Lisp
Functional Programming

Jens Egholm Pedersen and Anders Kalhauge

Spring 2018

Outline

Course and lecture formalia

Lisp introduction

- Linguistics

Lisp

- Installation
- Lisp syntax
- Variables in Lisp

Lambda calculus 1/2

- Functions in Lisp

Lists in Lisp

- Linked lists

Lambda calculus 2/2

Lisp syntax

Knowledge of:

- Functional programming paradigm
- Building blocks of a functional programming language
- How to support parallelism using a functional language
- Where to find additional information

Skills:

- Write basic web applications in Elm
- Understand and write programs in Lisp
- Understand and write simple programs in Haskell

Before we begin...

- Lecture format
- Learning to learn
- Metacognition

Lectures

- You have one job, and one job only

Lectures

- You have one job, and one job only
- Practical part - use your computer

Lectures

- You have one job, and one job only
- Practical part - use your computer
- Theoretical part - do *not* use your computer

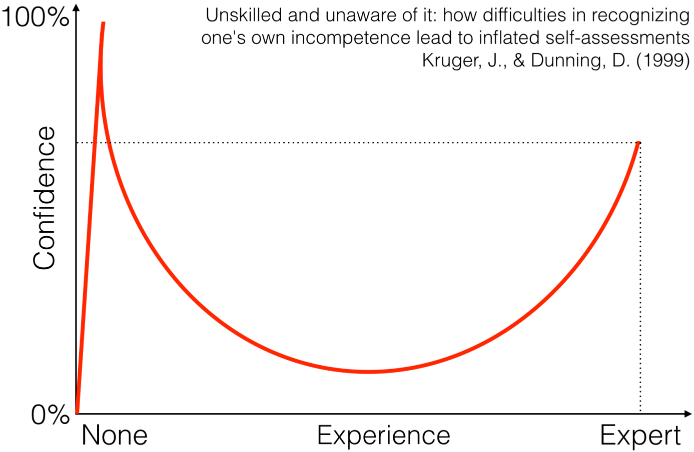
Memory is formed when you pay attention.

Learning to learn

Memory is formed when you pay attention.

- Computers and phones *seriously* distractions
- Take notes (internalisation)
- Try to understand and *relate* (Bloom's taxonomy)

Dunning-Kruger Effect



Metacognition

- Cognition $\approx$ thinking

- Cognition $\approx$ thinking
- *Meta*cognition $\approx$ thinking about thinking

Metacognition

- Cognition $\approx$ thinking
- *Meta*cognition $\approx$ thinking about thinking
- What would you like to get from this course?

Metacognition

- Cognition $\approx$ thinking
- *Meta*cognition $\approx$ thinking about thinking

- What would you like to get from this course?
- What is your level of ambition?

Metacognition

- Cognition $\approx$ thinking
- *Meta*cognition $\approx$ thinking about thinking

- What would you like to get from this course?
- What is your level of ambition?
- Could you change something?

Metacognition

- Cognition $\approx$ thinking
- *Meta*cognition $\approx$ thinking about thinking

- What would you like to get from this course?
- What is your level of ambition?
- Could you change something?

- *Please* experiment. And *please* be critical

Agenda for today

- Lisp and linguistics
- Lambda calculus 1/2
- Functions in Lisp
- Exercise
- Lists in Lisp
- Lambda calculus 2/2
- Exercise 2

Lisp

- Specified in 1958
- One of the oldest high-level programming languages
- Prefix notation
- First language to use lambda calculus

Programming language

- Low versus high abstraction
- *Computer think* are not for humans
- Can it be generalised?

Linguistics: Language science
Traditionally occupied with human language.

