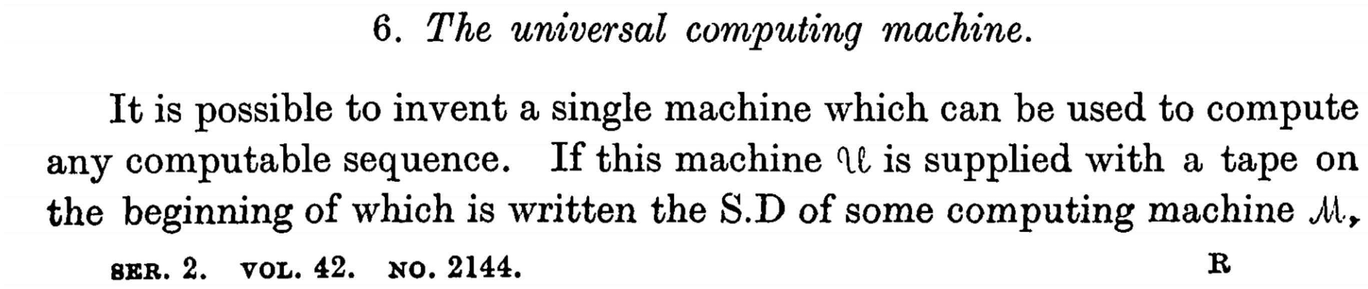
The Turing Machine basically consists of an

1., infinite tape, containing symbols
2., a head, which can do one of these:
   - reads a symbol
   - writes a symbol, according to the instructions
   - moves right
   - moves left
   - stops



** “Is there an effective procedure which, given a set of axioms and a mathematical proposition, decides whether it is or is not provable from the axioms?”*

CONFIDENTIAL | © 2022 EPAM Systems, Inc.



6. *The universal computing machine.*

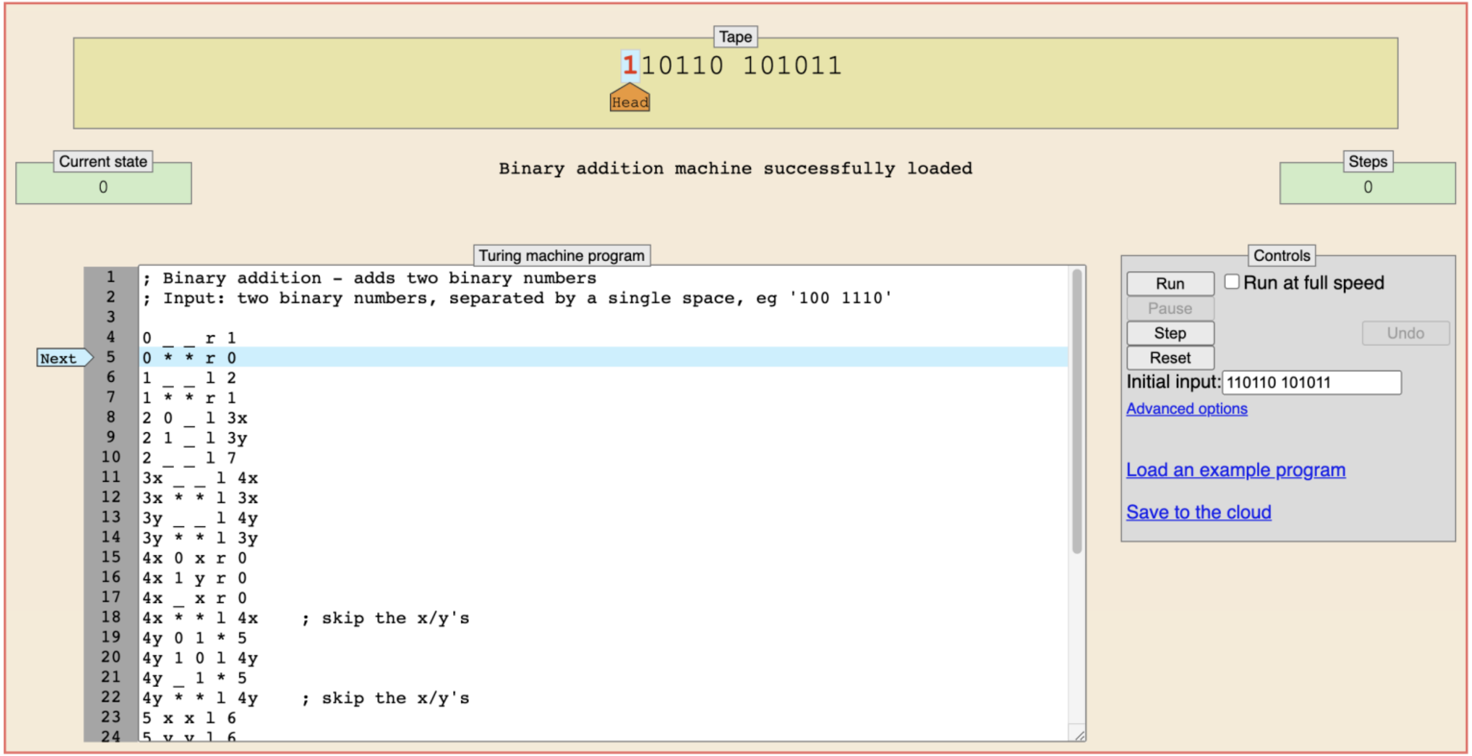
It is possible to invent a single machine which can be used to compute any computable sequence. If this machine $\mathfrak{U}$ is supplied with a tape on the beginning of which is written the S.D of some computing machine $\mathcal{M}$,

