

# Introduction and Stacks

## Functional Programming

Jens Egholm Pedersen and Anders Kalhauge



Spring 2017

## Introduction

- Who we are

- Plan

- Books

- Tools

  - Windows

  - Mac

  - Linux

  - Install packages

## Data Structures

- Stacks

- Lists

## Introduction

Who we are

Plan

Books

Tools

Windows

Mac

Linux

Install packages

## Data Structures

Stacks

Lists

# Jens Egholm Pedersen

jeep@cphbusiness.dk (25 12 27 52)

- 4 years professional 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 2 + 1 weeks** 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, and probably one you will meet in real life. LISP is great for an introduction to recursion.

**Elm 6 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 4 weeks** Pure evil and awesomeness!

W	Part	Subject	Assignment
5	Intro	Stacks	An RPN calculator
6		Recursion	A stack CPU
7	LISP	Lists and recursion	$\pi$ with 1000 decimals
8		High order functions	<b>Mapping and filtering</b>
9	Elm	The language	Install elm tools
10		Simple architecture	HTML page with tests
11		Commands	HTML page with
12			- RESTful
13		Subscriptions	<b>HTML page with</b>
14			<b>- RESTful &amp; WebSockets</b>
15	Easter		

W	Part	Subject	Assignment
15	Easter		
16	Theory	Tail recursiveness	<b>CPU Simulator for TR</b>
17	Haskell	Introduction	Web tutorial
18		Types	Bezier with Graphics - Gloss
19		IO and frameworks	<b>Simple web server</b>
20		Monoids and Monads	
21	Recap	LISP/Elm/Haskell	

The plan is preliminary and is subject to change



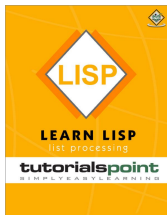
The exam is oral. The student will prepare a (app. ten minutes) presentation of the solution of one of the major assignments. Further discussions will be based on the presentation, but can include all aspects of the curriculum.

In order to be approved for the exam:

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

- Attendance: 20
- Hand in of major assignments (10 per assignment): 40
- Hand in of minor assignments (5 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/>

## An Introduction to



# elm

Download or read online:

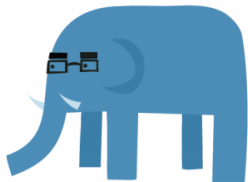
<https://guide.elm-lang.org>

## Haskell Book



# Learn You a Haskell for Great Good!

**A Beginner's Guide**



**Miran Lipovača**



Buy or read online:

<http://learnyouahaskell.com>

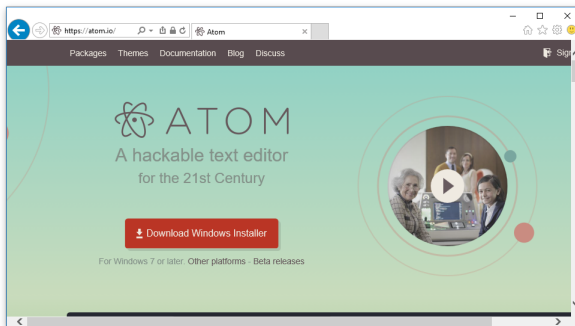
We found only one editor (there might be more) that:

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



<https://atom.io>

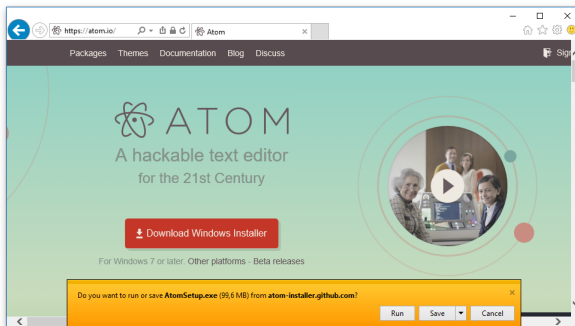
## Windows I



## Actions

Click “Download Windows Installer”

## Windows II



## Actions

Click "Run"