Noam Chomsky: Chomsky hierarchy

Type-3 grammar	Regular language (state automata)
Type-2 grammar	Context-free (no ambiguity)
Type-1 grammar	Context-sensitive (ambiguity)
Type-0 grammar	Unrestricted grammar (no restrictions on I/O)

One of the first higher-level programming languages

One of the first higher-level programming languages

Pioneered many inventions: **tree structures**, **dynamic types**, **higher-order functions** and many more

One of the first higher-level programming languages

Pioneered many inventions: **tree structures**, **dynamic types**, **higher-order functions** and many more

LISt **P**rocessor: everything in Lisp is `Lists`

One of the first higher-level programming languages

Pioneered many inventions: **tree structures**, **dynamic types**, **higher-order functions** and many more

LISt **P**rocessor: everything in Lisp is `Lists`

"The most intelligent way to misuse a computer" - Edgar W. Dijkstra

One of the first higher-level programming languages

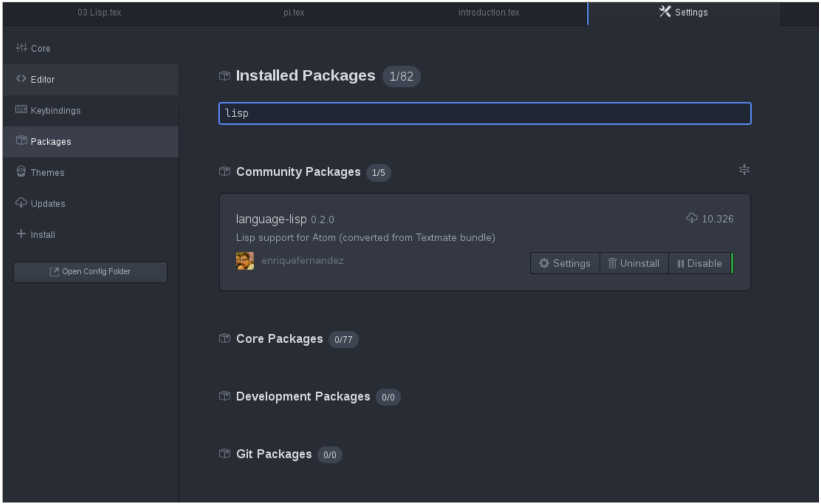
Pioneered many inventions: **tree structures**, **dynamic types**, **higher-order functions** and many more

LISt **P**rocessor: everything in Lisp is `Lists`

"The most intelligent way to misuse a computer" - Edgar W. Dijkstra

Many (!) dialects: `Scheme`, `Common Lisp`, `Emacs Lisp`, `AutoLisp`, `Racket`, `Clojure` (JVM), **`CLisp`**

Installing Lisp package in Atom



Installing CLisp

Go to `http://clisp.org/` and:

On Windows Download the Cygwin package by running the Cygwin installer

On Unix Download the package for your system or build it from source

Integration in Atom

The `atom-slime` Atom package provides compilation functionality

Integration in Atom

The `atom-slime` Atom package provides compilation functionality

See: `https://atom.io/packages/atom-slime`

Lisp syntax

- Prefix notation: *Function first* then arguments
- Function call surrounded by parenthesis

Lisp syntax

- Prefix notation: *Function first* then arguments
- Function call surrounded by parenthesis

```
(+ 1 1)
```

Lisp syntax

- Prefix notation: *Function first* then arguments
- Function call surrounded by parenthesis

```
(+ 1 1)

(* 1 (+ 2 3))
```

Lisp syntax

cphbusiness COPENHAGEN BUSINESS ACADEMY

- Prefix notation: *Function first* then arguments
- Function call surrounded by parenthesis

```
(+ 1 1)

(* 1 (+ 2 3))

(write (- 5 2))
```

Exercise 1

1.1: Divide $5 + 3$ with $4 - 2$

Exercise 1

1.1: Divide $5 + 3$ with $4 - 2$

1.2: Write $9 * 2 - 3 + 5$ to the console

Basic Lisp syntax

- Arithmetic: `* + /`
- Function call: `(name [arg1 [arg2 [arg3...]]])`
- Comments (`;;`): `(+ 3 1) ;; 4`

Procedural programming

`(setf variable 10)` ← mutable

Procedural programming

`(setf variable 10)` ← mutable

Functional programming

Local variables: `let`-binding

Procedural programming

`(setf variable 10)` ← mutable

Functional programming

Local variables: `let`-binding

```
(let ((a 10)) (write a) )
```

Procedural programming

`(setf variable 10)` ← mutable

Functional programming

Local variables: `let`-binding

```
(let ((a 10)) (write a) )
```

Why is the `let`-binding preferred in functional programming?