SER. 2. VOL. 42. NO. 2144. R

https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

yes, your mobile phone is a Turing Machine

A Turing Machine can solve formalized problems, and it was so valuable concept that computers existing today are real implementations of a Turing Machine.



```
110110 101011
```

Current state: 0

Binary addition machine successfully loaded

Steps: 0

```
; Binary addition - adds two binary numbers
; Input: two binary numbers, separated by a single space, eg '100 1110'

0 _ _ r 1
0 * * r 0
1 _ _ l 2
1 * * r 1
2 0 _ l 3x
2 1 _ l 3y
2 _ _ l 7
3x _ _ l 4x
3x * * l 3x
3y _ _ l 4y
3y * * l 3y
4x 0 x r 0
4x 1 y r 0
4x _ x r 0
4x * * l 4x    ; skip the x/y's
4y 0 1 * 5
4y 1 0 l 4y
4y _ 1 * 5
4y * * l 4y    ; skip the x/y's
5 x x l 6
5 y y l 6
```

□ Run at full speed

Initial input: 110110 101011

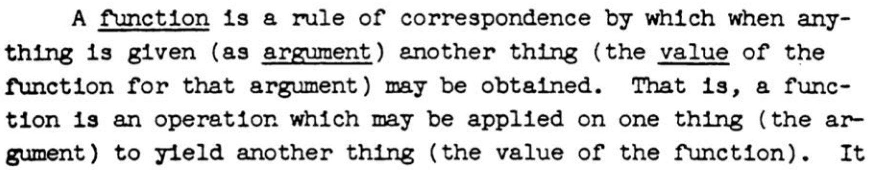
a Turing Machine Simulator, try it out! - http://morphett.info/turing/turing.html?dcfdc7472f99c7f69b5b76b6c238d655

λ-calculus

Interestingly, exactly the same time, an American mathematician, Alonzo Church, also solved the problem – in a completely different way: it was the λ-calculus

Lambda calculus is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution.

in essence: *a function call*



A function is a rule of correspondence by which when anything is given (as argument) another thing (the value of the function for that argument) may be obtained. That is, a function is an operation which may be applied on one thing (the argument) to yield another thing (the value of the function). It

An excerpt from the THE CALCULI OF LAMBDA-CONVERSION – remember, it is 1936 – no computers existed

