Introduction, Stacks and Lists

Functional Programming

Jens Egholm Pedersen and Anders Kalhauge

Spring 2018

Outline

Introduction
- Who we are
- Plan
- Books
- Tools
  - Install packages

What is a function

Data Structures
- Stacks
- Lists

Jens Egholm Pedersen

`jeep@cphbusiness.dk` (25 12 27 52)

- 4 years professionel experience in Java
- Studies IT and Cognition at KU
- Main interests
  - Programming and programming languages
  - Conscious automation
  - Critical systems

Anders Kalhauge

aka@cphbusiness.dk (21 72 44 11)

- 21 years experience as IT consultant in the private sector
- 16 years teaching computer science for students and private companies
- Main interests
  - Programming and programming languages
  - Development of large scale systems
  - Software architecture

We have decided on four major topics:

- Intro **1+ week** Our experience tells us, that not all of you are totally confident with the "intrails" of a computer on a low level. To understand functional programming and recursion, this knowledge is very helpful.
- LISP **2 weeks** LISP is one of the oldest functional languages but still going strong, so you will probably meet LISP in real life. LISP is great for an introduction to recursion.
- Elm **3+ weeks** Elm is a pure strongly typed functional language, designed to create SPA's on the net. It introduces many features of hardcore functional programming in an easy to understand responsive way.
- Haskell **5 weeks** Pure evil and awesomeness!

cphbusiness
COPENHAGEN BUSINESS ACADEMY

W	Part	Subject	Assignment
5	Intro	Stacks and Lists	**An RPN calculator**
LF		Recursion	**Tail rec. on virtual CPU**
8	LISP	Introduction	*TBD*
9		High-order functions	**Mapping and filtering**
10	Elm	The architecture	Install and run elm tools
11		Elm Commands	Elm client
12		Elm REST	**Elm REST**
LF		Elm Subscriptions	**Elm websocket client**
13	Easter		

Plan after Easter

W	Part	Subject	Assignment
13		Easter	
14	Haskell	Introduction	Install and run Haskell tools
15		IO and Monads	**Haskell Application**
16		Concurrency	Working with concurrency
17		Webserver	**Parallel web server**
18		Full-stack functional	**Full-stack chat**
19		*reserved*	
20	Recap	LISP/Elm/Haskell	

The plan is preliminary and is subject to change

Exam

The exam is oral. The student will prepare a (app. ten minutes) presentation of the solution of one of the major assignments. Further discussions will be based on the presentation, but can include all aspects of the curriculum.

In order to be approved for the exam:

- All four major assignments must be handed in
- At least 80 study points must be obtained

Study Points

- Hand in of major assignments (15 per assignment): 60
- Hand in of minor assignments (10 per assignment): 40

LISP Books

For the quick overview, sufficient for this course.

Download pdf or read online:
`http://www.tutorialspoint.com/lisp/`

For the details.

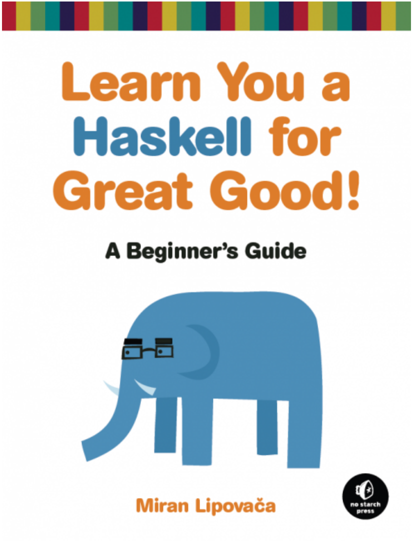
Read online: `http://www.gigamonkeys.com/book/`

Elm Book

An Introduction to elm

Download or read online:
`https://guide.elm-lang.org`

Haskell Book



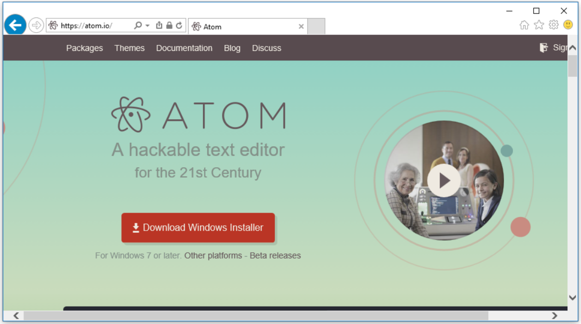
Buy or read online:
`http://learnyouahaskell.com`

