

# Introduction to functional programming and lambda calculus

The INFDEV@HR Team

Hogeschool Rotterdam  
Rotterdam, Netherlands

Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

# Introduction

## Course introduction

- Course topic: what is this course about?
- Examination: how will you be tested?
- Start with course

## Course topic: functional programming

- Lambda calculus
- From lambda calculus to functional programming
- Functional programming using F<sup>#</sup> and Haskell

## Advantages of functional programming

- Strong mathematical foundations
- Easier to reason about programs
- Parallelism for “free”
- Correctness guarantees through strong typing (optional)

Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

## Examination

- Theory exam: test understanding of theory
- Practical exam: test ability to apply theory in practice

## Theory exam: reduction and typing

- One question on reduction in lambda calculus
- One question on typing in lambda calculus,  $F^\sharp$ , or Haskell
- **Passing grade** if both questions answered correctly

## Practical exam: interpreter for a virtual machine

- In a group, build an interpreter for a virtual machine
- According to a specification that will be provided
- Groups may consist of up to 4 students
- Understanding of code tested **individually**



Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

## Lecture topics

- Semantics(meaning) of imperative languages
- Lambda calculus, the foundation for functional languages

# Semantics of imperative languages

## Imperative program: sequence of statements

- Statements directly depend on and alter memory
- Meaning of statements may depend on contents of memory
- Any statement may depend on (read) any memory location
- Any statement may alter any memory location

## Example: meaning of statement sequence

- Statement  $s_1$  changes the machine state from  $S_0$  to  $S_1$
- Statement  $s_2$  changes the machine state from  $S_1$  to  $S_2$
- Run statement  $s_1$ , then run statement  $s_2$ :  $s_1s_2$
- Statement  $s_1s_2$  changes the machine state from  $S_0$  to  $S_2$

$$(S_0 \xrightarrow{s_1} S_1) \wedge (S_1 \xrightarrow{s_2} S_2) \implies S_0 \xrightarrow{s_1s_2} S_2$$

## Example: meaning of statement sequence

- Statement  $s_1$  changes the machine state from  $S_0$  to  $S_1$
- Statement  $s_2$  changes the machine state from  $S_1$  to  $S_2$
- Run statement  $s_1$ , then run statement  $s_2$ :  $s_1s_2$
- Statement  $s_1s_2$  changes the machine state from  $S_0$  to  $S_2$

$$(S_0 \xrightarrow{s_1} S_1) \wedge (S_1 \xrightarrow{s_2} S_2) \implies S_0 \xrightarrow{s_1s_2} S_2$$

*What about  $s_2s_1$ ?*

## Swap order of $s_1 s_2$ : $s_2 s_1$

- Sometimes  $s_2 s_1$  has the same meaning as  $s_1 s_2 \dots$
- Sometimes  $s_2 s_1$  is completely different from  $s_1 s_2$ !
- It depends on  $s_1$ ,  $s_2$ , and the relevant machine state  $S_0$
- It depends on implementation details of  $s_1$  and  $s_2$
- Implementation details matter  $\implies$  leaky abstraction!

## Swap order of $s_1 s_2$ : $s_2 s_1$

- Sometimes  $s_2 s_1$  has the same meaning as  $s_1 s_2 \dots$
- Sometimes  $s_2 s_1$  is completely different from  $s_1 s_2$ !
- It depends on  $s_1$ ,  $s_2$ , and the relevant machine state  $S_0$
- It depends on implementation details of  $s_1$  and  $s_2$
- Implementation details matter  $\implies$  leaky abstraction!

*Can we do better?*

## Idea for better abstraction: remove implicit dependencies

- No implicit dependencies  $\implies$  all dependencies explicit
- No access to arbitrary machine state
- Only explicitly-mentioned state may be accessed



## Idea for better abstraction: remove implicit dependencies

- No implicit dependencies  $\implies$  all dependencies explicit
- No access to arbitrary machine state
- Only explicitly-mentioned state may be accessed

*What if  $s_1$  and  $s_2$  access the same state?*

What if  $s_1\{x\}$  and  $s_2\{x\}$  only read the same state  $x$ ?

- $s_1\{x\}$  calculates  $x + x$ , and  $s_2\{x\}$  calculates the square  $x^2$

*Can we reorder  $s_1\{x\}$  and  $s_2\{x\}$ ?*

# Semantics of imperative languages

Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

What if  $s_1\{x\}$  and  $s_2\{x\}$  only read the same state  $x$ ?

- $s_1\{x\}$  calculates  $x + x$ , and  $s_2\{x\}$  calculates the square  $x^2$

*Can we reorder  $s_1\{x\}$  and  $s_2\{x\}$ ?*

What if  $s_1\{x\}$  and  $s_2\{x\}$  alter the same state  $x$ ?