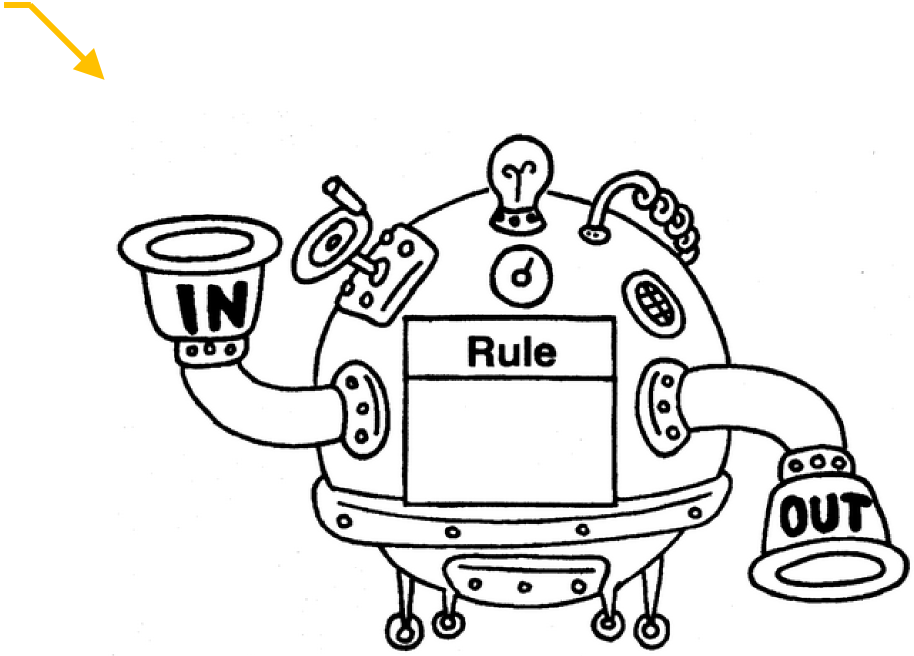
https://en.wikipedia.org/wiki/Lambda_calculus

Basically, that's it! Looks simple and funny, but it is a pure function: one input, one output and no side-effects.

In lambda calculus everything is a function.

There are no loops, numbers and even booleans can be defined as functions – it builds a computational system from simple functions as a building blocks.

Surprisingly, everything, which can be computed with a Turing Machine, can be calculated with lambda calculus also. They are equivalent.



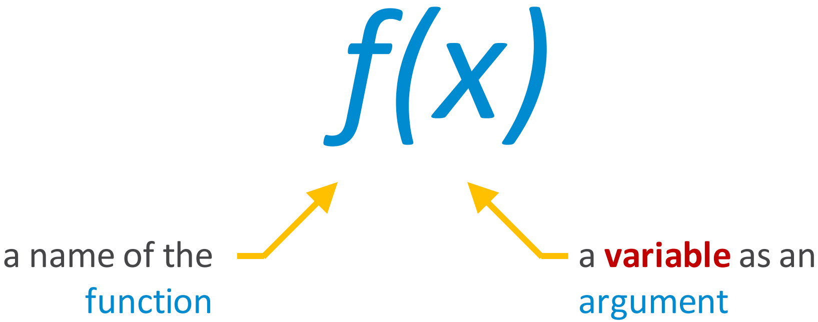
The *function*

We can formalize our robot as a function

The below notation can be read:

- we have a function ***f*** (it does something, like increment a number)
- and f is called with an argument ***x***

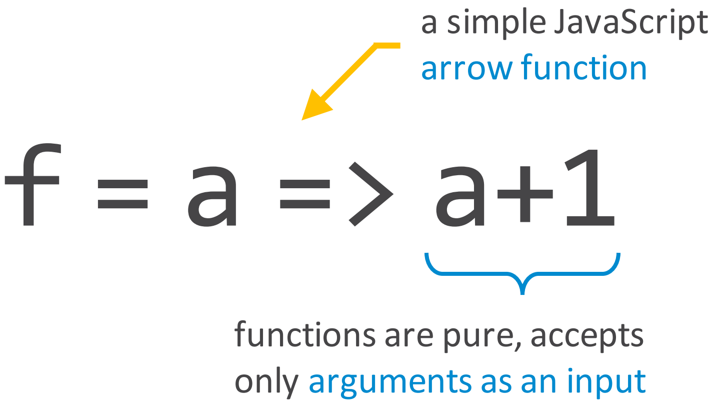
Nothing special, really. The parentheses can be skipped: ***f x***



$$f(x)$$

a name of the function

a **variable** as an argument



a simple JavaScript arrow function

$$f = a => \underbrace{a+1}$$

functions are pure, accepts only arguments as an input

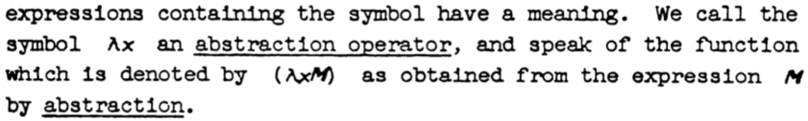
The *lambda*

So what is the lambda then?
It is the anonymous function!