We found only one editor (there might be more) that:

- Supported LISP, Elm, and Haskell
- Worked with Windows, Linux, and Mac

`https://atom.io`

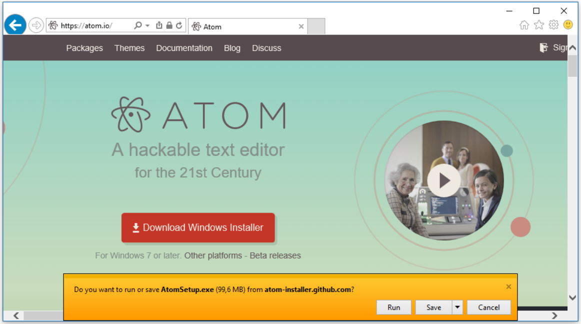
Windows I



Actions

Click "Download Windows Installer"

Windows II



Actions

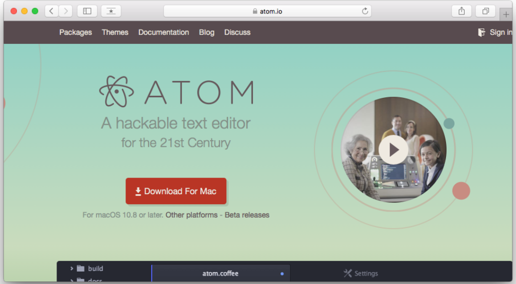
Click "Run"

Windows II

Actions

Wait for Atom to load

Mac I

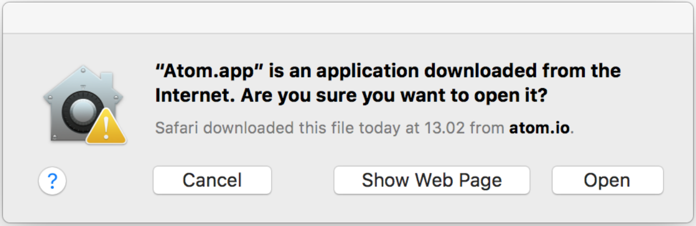


Actions

Click "Download for Mac"

Move `Atom.app` to the `Applications` folder

Mac II



Actions

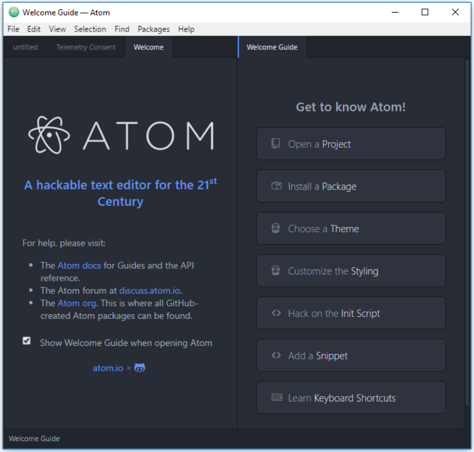
Open the `Applications` folder and double click `Atom.app`
Click "Open"

Linux

Actions

Haven't had access to Linux machine yet - sorry!
Follow the guide!?

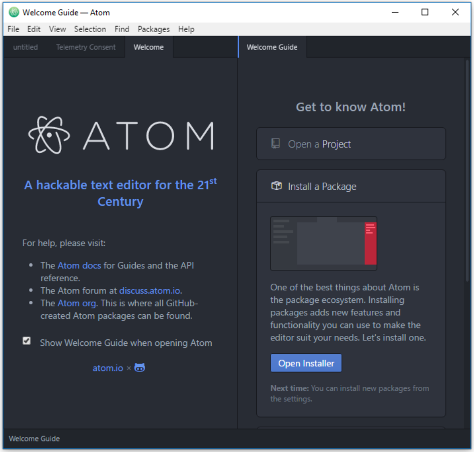
All platforms



Actions

Click “Install a package”

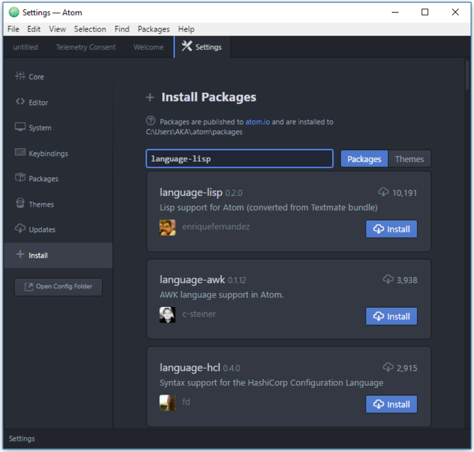
All platforms



Actions

Click "Open Installer"

