Lisp
Functional Programming

Jens Egholm Pedersen and Anders Kalhauge

cphbusiness

Spring 2017

Outline

Lambda calculus

Lisp syntax

- Exercises 1

Higher order functions

Lisp syntax

- Exercises 2

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* $x^2 + y^2$.

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

$f(5)(2) = 29$

$f(5) =??$

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

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

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

Lisp syntax

Boolean `T` and `nil`

Lisp syntax

Boolean	`T` and `nil`	
Conditional	`(if expr then else)`	`(if (= 0 0) x y)`

Lisp syntax

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)))`

Lisp syntax

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)`

Lisp syntax

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))`

Lisp syntax

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 syntax: Lambda

Lambdas are simply just functions without a name

Lisp syntax: Lambda

Lambdas are simply just functions without a name

Lambdas are defined as `(lambda (arguments) body)`

Lisp syntax: Lambda

Lambdas are simply just functions without a name

Lambdas are defined as `(lambda (arguments) body)` `(lambda (a) (+ a 2))`

To avoid confusing them with normal functions, use `funcall` to call them

Lisp syntax: Lambda

Lambdas are simply just functions without a name

Lambdas are defined as `(lambda (arguments) body)` `(lambda (a) (+ a 2))`

To avoid confusing them with normal functions, use `funcall` to call them

```
(funcall (lambda a (+ a 2)) 5)
```

Lisp exercises

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

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

Work on the `lambda.lisp` file

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

- Series of instructions
- Variables in memory = global state
- Seen in a CPU

Imperative programming

- Series of instructions
- Variables in memory = global state
- Seen in a CPU
- Interacts with side-effects

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

Functional programming

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

Part two: Taking functions as input

Higher order functions 2/2

Part two: Taking functions as input

Where could this be useful?

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

Higher order functions in Java

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

Higher order functions in Java

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

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

Lisp syntax

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 exercises

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?

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

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