Lambda is a function definition. It can be read:

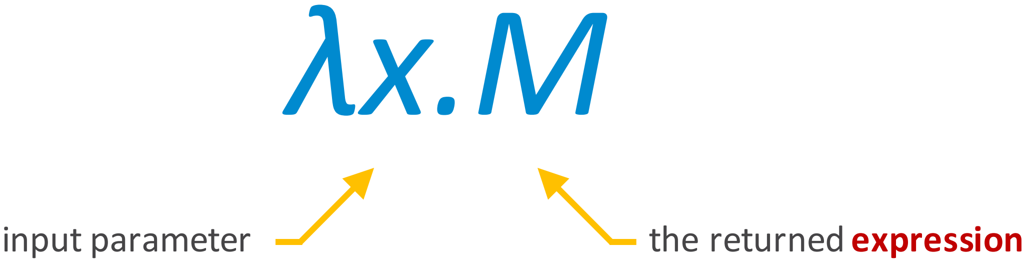
- we have an anonymous function
- it receives a parameter ***x***
- and it returns an expression ***M***
  (it does something, like increment a number)

It is simple as that.



expressions containing the symbol have a meaning. We call the symbol λx an abstraction operator, and speak of the function which is denoted by $(\lambda x M)$ as obtained from the expression M by abstraction.

*Lambda is also called an **abstraction** operator.*



$\lambda x.M$

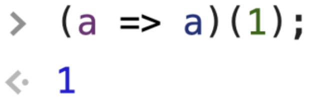
input parameter

the returned **expression**

Currying

functions are unary, take only a single argument
(this function is called Identity, because it returns itself)

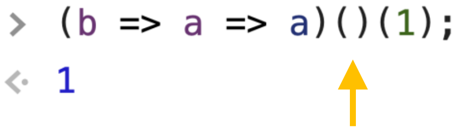
$$\lambda a.a$$



```
> (a => a)(1);
<· 1
```

because of that, we need currying* for multiple arguments

$$\lambda b.\underbrace{\lambda a.a}_{\text{functions can return a function}}$$

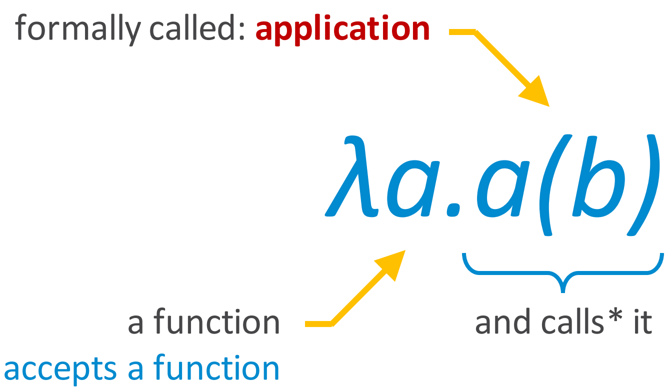


```
> (b => a => a)()(1);
<· 1
```

we just skipped to add an argument for the b parameter

* *the name comes from (guess what) from a mathematician: Haskell Brooks Curry.*
Now you also know why a functional language called Haskell ;)

Passing a function as an argument

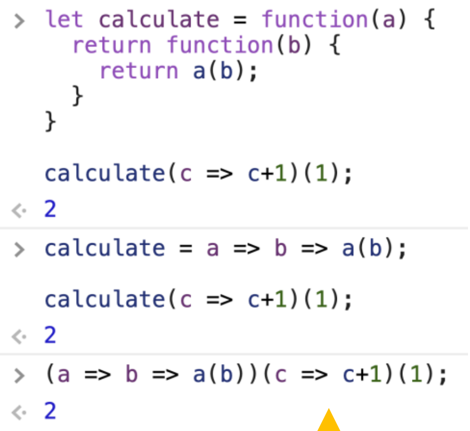


formally called: **application**

$\lambda a.\underbrace{a(b)}_{\text{and calls* it}}$

a function

accepts a function



```
> let calculate = function(a) {
    return function(b) {
      return a(b);
    }
  }

  calculate(c => c+1)(1);
< 2
> calculate = a => b => a(b);

  calculate(c => c+1)(1);
< 2
> (a => b => a(b))(c => c+1)(1);
< 2
```

