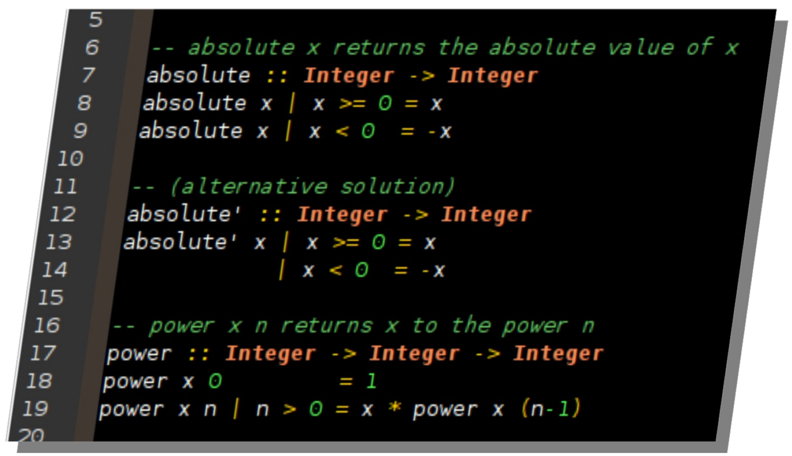
Introduction to Functional Programming



Slides by Koen Claessen and Emil Axelsson

Programming

- Exciting subject at the heart of computing
- Never programmed?
  - Learn to make the computer obey you!
- Programmed before?
  - Lucky you! Your knowledge will help a lot...
  - ...as you learn a completely new way to program
- *Everyone* will learn a great deal from this course!

Goal of the Course

- Start from the basics
- Learn to write small-to-medium sized programs in *Haskell*
- Introduce basic concepts of computer science

The Flow

Do not *break the flow*!

You prepare *in advance*

I explain *in lecture*

Tuesdays,Fridays

You learn *with exercises*

Mondays

You put to practice with *lab assignments*

Submit end of each week

Exercise Sessions

- Mondays
  - Group rooms
- Come prepared
- Work on exercises together
- Discuss and get help from tutor
  - Personal help
- Make sure you understand this week’s things before you leave

Lab Assignments

- General information

  http://www.cse.chalmers.se/edu/course/TDA555/labs.html

- Start working on lab when you have understood the matter
- Submit end of each week

even this week!

Getting Help

- Weekly group sessions
  - Personal help to understand material
- Lab supervision
  - Specific questions about programming assignment at hand
- Discussion forum
  - General questions, worries, discussions
  - *Finding lab partners*

Assessment

- Written exam (4.5 credits)
  - Consists of small programming problems to solve on paper
  - You need Haskell “in your fingers”
- Course work (3 credits)
  - Complete all labs successfully

A Risk

- 8 weeks is a short time to learn programming
- So the course is fast paced
  - Each week we learn a lot
  - Catching up again is hard
- So do keep up!
  - Read the material for each week
  - Make sure you can solve the problems
  - Go to the weekly exercise sessions
  - *From the beginning*

Course Homepage

The course homepage will have ALL up-to-date information relevant for the course

- Schedule and slides
- Lab assignments
- Exercises
- Last-minute changes
- (etc.)

Or go via the student portal

http://www.cse.chalmers.se/edu/course/TDA555/

Software

Software = Programs + Data

Software = Programs + Data

- Data is any kind of storable information, e.g:
  - numbers, letters, email messages
  - maps, video clips
  - mouse clicks, *programs*
- Programs compute new data from old data:
  - A computer game computes a sequence of screen images from a sequence of mouse clicks
  - vasttrafik.se computes an optimal route given a source and destination bus stop

Building Software Systems

- A large system may contain many *millions* of lines of code
- Software systems are among the most complex artefacts ever made by humans
- Systems are built by combining existing components as far as possible.

