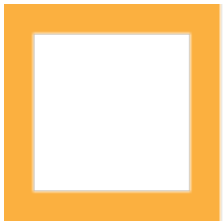
# Lisp
## Functional Programming

Jens Egholm Pedersen and Anders Kalhauge



cphbusiness

Spring 2017

# Outline



Lambda calculus

Higher order functions

Lisp syntax
- Exercise

Map and flat map
- Hand-in

# Application as data

A computer is a thing that follows an algorithm = computation.

# Application as data

A computer is a thing that follows an algorithm = computation.

What can be computed?

# Application as data

A computer is a thing that follows an algorithm = computation.

What can be computed? Memory!

# Application as data

A computer is a thing that follows an algorithm = computation.

What can be computed? Memory!

A computer basically treats your applications as memory.

# Application as data



A computer is a thing that follows an algorithm = computation.

What can be computed? Memory!

A computer basically treats your applications as memory.

If we can treat functions as memory, they simply become data

# Lambda calculus



Invented by Alonzo Church in the 1930. *Before* computers!

# Lambda calculus

Invented by Alonzo Church in the 1930. *Before* computers!

$$square\_sum(x, y) = x^2 + y^2$$

# Lambda calculus

Invented by Alonzo Church in the 1930. *Before* computers!

$square\_sum(x, y) = x^2 + y^2$

$(x, y) \mapsto x^2 + y^2$ The pair $x$ and $y$ is *mapped to* $x^2 + y^2$.

# Lambda calculus

Invented by Alonzo Church in the 1930. *Before* computers!

$square\_sum(x, y) = x^2 + y^2$

$(x, y) \mapsto x^2 + y^2$ The pair $x$ and $y$ is *mapped to* $x^2 + y^2$.

$x \mapsto (y \mapsto x^2 + y^2)$

# Lambda calculus

Invented by Alonzo Church in the 1930. *Before* computers!

$square\_sum(x, y) = x^2 + y^2$

$(x, y) \mapsto x^2 + y^2$ The pair $x$ and $y$ is *mapped to* $x^2 + y^2$.

$x \mapsto (y \mapsto x^2 + y^2)$

$f(5)(2) = 29$

# Lambda calculus

Invented by Alonzo Church in the 1930. *Before* computers!

$square\_sum(x, y) = x^2 + y^2$

$(x, y) \mapsto x^2 + y^2$ The pair $x$ and $y$ is *mapped to* $x^2 + y^2$.

$x \mapsto (y \mapsto x^2 + y^2)$

$f(5)(2) = 29$

$f(5) = ??$

# Lambda calculus

Invented by Alonzo Church in the 1930. *Before* computers!

$square\_sum(x, y) = x^2 + y^2$

$(x, y) \mapsto x^2 + y^2$ The pair $x$ and $y$ is *mapped to* $x^2 + y^2$.

$x \mapsto (y \mapsto x^2 + y^2)$

$f(5)(2) = 29$

$f(5) = ??$

$f(5) = y \mapsto y^2 + 25$

# Lambda calculus

Invented by Alonzo Church in the 1930. *Before* computers!

$square\_sum(x, y) = x^2 + y^2$

$(x, y) \mapsto x^2 + y^2$ The pair $x$ and $y$ is *mapped to* $x^2 + y^2$.

$x \mapsto (y \mapsto x^2 + y^2)$

$f(5)(2) = 29$

$f(5) = ??$

$f(5) = y \mapsto y^2 + 25 \quad = \lambda y.y^2 + 25$

# Lambda calculus

Invented by Alonzo Church in the 1930. *Before* computers!

$square\_sum(x, y) = x^2 + y^2$

$(x, y) \mapsto x^2 + y^2$ The pair $x$ and $y$ is *mapped to* $x^2 + y^2$.

$x \mapsto (y \mapsto x^2 + y^2)$

$f(5)(2) = 29$

$f(5) = ??$

$f(5) = y \mapsto y^2 + 25 \quad = \lambda y.y^2 + 25$

Known as **currying**

- evaluating a function with multiple arguments in a sequence

# Higher order function 1/2



A function that either:

- Takes a function as an argument
- Returns a function as its result

# Higher order function 1/2



A function that either:

- Takes a function as an argument
- Returns a function as its result

**First part:** Returning a function

# Higher order function 1/2

A function that either:

- Takes a function as an argument
- Returns a function as its result

**First part:** Returning a function

$$x \mapsto (y \mapsto x + y)$$

# Higher order function 1/2

A function that either:

- Takes a function as an argument
- Returns a function as its result

**First part:** Returning a function

$$x \mapsto (y \mapsto x + y)$$

$$25 \mapsto (y \mapsto 25 + y)$$

# Imperative programming

- Series of instructions

# Imperative programming

- Series of instructions
- Variables in memory

# Imperative programming



- Series of instructions
- Variables in memory = global state

# Imperative programming

cphbusiness
COPENHAGEN BUSINESS ACADEMY

- Series of instructions
- Variables in memory = global state

- This is how a CPU works

# Imperative programming



- Series of instructions
- Variables in memory = global state

- This is how a CPU works
- 1. Input: setting memory
  2. **black box magic**
  3. Output: setting memory