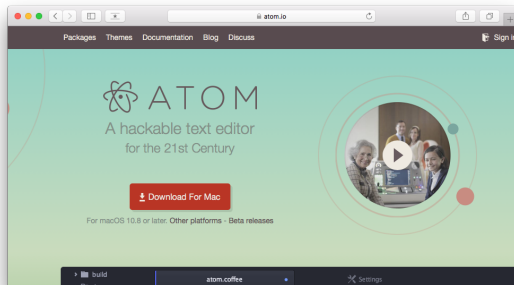
## Windows II



## Actions

Wait for Atom to load

## Mac I

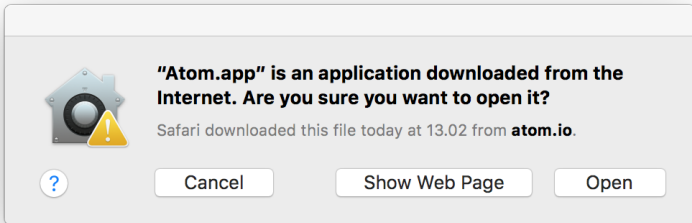


## Actions

Click “Download for Mac”

Move Atom.app to the Applications folder

## Mac II



## Actions

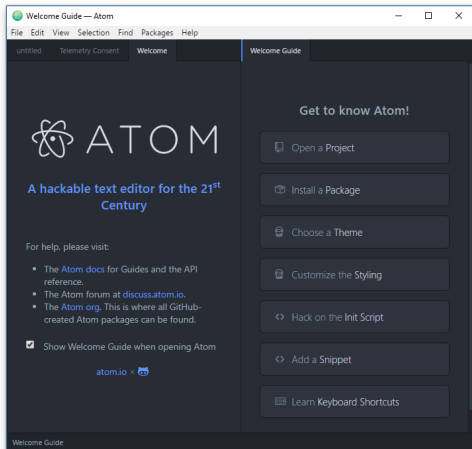
Open the Applications folder and double click Atom.app  
Click "Open"

Linux

## Actions

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

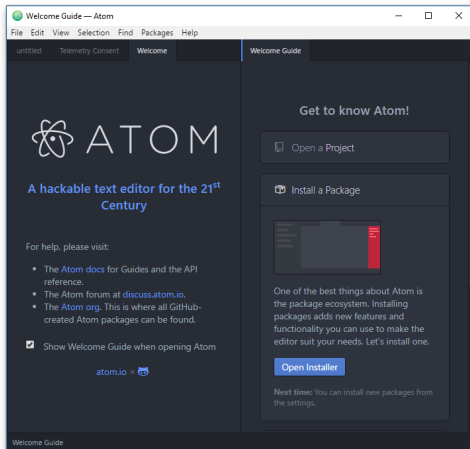
All platforms



## Actions

Click “Install a package”

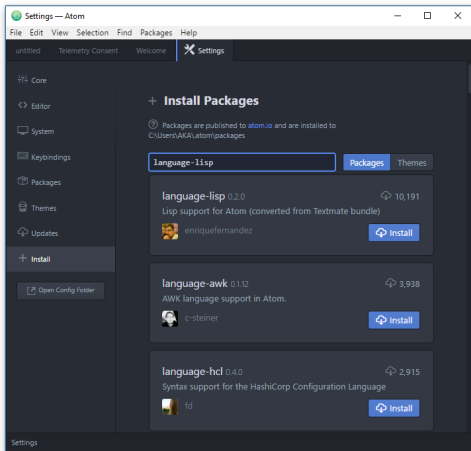
All platforms



## Actions

Click “Open Installer”