Volvo buys engines from Mitsubishi. ⟷ Facebook buys video player from Adobe

Programming Languages

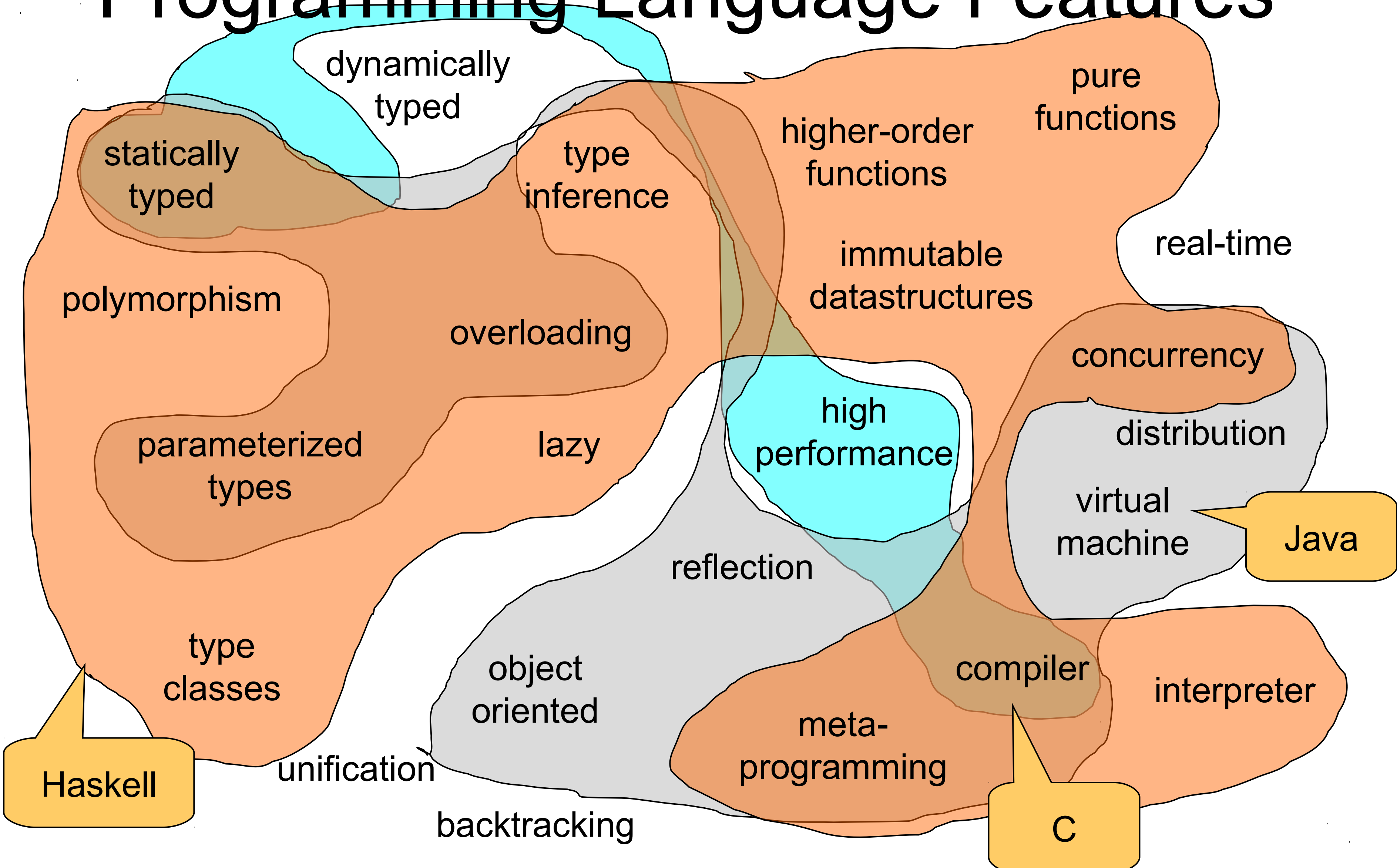
- Programs are written in *programming languages*
- There are hundreds of different programming languages, each with their strengths and weaknesses
- A large system will often contain components in many different languages

Programming Languages

which language should we teach?

Lisp
Scheme
C
BASIC
Haskell
Java
C++
C#
ML
Python
JavaScript
O'CaML
Curry
csh
Perl
bash
Erlang
Prolog
Ruby
Lustre
Mercury
PostScript
VHDL
Esterel
SQL
PDF
Verilog

Programming Language Features



Teaching Programming

- Give you a broad basis
  - Easy to learn more programming languages
  - Easy to adapt to new programming languages
    - Haskell is defining state-of-the-art in programming language development
  - Appreciate differences between languages
  - Become a better programmer!