calling the function, which we got as an argument

the same with arrow functions

as an IIFE expression

passing a function as an argument

Lambda calculus syntax

Basically, you can compute everything with
this simple language:

expression := *variable* | *abstraction* | *application*

abstraction := λ *variable* . *expression*

application := *expression* (*expression*)

parentheses are
usually skipped

Congratulations! You've just learnt the complete lambda calculus!
But we won't ask it in interviews. I promise.

λ-calculus / Functional Programming

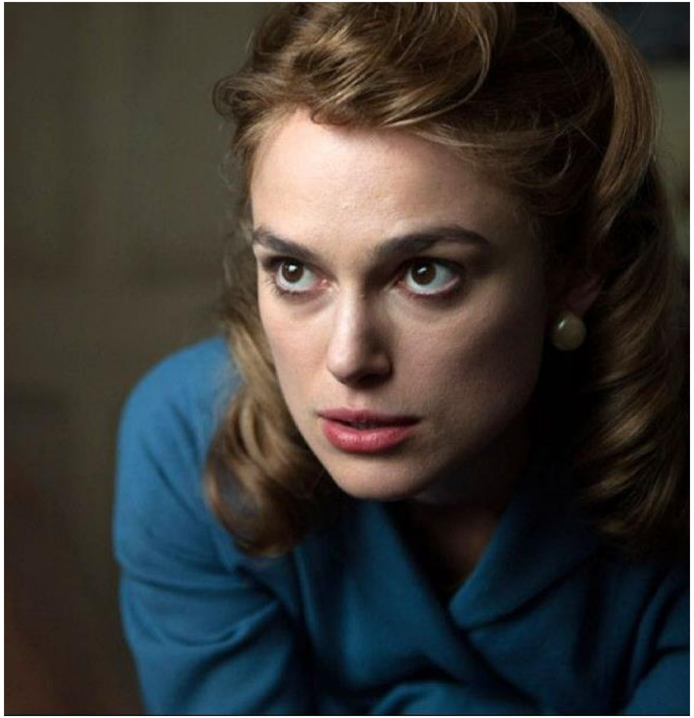
As you may guess now, Functional Programming is the exact implementation of the λ-calculus

…such as the physical computers are the implementation of the Turing Machine.

Do we need to understand the λ-calculus to write good Functional Programming (FP) code? Well, not really, but we do need to recognize that the science behind that is tightly coupled, and FP is not just a paradigm (like Object Oriented Programming), but it is the fundament of computing itself.

From now on, you can make fun with your fellow OOP ninja, that FP is even older than the programming itself, but you should not.

Please be humble and try to understand FP better *than an average OOP developer comprehends what messaging is really in OOP.*



Alan, billions of computers will be built on your invention – and all of them will run λ-calculus. Live with it!

FUNCTIONAL PROGRAMMING

Functional programming

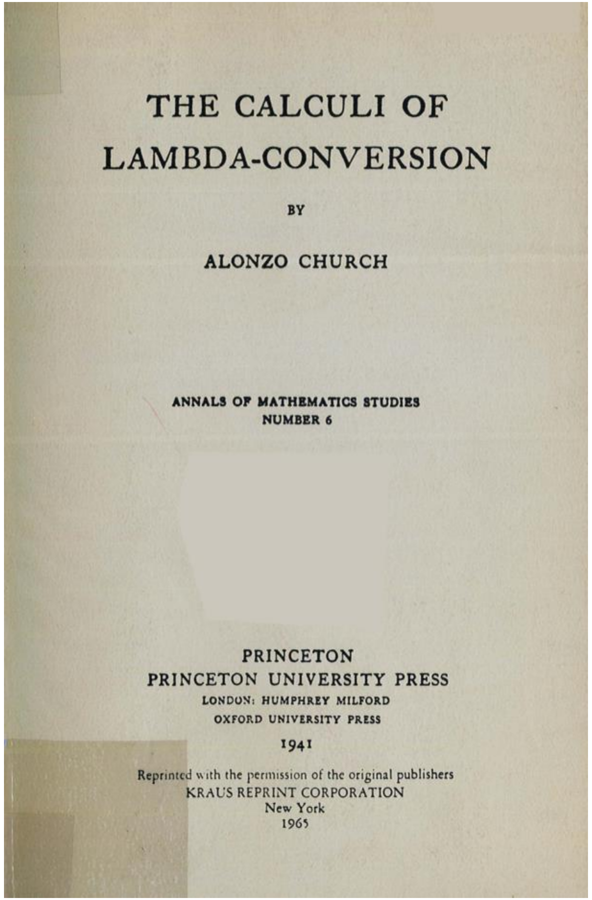
What is functional programming (FP) then?