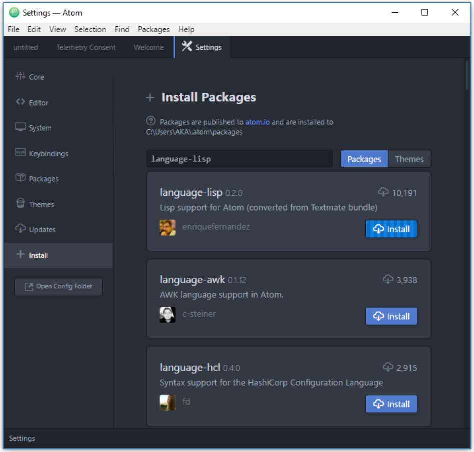
All platforms



Actions

Enter the package name
Press return to search

All platforms



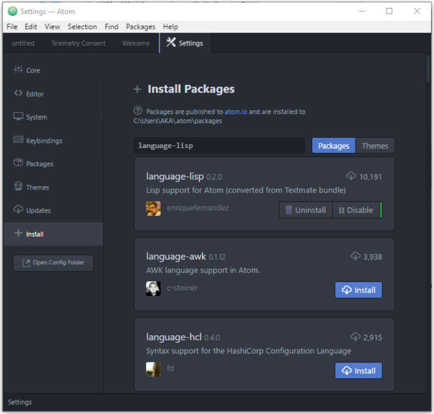
Actions

Click “Install”

Wait patiently

cphbusiness
COPENHAGEN BUSINESS ACADEMY

All platforms



Actions

Package installed

Repeat for

- ☐ `language-elm`
- ☐ `language-haskell`
- ☐ `autocomplete-haskell`
- ☐ `save-commands`
- ☐ `git-plus`

Function properties

Functions in math, and in functional programming have these properties:

- They have no side effects.
- Same input always give the same results
- They are never "`void`"

Function properties

Functions in math, and in functional programming have these properties:

- They have no side effects.
  - Can run in parallel with no raise conditions
- Same input always give the same results
- They are never “`void`”

Functions in math, and in functional programming have these properties:

- They have no side effects.
  - Can run in parallel with no raise conditions
- Same input always give the same results
  - They are very testable
- They are never "`void`"

Function properties

Functions in math, and in functional programming have these properties:

- They have no side effects.
  - Can run in parallel with no raise conditions
- Same input always give the same results
  - They are very testable
- They are never "`void`"
  - They can not stand alone as an application

Function properties

Functions in math, and in functional programming have these properties:

- They have no side effects.
  - Can run in parallel with no raise conditions
  - They can not stand alone as an application
- Same input always give the same results
  - They are very testable
  - They can not stand alone as an application
- They are never "`void`"
  - They can not stand alone as an application

As simple as it can be

$$f(x) = x^2$$

And with more than one argument

$$f(x) = x^2$$
$$g(x, y) = 2x + 4y^3$$

Even constants can be considered functions

$$c() = 7$$

$$f(x) = x^2$$

$$g(x, y) = 2x + 4y^3$$

cphbusiness
COPENHAGEN BUSINESS ACADEMY

In Java we could define the functions as:

```java
int c() { return 7; }
int f(int x) { return x*x; }
int g(int x, int y) { return 2*x + 4*y*y*y; }
```

Function types

In Java we could define the functions as:

```
int c() { return 7; }
int f(int x) { return x*x; }
int g(int x, int y) { return 2*x + 4*y*y*y; }
```

Note that types, in this case `int`, is only defined for the individual parameters and for the return values, not for the function itself!

Function types

In Java we could define the functions as:

```
int c() { return 7; }
int f(int x) { return x*x; }
int g(int x, int y) { return 2*x + 4*y*y*y; }
```

Note that types, in this case `int`, is only defined for the individual parameters and for the return values, not for the function itself!

In Java functions do **not** have types, functions are not instantiable in Java. We say that Java functions are **not** first class members as fields and classes are. Let's look at Kotlin . . .