“Functional Programming”

- **Functions** are the basic building blocks of programs
- **Functions** are used to compose these building blocks into larger programs

- A (pure) **function** computes results from arguments – *consistently the same*

Industrial Uses of Functional Languages

Intel (microprocessor verification)

Hewlett Packard (telecom event correlation)

Ericsson (telecommunications)

Jeppesen (air-crew scheduling)

Facebook (chat engine)

Credit Suisse (finance)

Barclays Capital (finance)

Hafnium (automatic transformation tools)

Shop.com (e-commerce)

Motorola (test generation)

Thompson (radar tracking)

Microsoft (F#)

Jasper (hardware verification)

And many more!

Microsoft chockar programmerarna

Med funktionella språk måste utvecklarna tänka om

När all världens programmerare fått koll på objektorientering är det dags för nästa paradigmskifte. Med Microsoft som härförare vinner funktionella språk mark. Programmerarna får räkna med att lära om.

SPRÅKFUNKIS. Dags att lära sig ett nytt språk, ett funktionellt den här gången.

Computer Sweden, **2010**

Why Haskell?

- Haskell is a very *high-level language* (many details taken care of automatically).
- Haskell is expressive and concise (can achieve a lot with a little effort).
- Haskell is good at handling complex data and combining components.
- Haskell is *not* a particularly high-performance language (prioritise programmer-time over computer-time).

Cases and recursion

Example: The squaring function

- Example: a function to compute x^2

```
-- sq x returns the square of x
sq :: Integer -> Integer
sq x = x * x
```

Evaluating Functions

- To evaluate sq 5:
  - *Use the definition*—substitute 5 for x throughout
    - sq 5 = 5 * 5
  - Continue evaluating expressions
    - sq 5 = 25
- Just like working out mathematics on paper

```
sq x = x * x
```

Example: Absolute Value

- Find the absolute value of a number

```
-- absolute x returns the absolute value of x
absolute :: Integer -> Integer
absolute x = undefined
```

Example: Absolute Value

- Find the absolute value of a number
- Two cases!
  - If x is positive, result is x
  - If x is negative, result is -x

Programs must often choose between alternatives

```
-- absolute x returns the absolute value of x
absolute :: Integer -> Integer
absolute x | x > 0 = undefined
absolute x | x < 0 = undefined
```

Think of the cases! These are *guards*

Example: Absolute Value

- Find the absolute value of a number
- Two cases!
  - If x is positive, result is x
  - If x is negative, result is -x

```
-- absolute x returns the absolute value of x
absolute :: Integer -> Integer
absolute x | x > 0 = x
absolute x | x < 0 = -x
```

Fill in the result in each case

Example: Absolute Value

- Find the absolute value of a number
- Correct the code

```
-- absolute x returns the absolute value of x
absolute :: Integer -> Integer
absolute x | x >= 0 = x
absolute x | x < 0   = -x
```

> \>= is *greater than or equal*, $\geq$

Evaluating Guards

- Evaluate absolute (-5)
  - We have two equations to use!
  - Substitute
    - absolute (-5) | -5 >= 0 = -5
    - absolute (-5) | -5 < 0 = -(-5)

```
absolute x | x >= 0 = x
absolute x | x < 0    = -x
```

Evaluating Guards

- Evaluate absolute (-5)
  - We have two equations to use!
  - Evaluate the guards
    - absolute (-5) | False = -5
    - absolute (-5) | True = -(-5)

Discard this equation

Keep this one

```
absolute x | x >= 0 = x
absolute x | x < 0    = -x
```

Evaluating Guards

- Evaluate absolute (-5)
  - We have two equations to use!
  - Erase the True guard
    - absolute (-5) = -(-5)

```
absolute x | x >= 0 = x
absolute x | x < 0    = -x
```

Evaluating Guards

- Evaluate absolute (-5)
  - We have two equations to use!
  - Compute the result
    - absolute (-5) = 5

```
absolute x | x >= 0 = x
absolute x | x < 0   = -x
```

Notation

- We can abbreviate repeated left hand sides

```
absolute x | x >= 0 = x
absolute x | x < 0   = -x
```

```
absolute x | x >= 0 = x
           | x < 0   = -x
```