FP is the application of the principles that Alonzo Church laid down in the *THE CALCULI OF LAMBDA-CONVERSION:*

- utilizes functions exclusively
- functions are pure functions, the only input is the argument
- no side effects, referential transparency
- higher-order functions can accept other functions as a parameter
- first-class functions can be passed as an argument or returned
- values are immutable
- no loops, uses recursion

At this point – if you don't mind – we'd miss the *functors* and *monads*, as maybe there is a little value if we know that a map is a functor, and a promise is a monad.

Let's focus on what is important!



THE CALCULI OF
LAMBDA-CONVERSION

BY

ALONZO CHURCH

ANNALS OF MATHEMATICS STUDIES
NUMBER 6

PRINCETON
PRINCETON UNIVERSITY PRESS
LONDON: HUMPHREY MILFORD
OXFORD UNIVERSITY PRESS
1941

Reprinted with the permission of the original publishers
KRAUS REPRINT CORPORATION
New York
1965

blame him, really…

Referential transparency

It is so lovely that complex phrases can have very simple meanings

Because FP uses pure functions, the output value depends on the input value only.
That is the referential transparency.

a => a+1

the parameter is the single input here, there is no hidden state manipulation

The output depends on the parameter only.

Because of this, the input can be a reference for the output, and the function body does not matter anymore: it is transparent.

As a result, the programmer or the interpreter itself is able to optimize code with memoization, and with other techniques.

think about replacing the function body with a simple key-value dictionary

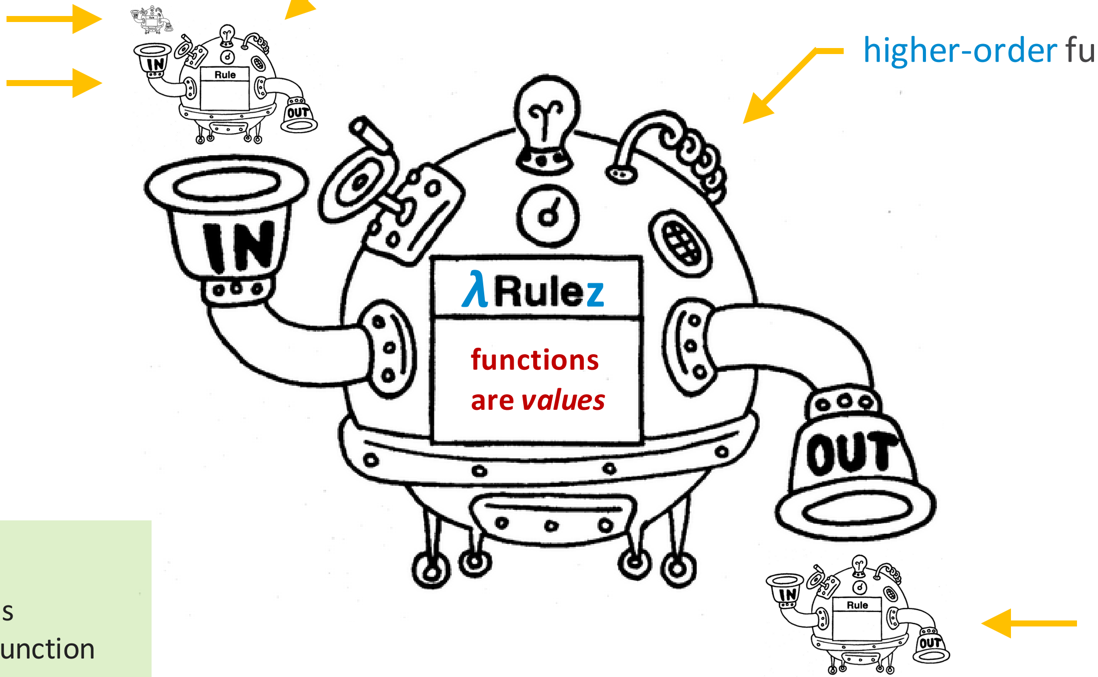
It is also lovely that such a simple things can have significant effect and importance.

