Lisp
Functional Programming

Jens Egholm Pedersen and Anders Kalhauge

Spring 2017

Outline

Lisp

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

Programming language

cphbusiness
COPENHAGEN BUSINESS ACADEMY

- 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)

Lisp

One of the first higher-level programming languages

Lisp

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`

Lisp

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 cphbusiness COPENHAGEN BUSINESS ACADEMY

One of the first higher-level programming languages

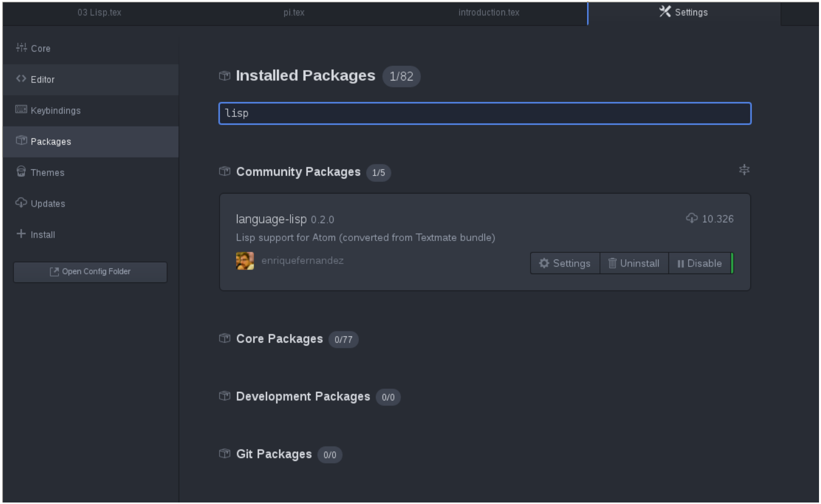
Pioneered many inventions: **tree structures**, **dynamic types**, **higher-order functions** and many more

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

Lisp syntax

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

- 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

- 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

Lisp syntax: Variables

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) )
```

cphbusiness
COPENHAGEN BUSINESS ACADEMY

Procedural programming

`(setf variable 10)` $\leftarrow$ mutable

Functional programming

Local variables: `let`-binding

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

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

Lisp exercises

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.

Lambda calculus

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!

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* $x^2 + y^2$.

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

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* $x^2 + y^2$.

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

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* $x^2 + y^2$.

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

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

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* $x^2 + y^2$.

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

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

$f(5)(2) = 29$

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* $x^2 + y^2$.

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

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

$f(5)(2) = 29$

$f(5) =??$

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* $x^2 + y^2$.

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

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

$f(5)(2) = 29$

$f(5) =??$

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

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* $x^2 + y^2$.

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

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

$f(5)(2) = 29$

$f(5) =??$

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

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)
```

Lambdas in Lisp

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

Lambdas in Lisp

```
(lambda () ())

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

```
(lambda () ())

(lambda (x) (* x x))