# Imperative programming



- Series of instructions
- Variables in memory = global state

- This is how a CPU works
- 1. Input: setting memory
  2. **black box magic**
  3. Output: setting memory

- **Statements** that changes a program's **state**

# Side effects

**State** = The values in your memory

# Side effects

**State** = The values in your memory

**Mutability** = Changing state

# Side effects

**State** = The values in your memory

**Mutability** = Changing state

**Side effects** = Behaviour outside scope

# Side effects



**State** = The values in your memory

**Mutability** = Changing state

**Side effects** = Behaviour outside scope

**Mutability + Concurrency** =

# Side effects



**State** = The values in your memory

**Mutability** = Changing state

**Side effects** = Behaviour outside scope

**Mutability + Concurrency** = Disaster

Mathematical functions does not have

- State

# Functional programming

Mathematical functions does not have

- State
- Side effects

# Functional programming

Mathematical functions does not have

- State
- Side effects

Data is changed using $\mapsto$ - immutably

# Functional programming

Mathematical functions does not have

- State
- Side effects

Data is changed using $\mapsto$ - immutably

**Immutability + concurrency** $=$

# Functional programming

Mathematical functions does not have

- State
- Side effects

Data is changed using $\mapsto$ - immutably

**Immutability + concurrency** = World domination

# Higher order functions 2/2

cphbusiness
COPENHAGEN BUSINESS ACADEMY

**Part two:** Taking functions as input

# Higher order functions 2/2

cphbusiness
COPENHAGEN BUSINESS ACADEMY

**Part two:** Taking functions as input

Where could this be useful?

# Higher-order functions in Java 1/2

cphbusiness
COPENHAGEN BUSINESS ACADEMY

```
interface DoSomething {
  void something(int i);
}
```

# Higher-order functions in Java 1/2

cphbusiness
COPENHAGEN BUSINESS ACADEMY

```
interface DoSomething {
  void something(int i);
}
```

```
interface List<T> {
  void foreach(DoSomething function);
}
```

# Higher-order functions in Java 1/2

cphbusiness
COPENHAGEN BUSINESS ACADEMY

```
interface DoSomething {
  void something(int i);
}
```

```
interface List<T> {
  void foreach(DoSomething function);
}
```

```
  myList.foreach(
    new DoSomething() {
      void something(int i) { return i * 2; }
    }
  );
```

# Higher-order functions in Java 2/2

cphbusiness
COPENHAGEN BUSINESS ACADEMY

How many functions can we choose from in `DoSomething`?

# Higher-order functions in Java 2/2

How many functions can we choose from in `DoSomething`?

```
@FunctionalInterface
interface DoSomething {
  void something(int i);
}
```

# Higher-order functions in Java 2/2



How many functions can we choose from in `DoSomething`?

```
@FunctionalInterface
interface DoSomething {
  void something(int i);
}
```

```
interface List<T> {
  void foreach(DoSomething function);
}
```

# Higher-order functions in Java 2/2

cphbusiness
COPENHAGEN BUSINESS ACADEMY

How many functions can we choose from in `DoSomething`?

```
@FunctionalInterface
interface DoSomething {
  void something(int i);
}
```

```
interface List<T> {
  void foreach(DoSomething function);
}
```

```
  myList.foreach(item -> item + 2);
```

# Lisp syntax

cphbusiness
COPENHAGEN BUSINESS ACADEMY

| | | |
|---|---|---|
| Boolean | `T` and `nil` | |
| Conditional | `(if expr then else)` | `(if (= 0 0) x y)` |
| Lists | `(list elements)` or | |
| | `(cons tail)` | `(list 1 2)` or |
| | | `(cons 1 (cons 2 nil))` |
| Let binding | `(let ((variables))` | |
| | `(body))` | `(let ((a 10)) a)` |
| Functions | `(defun name` | |
| | `(arguments) body)` | `(defun sum (a b) (+ a b)` |
| Lambda | `(lambda (arguments))` | |
| | `body` | |
| | | `(lambda (a b) (+ a b))` |

# Lisp exercise

Clone the `lisp-exercises` from `cphbus-functional-programming`

`https://github.com/cphbus-functional-programming/lisp-exercises`

Work on the `function.lisp` file

# Map and flat map



What was common about the exercises?

# Map and flat map



What was common about the exercises?

What was the input?

# Map and flat map



What was common about the exercises?

What was the input?

What was the output?

# Map and flat map

What was common about the exercises?

What was the input?

What was the output?

This is called **mapping**: $x \mapsto y$

# Mapping

Mapping from one side to the other.

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \mapsto \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix} \tag{1}$$

Mapping from one side to the other.

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \mapsto \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix} \tag{1}$$

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \mapsto \begin{bmatrix} 5 \\ 6 \\ 7 \end{bmatrix} \tag{2}$$

# Flattening

Flattens a two dimensional list into one dimension

# Flat map

Flattens a two dimensional list into one dimension, and uses `map` on the output

# Hand-in

Clone the `lisp-exercises` from `cphbus-functional-programming`

`https://github.com/cphbus-functional-programming/lisp-exercises`

Work on the hand-in in the `flatmap.lisp` file.

- Implement a `map` function
- Implement a `flatten` function
- Implement a `flatmap` function