... also, a higher-order function

first-class function

first-class function...

higher-order function



first-class function

first-class functions can:

- be stored in variables
- be returned from a function
- be passed as arguments into another function.

No loops

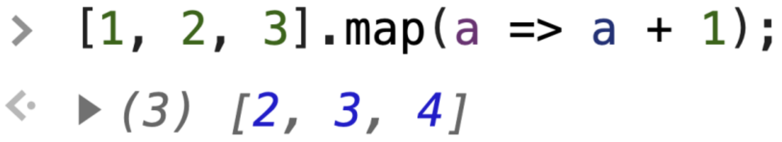
No loops? *Recursion???*

From this point it is maybe clear, that we don’t really use clear FP from every aspects.

While it is possible to build the *whole* application in FP, usually it makes little sense in JavaScript.

However, considering the FP, we can re-evaluate some methods in JavaScript.

Let’s start with the map!



```
> [1, 2, 3].map(a => a + 1);
<· ▶ (3) [2, 3, 4]
```

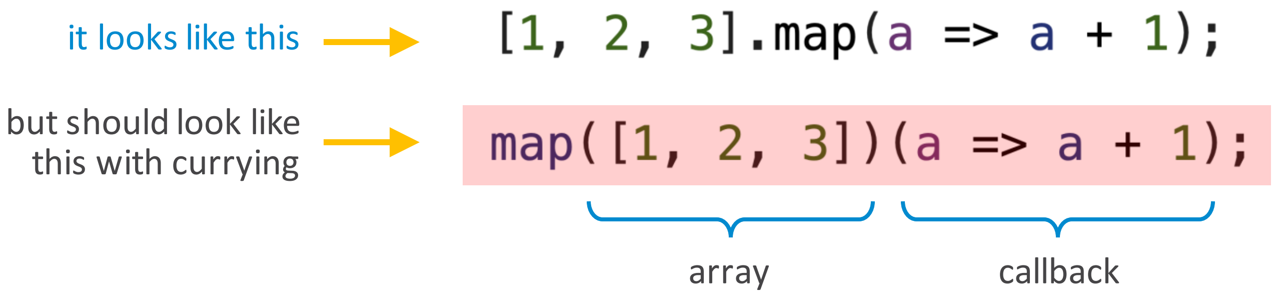
For the first sight, it may seem that it goes step by step, on all the elements of the array, and returns the incremented values.

Right? Not entirely…

Array prototype

First, we need to get a rid of an issue here: map only does have the callback as a parameter, where is the input array?

In JavaScript, the functional *“functions”* are methods on the Array prototype (we’ll have a deep dive on it in the OOP lecture). This has the advantage that because the return value is an array too, it also will have methods (like *.map*), therefore these can be chained.



it looks like this

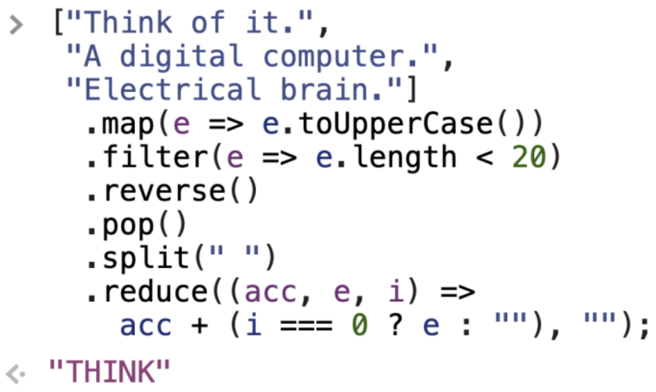
```
[1, 2, 3].map(a => a + 1);
```

but should look like this with currying

```
map([1, 2, 3])(a => a + 1);
```

© 2022 EPAM Systems, Inc.