In math, we would define the types as:

$$c : \mathbb{Z}$$

$$f : \mathbb{Z} \to \mathbb{Z}$$

$$g : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$

Where $\mathbb{Z}$[1] is the set of integer numbers

[1] German **Z**ahlen

Function types

In Kotlin types are written as above (and in UML) with a colon after the variable followed by the type:

```
fun c(): Int { return 7 }
fun f(x: Int): Int { return x*x }
fun g(x: Int, y: Int): Int { return 2*x + 4*y*y*y }
```

In Kotlin types are written as above (and in UML) with a colon after the variable followed by the type:

```
fun c(): Int { return 7 }
fun f(x: Int): Int { return x*x }
fun g(x: Int, y: Int): Int { return 2*x + 4*y*y*y }
```

Still only types on parameters and return values, but Kotlin also supports:

```
var c: () -> Int = { 5 }
var f: (Int) -> Int = { x -> x*x }
var g: (Int, Int) -> Int = { x, y -> 2*x + 4*y*y*y }
```

If functions have types, then functions can take other functions as parameters, and return functions.
What if the function g from above had the following type definition:

$$g : \mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z})$$

Then g would be a function taking one integer parameter, returning another function. The other function would also take one integer parameter but return an integer.

$$g(x)(y) = 2x + 4y^3$$

Then

$$h = g(3)$$
$$h(y) = 6 + 4y^3$$

cphbusiness
COPENHAGEN BUSINESS ACADEMY

In Elm functions are defined as:

```
c : Int
c = 7

f : Int -> Int
f x = x^2

g : Int -> Int -> Int
g x y = 2*x + 4*y^3
```

Immutable variables

In pure functional programming languages, variables can have a value only once.

- Eliminates side effects
- Eliminates iterations

- Introduction
  - Who we are
  - Plan
  - Books
  - Tools
    - Install packages
- What is a function
- Data Structures
  - Stacks
  - Lists

Stacks

Stacks are LIFO queues

A stack is a **L**ast **I**n **F**irst **O**ut queue.
Normally these methods are implemented:

`push(item: Item)` Push an item on the stack *"place a new plate on the top of the stack of plates"*

`pop(): Item` Pop an item from the stack *"remove the top plate from the stack"*

`peek(): Item` (Optional) Peek at the top item on the stack, without removing it. *"look at the top plate in the stack"*

`isEmpty(): Boolean` (Optional) Whether the stack is empty or not. *"are there any plates left?"*

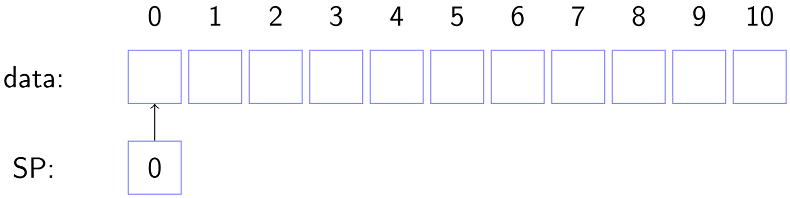
Stack in Java

Since 1.0

Java has the following definition in the `java.util` package:

```
public class Stack<E> extends Vector implements ... {
  public boolean empty() { ... }
  public E push(E item) { ... }
  public E pop() { ... }
  public E peek() { ... }
  public int search(Object o) { ... }
  }
```

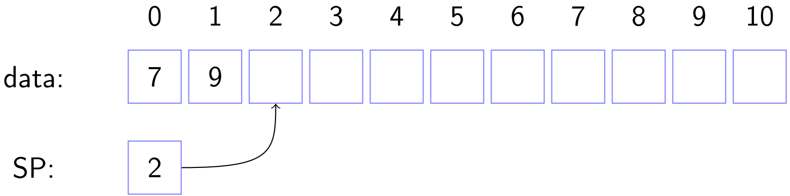
```
Stack stack = new Stack();
```



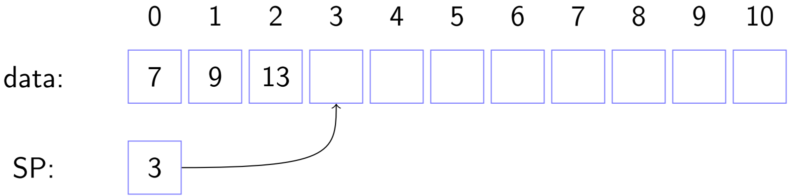
```
stack.push(7);
```

	0	1	2	3	4	5	6	7	8	9	10
data:	7										
SP:	1										

```
stack.push(9);
```



```
stack.push(13);
```



```
stack.pop(); // = 13
```