- Haskell also has if then else

absolute x = **if** x >= 0 **then** x **else** -x

Example: Computing Powers

- Compute x^n (without using built-in x^n)

Example: Computing Powers

- Compute x^n (without using built-in x^n)
- Name the function

power

Example: Computing Powers

- Compute x^n (without using built-in x^n)
- Name the inputs

```
power x n = undefined
```

Example: Computing Powers

- Compute x^n (without using built-in x^n)
- Write a comment

```
-- power x n returns x to the power n
power x n = undefined
```

Example: Computing Powers

- Compute x^n (without using built-in x^n)
- Write a type signature

```
-- power x n returns x to the power n
power :: Integer -> Integer -> Integer
power x n = undefined
```

How to Compute power?

- We cannot write
  - power x n = x * … * x

n times

A Table of Powers

n	power x n
0	1
1	x
2	x*x
3	x*x*x

$$x^n = x \cdot x^{(n-1)}$$

- Each row is x* the previous one
- Define (power x n) to compute the nth row

A Definition?

```
power x n = x * power x (n-1)
```

- Testing:
  Main> power 2 2
  ERROR - stack overflow

Why?

A Definition?

```
power x n | n > 0 = x * power x (n-1)
```

- Testing:
  Main> power 2 2
  Program error: pattern match failure: power 2 0

A Definition?

```
power x 0 = 1
power x n | n > 0 = x * power x (n-1)
```

First row of the table

- Testing:
  Main> power 2 2
  4

The **BASE CASE**

Recursion

- First example of a *recursive* function
  - Defined in terms of itself!

```
power x 0 = 1
power x n | n > 0 = x * power x (n-1)
```

- Why does it work? Calculate:
  - power 2 2 = 2 * power 2 1
  - power 2 1 = 2 * power 2 0
  - power 2 0 = 1

Recursion

- First example of a *recursive* function
  - Defined in terms of itself!

```
power x 0 = 1
power x n | n > 0 = x * power x (n-1)
```

- Why does it work? Calculate:
  - power 2 2 = 2 * power 2 1
  - power 2 1 = 2 * 1
  - power 2 0 = 1

Recursion

- First example of a *recursive* function
  - Defined in terms of itself!

```
power x 0 = 1
power x n | n > 0 = x * power x (n-1)
```

- Why does it work? Calculate:
  - power 2 2 = 2 * 2
  - power 2 1 = 2 * 1
  - power 2 0 = 1

No circularity!

Recursion

- First example of a *recursive* function
  - Defined in terms of itself!

```
power x 0 = 1
power x n | n > 0 = x * power x (n-1)
```

- Why does it work? Calculate:
  - power 2 2 = 2 * power 2 1
  - power 2 1 = 2 * power 2 0
  - power 2 0 = 1

The **STACK**

Recursion

- Reduce a problem (e.g. power x n) to a *smaller* problem of the same kind jag
- So that we eventually reach a “smallest” *base case*
- Solve base case separately
- Build up solutions from smaller solutions

Powerful problem solving strategy
in *any* programming language!

Example: Replication

- Implement a function that replicates a given word n times

```
repli :: Integer -> String -> String
repli ...
```

```
GHCi> repli 3 “apa”
“apaapaapa”
```