- $s_1\{x\}$  sets  $x$  to 1, and  $s_2\{x\}$  sets  $x$  to 2

*Can we reorder  $s_1\{x\}$  and  $s_2\{x\}$ ?*

# Semantics of imperative languages

Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

What if  $s_1\{x\}$  and  $s_2\{x\}$  only read the same state  $x$ ?

- $s_1\{x\}$  calculates  $x + x$ , and  $s_2\{x\}$  calculates the square  $x^2$

*Can we reorder  $s_1\{x\}$  and  $s_2\{x\}$ ?*

What if  $s_1\{x\}$  and  $s_2\{x\}$  alter the same state  $x$ ?

- $s_1\{x\}$  sets  $x$  to 1, and  $s_2\{x\}$  sets  $x$  to 2

*Can we reorder  $s_1\{x\}$  and  $s_2\{x\}$ ?*

*Can we do better?*

## Idea for better abstraction: remove implicit dependencies

- No implicit dependencies  $\implies$  all dependencies explicit
- No reading of arbitrary machine state
- No mutating of arbitrary machine state
- Only explicitly-mentioned machine state may be read

## Idea for better abstraction: remove implicit dependencies

- No implicit dependencies  $\implies$  all dependencies explicit
- No reading of arbitrary machine state
- No mutating of arbitrary machine state
- Only explicitly-mentioned machine state may be read

NB: No provision at all is made for mutating machine state

## Wait a minute, this is just like functions

- Not statements, but (mathematical) functions
- Functions depend only on arguments
- Functions do not mutate machine state
- Can calculate function value when all arguments are known
- Can always replace a function call by its value

# Semantics of imperative languages

Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

## Wait a minute, this is just like functions

- Not statements, but (mathematical) functions
- Functions depend only on arguments
- Functions do not mutate machine state
- Can calculate function value when all arguments are known
- Can always replace a function call by its value

**NB: Imperative “functions” need not be functions!**

Non-function “functions” are more properly called procedures



## Referential transparency:

It is always valid to replace a function call by its value

## Referential transparency:

It is always valid to replace a function call by its value

## Advanced topic:

Allow mutation of state without losing referential transparency

Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

# Lambda calculus

## What is lambda calculus?

- Model of computation based on functions
- Completely different from Turing machines, but equivalent
- Foundation of all functional programming languages
- Truly tiny when compared with its power
- Consists of only (function) abstraction and application

A lambda calculus term is one of three things:

- a variable (from some arbitrary infinite set of variables)
- an abstraction (a “function of one variable”)
- an application (a “function call”)

## Variables (arbitrary infinite set):

$a, b, c, \dots$

$a_0, a_1, \dots$

$b_0, b_1, \dots$

## Abstractions:

For any variable  $x$  and lambda term  $T$ :  $(\lambda x.T)$

## Applications:

For any lambda terms  $F$  and  $T$ :  $(FT)$

A simple example: the identity function (just returns its input)

$$(\lambda x.x)$$

A simple example: call the identity function on a variable

$$((\lambda x.x) \ v)$$



## $\beta$ -reduction

- Redex: application of an abstraction to an argument
- Result: in abstraction body replace parameter by argument

$$((\lambda x.B)A) \rightarrow_{\beta} B[x \mapsto A]$$

Multiple parameters via nested abstractions:

$$(\lambda x. (\lambda y. (xy)))$$

The parameters are then given one at a time:

$$(((\lambda x \ y. (x \ y)) \ A) \ B)$$

$$(((\lambda x \ y. (x \ y)) \ A) \ B)$$

$$(((\lambda x \ y. (x \ y)) \ A) \ B)$$
$$((\lambda x \ y. (x \ y)) \ A) \ B$$

$$((\lambda x. y. (x \ y)) \ A) \ B$$

$$((\lambda x. y. (x \ y)) \ A) \ B$$
$$((\lambda y. (A \ y)) \ B)$$

$$((\lambda y.(A \ y)) \ B)$$

$$((\lambda y.(A \ y)) \ B)$$
$$((\lambda y.(A \ y)) \ B)$$



$$((\lambda y.(A \ y)) \ B)$$

$$((\lambda y.(A \ y)) \ B)$$
$$(A \ B)$$

Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

# Closing up

## Example executions of (apparently) nonsensical programs

- Manual execution of various lambda programs.
- Try to work out the result of these programs.