Introduction exercise

cphbusiness COPENHAGEN BUSINESS ACADEMY

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

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

Work on the `variables.lisp` file

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

A computer is a thing that follows an algorithm = computation.
Imagine living in 1900; How do you 'compute'?

A computer is a thing that follows an algorithm = computation.
Imagine living in 1900; How do you 'compute'?
What do you have to work with?

A computer is a thing that follows an algorithm = computation.
Imagine living in 1900; How do you 'compute'?
What do you have to work with?
Mathematics!

Lambda calculus

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

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

$sum(x, y) = x + y$

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

$sum(x, y) = x + y$

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

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

$sum(x, y) = x + y$

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

$x \mapsto (y \mapsto ...)$

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

$sum(x, y) = x + y$

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

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

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

$sum(x, y) = x + y$

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

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

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

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

$sum(x, y) = x + y$

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

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

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

$f(5)(2) =$

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

$sum(x, y) = x + y$

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

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

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

$f(5)(2) = 7$

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

$sum(x, y) = x + y$

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

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

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

$f(5)(2) = 7$

$f(5) =$

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

$sum(x, y) = x + y$

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

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

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

$f(5)(2) = 7$

$f(5) = y \mapsto y + 5$

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

$sum(x, y) = x + y$

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

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

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

$f(5)(2) = 7$

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

Lisp functions

- Functions defined with `defun`
- Takes three expressions: name, arguments and function body

Lisp functions

- Functions defined with `defun`
- Takes three expressions: name, arguments and function body

```
(defun test (a) (write a))
```

Lisp functions

- Functions defined with `defun`
- Takes three expressions: name, arguments and function body

```
(defun test (a) (write a))

(test 10)
```

```
(lambda () ())
```

Lambdas in Lisp

```
(lambda () ())
```

```
(lambda (x) (* x x))
```

Lambdas in Lisp

```
(lambda () ())

(lambda (x) (* x x))

((lambda (x) (* x x)) 5)
```

What do you need to know in an `if` statement?

`if` statements

What do you need to know in an `if` statement?

```
(if condition then else)
```

What do you need to know in an `if` statement?

```
(if condition then else)
```

eq: Equality function

```
(eq 1 2) ;; false
```

What do you need to know in an `if` statement?

```
(if condition then else)
```

`eq`: Equality function

```
(eq 1 2) ;; false
```

```
(if (= a 0) 0 1)
```

Lists are made by calling the function `list` followed by the list content

Lists are made by calling the function `list` followed by the list content

```
(list 10 5 2)
```

Lists

cphbusiness
COPENHAGEN BUSINESS ACADEMY

Lists are made by calling the function `list` followed by the list content

`(list 10 5 2)` $\mapsto$ `[10, 5, 2]`

Lists are made by calling the function `list` followed by the list content

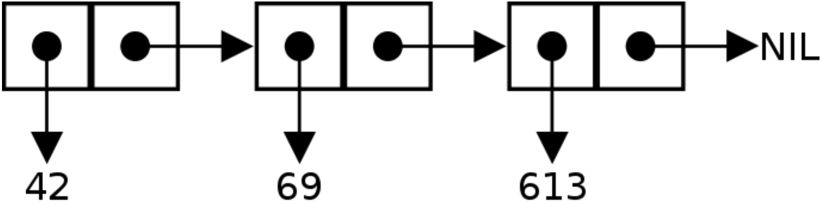
`(list 10 5 2)` $\mapsto$ `[10, 5, 2]`

```
(list 10 (list 5 2))
```