	0	1	2	3	4	5	6	7	8	9	10
data:	7	9	13								

SP: 2

Exercise 1 - Array Stack

Implement a stack, that uses an array as the internal data structure. The skelleton would look like:

```
public class ArrayStack<T> {
    private T[] data = null;
    private int top = 0;
    // ------------------------------
    public void push(T element) { ... }
    public T pop() { ... }
    public boolean isEmpty() { ... }
    }
```

You will need a constructor that initialises the data array to some size. If you have time, make `ArrayStack` implement `Iterable<T>`.

Recursion is all about divide and conquor. Split the problem into two or more similar but smaller problems, then combine the solutions of the minor problems.
A very important property of recursiveness is the existence of base cases. Problems that are so small, that they can be solved without further splitting.
Examples of recursive algorithms:

- Merge sort: Split the list to sort into two equally sized lists, and sort them. Merge the two sorted lists. The base case is a list with only one element.
- Factorial: Is the product of all number from 1 to a number n. Multiply n with the product of the numbers from 1 to $n-1$. The base case is 0, which factorial value is defined to 1.

Recursive algorithms on lists, often do something with the first element in the list and the rest of the list:

- Sum of all elements in a list: Add the value of the first element in the list to the sum of the rest of the list. The base case: The sum of an empty list is 0.
- Maximum of all elements in a list: Compare the value of the first element with the maximum of the rest. The base case: The maximum of a list with a single element is the value of that element.

Brute force

Most datastructures in procedural (and object-oriented) languages are designed for iterative processing.

```
public static int sum(int[] list) {
    if (list.length == 0) return 0;
    int first = list[0];
    int[] rest =
             Arrays.copyOfRange(list, 1, list.length);
    return first + sum(rest);
    }
```

The method `Arrays.copyOfRange(...)` is not very efficient, it traverses the array at every call, giving $O(n^2)$ for a simple sum.

cphbusiness
COPENHAGEN BUSINESS ACADEMY

Efficient, but not very elegant

Adding an index to the method signature removes the need to copy the list. Also, which is important, the list is "unharmed" after the call.

```
public static int max(int[] list) {
    return max(0, list);
    }

public static int max(int index, int[] list) {
    if (index == list.length - 1) return list[index];
    int first = list[index];
    int maxOfRest = max(index + 1, list);
    return first > maxOfRest ? first : maxOfRest;
    }
```

Sum and maximum in code

Elegant and efficient

Consider a special type of integer list IntPath:

```
public static int sum(IntPath list) {
    if (list == null) return 0; // empty path = null
    return list.getFirst() + sum(list.getRest());
    }

public static int max(IntPath list) {
    int first = list.getFirst();
    if (list.getRest() == null) return first;
    int maxOfRest = max(list.getRest());
    return first > maxOfRest ? first : maxOfRest;
    }
```

Implement in Java (or Kotlin) a special kind of list here called a Path with the following interface.

```
public interface Path<T> {
    T getFirst();
    Path<T> getRest();
    }
```

Be aware of the fact that the data structure neither is a List (which is indexable) nor Collection (which is countable) in the Java sense.
If you have spare time, you can convince yourself by implementing a NumberPath that has all numbers from 0 to Long.MAX_VALUE.

```
public class NumberPath implements Path<Long> {
    Long getFirst() { ... }
    Path<Long> getRest() { ... }
    }
```

Sum in Kotlin

Maybe one should try that one day?

```
sealed class Path<T> {
    class Steps<T>(
        val first: T,
        val rest: Path<T> = Empty()
        ) : Path<T>()
    class Empty<T> : Path<T>()
    }

fun sum(list: Path<Int>) : Int =
    when (list) {
        is Path.Empty -> 0
        is Path.Steps -> list.first + sum(list.rest)
        }
```

Exercise 3 - Path Stack

Implement a stack, that uses the `Path` from Exercise 2 as the internal data structure. The skeleton would look like:

```
public class PathStack<T> {
    private Path<T> data = null;
    // ------------------------------
    public void push(T element) { ... }
    public T pop() { ... }
    public boolean isEmpty() { ... }
    }
```

You will need this class for the first Assignment.