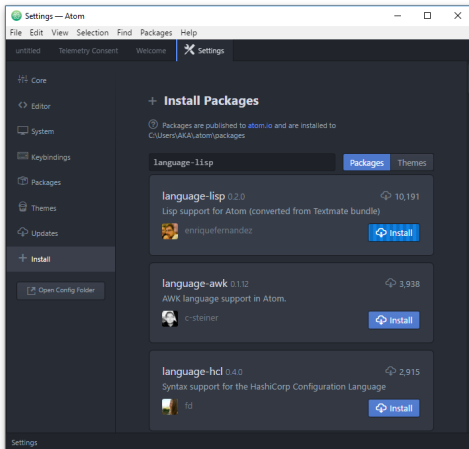
All platforms



## Actions

Enter the package name  
Press return to search

All platforms



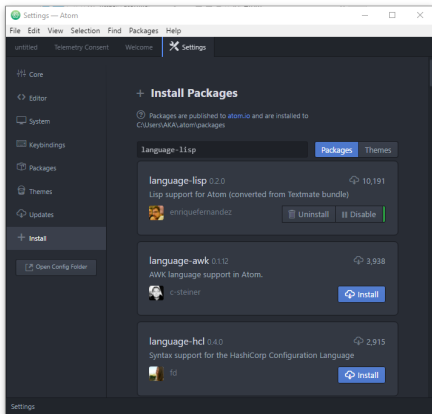
## Actions

Click “Install”

Wait patiently








All platforms



## Actions

Package installed

Repeat for

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

## Introduction

Who we are

Plan

Books

Tools

Windows

Mac

Linux

Install packages

## Data Structures

Stacks

Lists



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”*

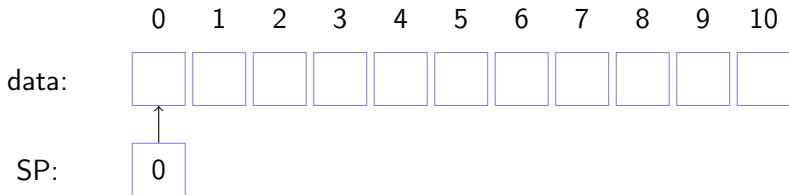
`isEmpty(): Boolean` (Optional) Whether the stack is empty or not. *“are there any plates left?”*

Since 1.0

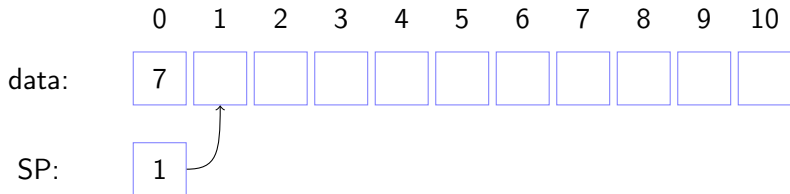
Java has the following definition in the `java.util` package:

```
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) { ... }  
}
```

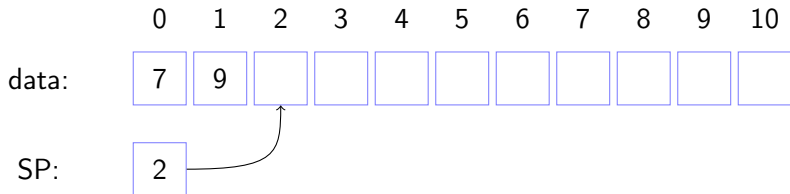
```
Stack stack = new Stack();
```



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

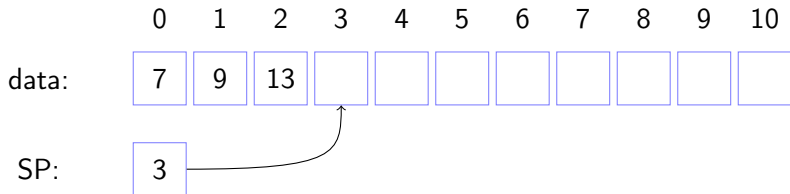


```
stack.push(9);
```

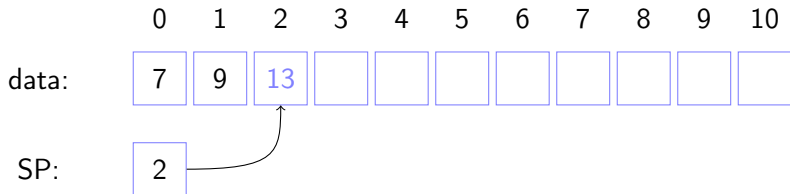




```
stack.push(13);
```



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



Implement a stack, that uses an array as the internal data structure. The skeleton 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 conquer. 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.

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 “unharmd” 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;  
}
```

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() { ... }  
}
```

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)  
    }
```

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.