*What is the result of this program?*

$$(((\lambda x \ y. (x \ y)) (\lambda z. (z \ z))) A)$$

$$(((\lambda x \ y. (x \ y)) \ (\lambda z. (z \ z)))) \ A)$$

```
(((\lambda x y.(x y)) (\lambda z.(z z))) A)
```

```
( ((\lambda x y.(x y)) (\lambda z.(z z))) A )
```

```
( ((λx y.(x y)) (λz.(z z))) A )
```



$$((\lambda x. y. (x \ y)) (\lambda z. (z \ z))) \ A)$$
$$((\lambda y. ((\lambda z. (z \ z)) \ y)) \ A)$$

Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

$$((\lambda y.((\lambda z.(z \ z)) \ y)) \ A)$$

$$((\lambda y.((\lambda z.(z\ z))\ y))\ A)$$
$$((\lambda y.((\lambda z.(z\ z))\ y))\ A)$$

Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

```
((λy.((λz.(z z)) y)) A)
```

$$((\lambda y.((\lambda z.(z \ z)) \ y)) \ A)$$
$$((\lambda z.(z \ z)) \ A)$$

$$((\lambda z.(z \ z)) \ A)$$

$$((\lambda z.(z \ z)) \ A)$$
$$((\lambda z.(z \ z)) \ A)$$

Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

$$((\lambda z.(z\ z))\ A)$$



$$((\lambda z.(z\ z))\ A)$$
$$(A\ A)$$

*What is this program's result? Hint: scope!*

```
(( $(\lambda x \ x.(x \ x))$  A) B)
```

Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

$$(((\lambda x \ x.(x \ x)) \ A) \ B)$$

```
((λx x.(x x)) A) B)
```

```
( ((λx x.(x x)) A) B)
```

Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

$$((\lambda x. x. (x x)) A) B$$

$$((\lambda x. (x \ x)) \ A) \ B$$
$$((\lambda x. (x \ x)) \ B)$$

Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

$$((\lambda x. (x \ x)) \ B)$$

$$((\lambda x. (x \ x)) \ B)$$
$$((\lambda x. (x \ x)) \ B)$$



Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

$$((\lambda x.(x\ x))\ B)$$

$$((\lambda x.(x\ x))\ B)$$
$$(B\ B)$$

Outer  $x$  is shadowed by inner  $x$ !

```
((λx x.(x x)) A) B)
```

To disambiguate, turn:

$$(((\lambda x \ x.(x \ x)) \ A) \ B)$$

into:

$$(((\lambda y \ x.(x \ x)) \ A) \ B)$$

Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

$$(((\lambda y \ x.(x \ x)) \ A) \ B)$$

$$(((\lambda y \ x.(x \ x)) \ A) \ B)$$
$$((\lambda y \ x.(x \ x)) \ A) \ B$$

Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

$$((\lambda y \ x. (x \ x)) \ A) \ B$$

$$((\lambda y. (x \ x)) \ A) \ B$$
$$((\lambda x. (x \ x)) \ B)$$



Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

$$((\lambda x. (x \ x)) \ B)$$

$$((\lambda x. (x \ x)) \ B)$$
$$((\lambda x. (x \ x)) \ B)$$

Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

$$((\lambda x.(x\ x))\ B)$$

```
((λx.(x x)) B)
```

```
( B B )
```

*What is this program's result? Is there even one?*

```
((λx.(x x)) (λx.(x x)))
```

Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

$$((\lambda x. (x \ x)) \ (\lambda x. (x \ x)))$$

$$((\lambda x. (x \ x)) \ (\lambda x. (x \ x)))$$
$$((\lambda x. (x \ x)) \ (\lambda x. (x \ x)))$$

```
((λx.(x x)) (λx.(x x)))
```



$$((\lambda x.(x\ x))\ (\lambda x.(x\ x)))$$
$$(\ (\lambda x.(x\ x))\ (\lambda x.(x\ x))\ )$$

Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

$$((\lambda x. (x \ x)) \ (\lambda x. (x \ x)))$$

$$((\lambda x. (x \ x)) \ (\lambda x. (x \ x)))$$
$$((\lambda x. (x \ x)) \ (\lambda x. (x \ x)))$$

Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

$$((\lambda x. (x \ x)) (\lambda x. (x \ x)))$$

$$((\lambda x.(x\ x))\ (\lambda x.(x\ x)))$$
$$(\ (\lambda x.(x\ x))\ (\lambda x.(x\ x))\ )$$

```
((λx.(x x)) (λx.(x x)))
```

It never ends! Like a while true: ..!

# Closing up

Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

What you are all thinking:

This is no real programming language!

Is this a joke?

- We have some sort of functions and function calls
- We do not have booleans and if's
- We do not have integers and arithmetic operators
- We do not have a lot of things!

What you are all thinking:

This is no real programming language!

Is this a joke?

- We have some sort of functions and function calls
- We do not have booleans and if's
- We do not have integers and arithmetic operators
- We do not have a lot of things!

Surprise!

All these things are in there awaiting discovery!



# Closing up

Introduction  
to functional  
programming  
and lambda  
calculus

The  
INFDEV@HR  
Team

Introduction

Semantics of  
imperative  
languages

Lambda  
calculus

Closing up

What you are all thinking:

This is no real programming language!

Is this a joke?

- We have some sort of functions and function calls
- We do not have booleans and if's
- We do not have integers and arithmetic operators
- We do not have a lot of things!

Surprise!

All these things are in there awaiting discovery!

Stay tuned

This will be a marvelous voyage!

The best of luck, and thanks for the  
attention!