Chaining



```
> ["Think of it.",
   "A digital computer.",
   "Electrical brain."]
    .map(e => e.toUpperCase())
    .filter(e => e.length < 20)
    .reverse()
    .pop()
    .split(" ")
    .reduce((acc, e, i) =>
      acc + (i === 0 ? e : ""), "");
<· "THINK"
```

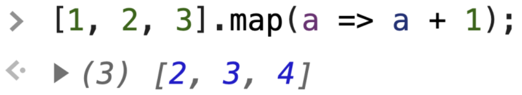
nah, there is no other way to do that

A self-organising team is analysing their career decision after facing with some *simple* functional JavaScript code.

Map

Let’s see from a different viewpoint!

What it really does: it maps (surprise) the values of the input array to another values.



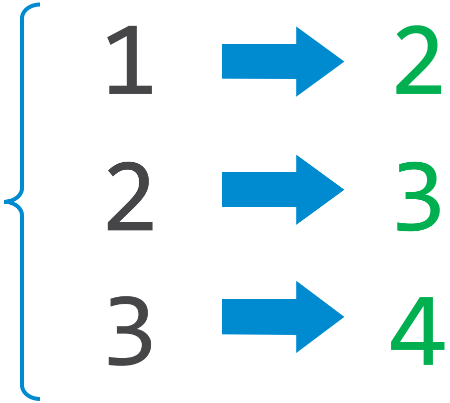
```
> [1, 2, 3].map(a => a + 1);
<· ▶(3) [2, 3, 4]
```

The code that does not iterate.
Well, it does, we just consider now it does not!

Think about it:

- we have input values
- all immutable

Does it really matter in which order we map the elements?



Conceptually, map does *not* mean: iterate.

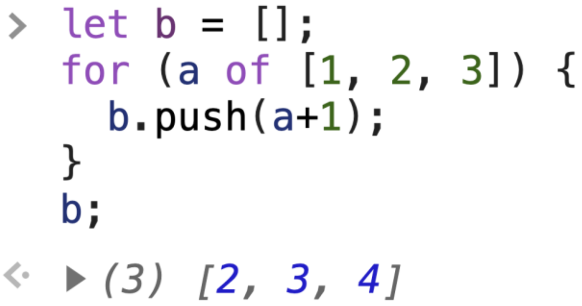
It just assigns values to another values.

If we have 3 processors, we could do it in one step!

Also, because of the referential transparency, we could skip the function body entirely.

Paradigms: imperative, declarative

statements



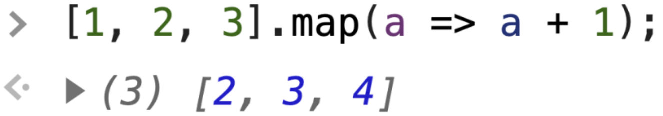
```
> let b = [];
  for (a of [1, 2, 3]) {
    b.push(a+1);
  }
  b;
<· ▶ (3) [2, 3, 4]
```

the imperative way: we explicitly declares the steps – how to do it

the order of running the statements is important

while the callback body can contain statements, it does not really matter: from outside – it is an expression

expression



```
> [1, 2, 3].map(a => a + 1);
<· ▶ (3) [2, 3, 4]
```

the declarative way: we explicitly declares the result – what we expect

the order of evaluating the expressions is not important

Parallel computing + optimizations

You may ask now: JavaScript is a single threaded language - will it utilize 3 cores?

JavaScript is single treaded, but that is only from programming / API perspective. The browsers can and runs the code on multiple cores in some cases – the internals of the processing is entirely the concern of the browser, but if you do mutate state in a *.map callback*, then the browser won't optimize your code, that's for sure.

Writing clean and high-quality code, that can be optimized in future / on browser level is important – if the browser is able to optimize the code, then it is probably less complex and easier to reason about for yourself as well.

A FP code maybe less readable for an inexperienced eye, and it could be very complicated, indeed – but learning it and getting more familiar with it is an absolute necessity and an investment for the future.

Not just from project, but from personal perspective as well.

Joan Clarke likes FP!

Q&A

edu_hu@epam.com