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

`if` statements

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)
```

`if` statements

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

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

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

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

Lists

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

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

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

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

Lists

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

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

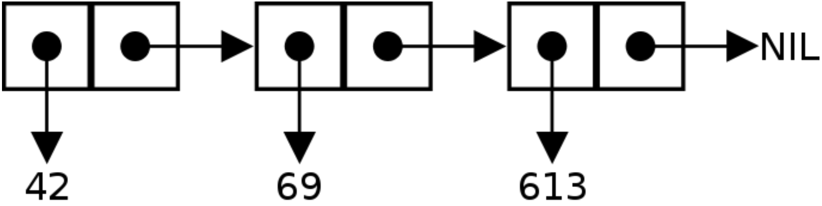
`(list 10 (list 5 2))`

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

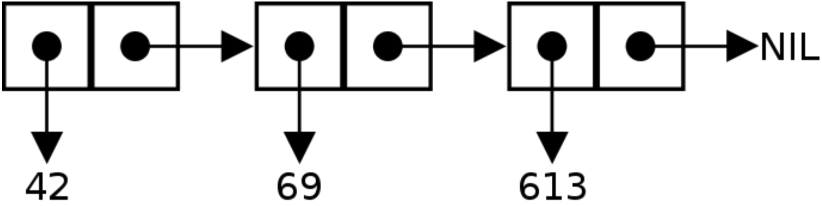
`(list 10 5 2)` $\mapsto$ `[10, 5, 2]`

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

Lists in lisp is built using *linked lists*



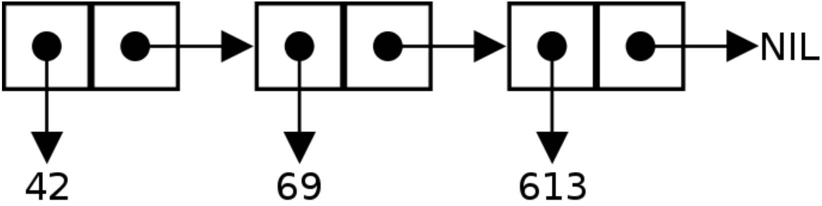
Lists in lisp is built using *linked lists*



An empty list is called `nil`

Linked lists

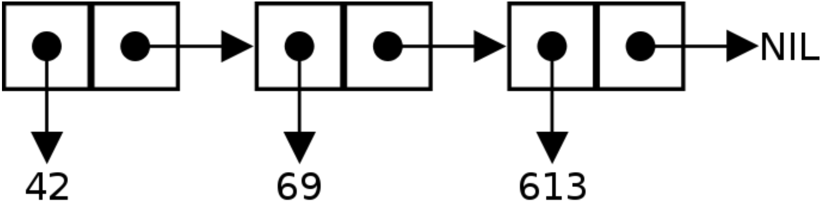
Lists in lisp is built using *linked lists*



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))
```

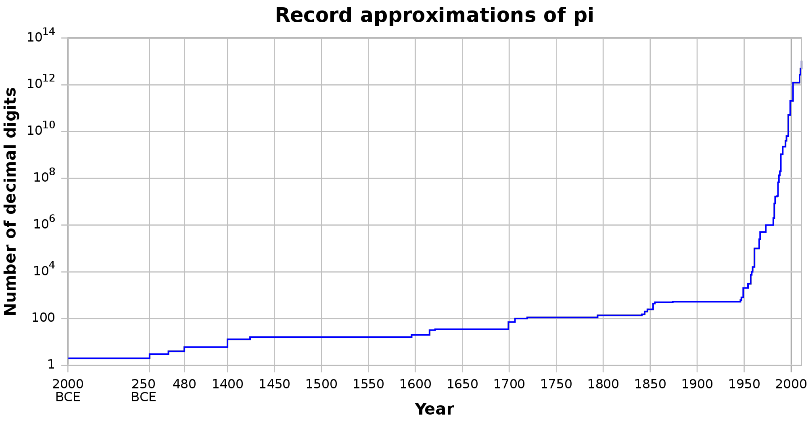
Reverse a list with `nreverse`

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

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

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

Work on the `lists.lisp` file


Record approximations of pi
Number of decimal digits
10^14
10^12
10^10
10^8
10^6
10^4
100
1
2000 BCE
250 BCE
480
1400
1450
1500
1550
1600
1650
1700
1750
1800
1850
1900
1950
2000
Year

Theory: Elliptic integrals

Theory: Elliptic integrals

$$\pi \approx \frac{(a_{n+1} + b_{n+1})}{4t_{n+1}} \tag{1}$$

Theory: Elliptic integrals

$$\pi \approx \frac{(a_{n+1} + b_{n+1})}{4t_{n+1}} \tag{1}$$

$$\begin{aligned} a_{n+1} = \tfrac{a_n + b_n}{2}, &\quad b_{n+1} = \sqrt{a_n b_n} \\ t_{n+1} = t_n - p_n(a_n - a_{n+1})^2, &\quad p_{n+1} = 2p_n \end{aligned} \tag{2}$$

Theory: Elliptic integrals

$$\pi \approx \frac{(a_{n+1} + b_{n+1})}{4t_{n+1}} \tag{1}$$

$$\begin{aligned} a_{n+1} &= \tfrac{a_n+b_n}{2}, \quad b_{n+1} = \sqrt{a_n b_n} \\ t_{n+1} &= t_n - p_n(a_n - a_{n+1})^2, \quad p_{n+1} = 2p_n \end{aligned} \tag{2}$$

Start values:

$$a_0 = 1 \qquad b_0 = \frac{1}{\sqrt{2}} \qquad t_0 = \frac{1}{4} \qquad p_0 = 1 \tag{3}$$