Lists are made by calling the function `list` followed by the list content

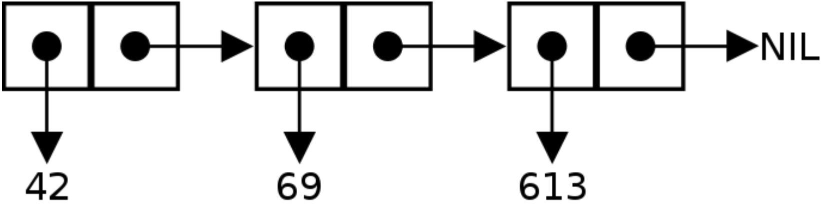
`(list 10 5 2)` $\mapsto$ `[10, 5, 2]`

`(list 10 (list 5 2))` $\mapsto$ `[10, [5, 2]]` Lists in lisp is built using *linked lists*



Linked lists

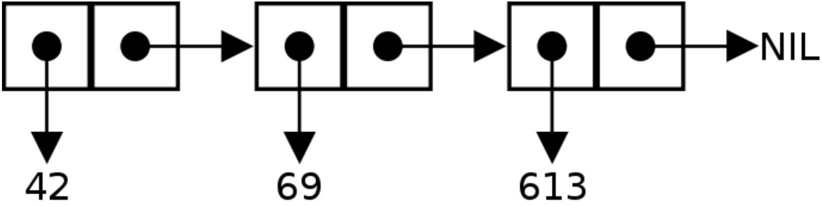
Lists in lisp is built using *linked lists*



An empty list is called `nil`

Linked lists

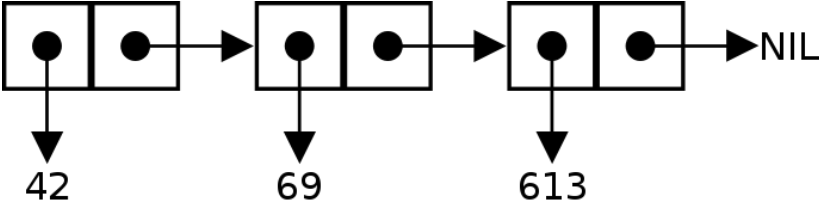
Lists in lisp is built using *linked lists*



An empty list is called `nil`

A cell is called a *cons*

Lists in lisp is built using *linked lists*



An empty list is called `nil`

A cell is called a *cons*

The two pointers is called `car` and `cdr`

Cons, car and cdr

A list can be constructed using `cons`: `(cons 4 nil)`

Cons, car and cdr

A list can be constructed using `cons`: `(cons 4 nil)`

What is `(car (cons 4 nil))`?

Cons, car and cdr

A list can be constructed using `cons`: `(cons 4 nil)`

What is `(car (cons 4 nil))`?

What is `(cdr (cons 4 nil))`?

append and reverse

Append appends a list on another

```
(append (list 1 2) (list 3 4))
```

append and reverse

Append appends a list on another

```
(append (list 1 2) (list 3 4))
```

Reverse a list with `nreverse`

```
(nreverse (list 1 2 3))
```

List exercises

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

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

Work on the `lists.lisp` file

Application as data

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

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

What can be computed?

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

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

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

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

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* $x^2 + y^2$.

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* $x^2 + y^2$.

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

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* $x^2 + y^2$.

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

$f(5)(2) =$

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* $x^2 + y^2$.

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

$f(5)(2) = 29$

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* $x^2 + y^2$.

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

$f(5)(2) = 29$

$f(5) =$

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* $x^2 + y^2$.

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

$f(5)(2) = 29$

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

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* $x^2 + y^2$.

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

$f(5)(2) = 29$

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

Boolean `T` and `nil`

Lisp syntax

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

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)))`

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)`

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)
```

Lambda exercise

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

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

Work on the `lambda.lisp` file