An Answer

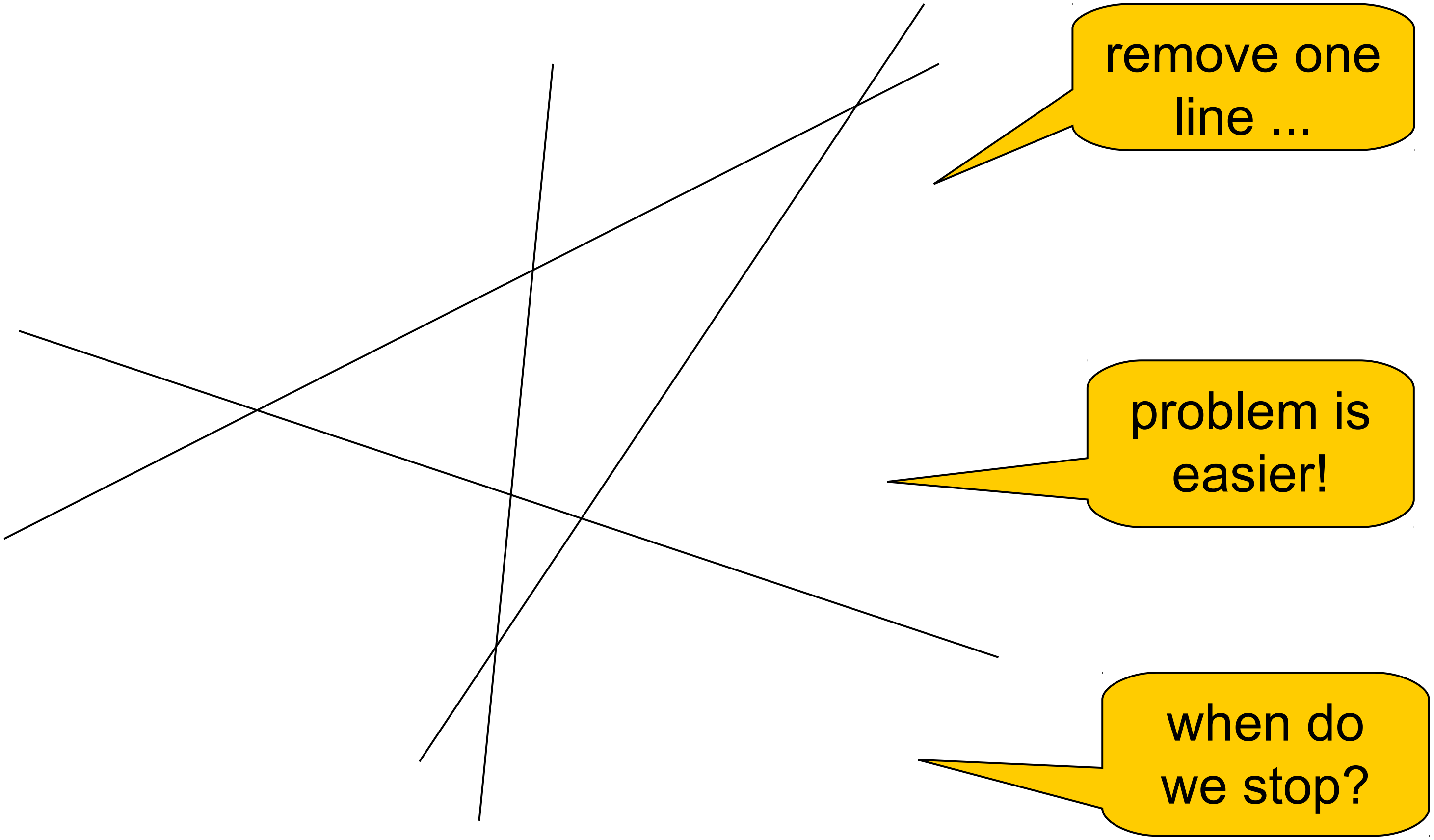
```
repli :: Integer -> String -> String
repli 1 s             = s
repli n s | n > 1 = s ++ repli (n-1) s
```

```
repli :: Integer -> String -> String
repli 0 s             = “”
repli n s | n > 0 = s ++ repli (n-1) s
```

make base case as simple as possible!

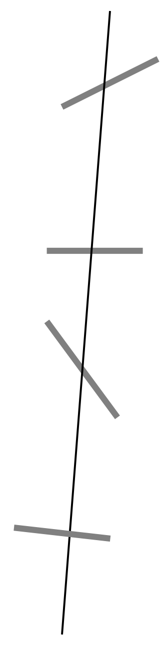
Counting the regions

- n lines. How many regions?



Counting the regions

- The nth line creates n new regions



A Solution

- Don't forget a base case

```
regions :: Integer -> Integer
regions 1             = 2
regions n | n > 1 = regions (n-1) + n
```

A Better Solution

- Always make the base case as simple as possible!

```
regions :: Integer -> Integer
regions 0             = 1
regions n | n > 0 = regions (n-1) + n
```

Group

- Divide up a string into groups of length n

```
group :: ...
group n s = ...
```

Types

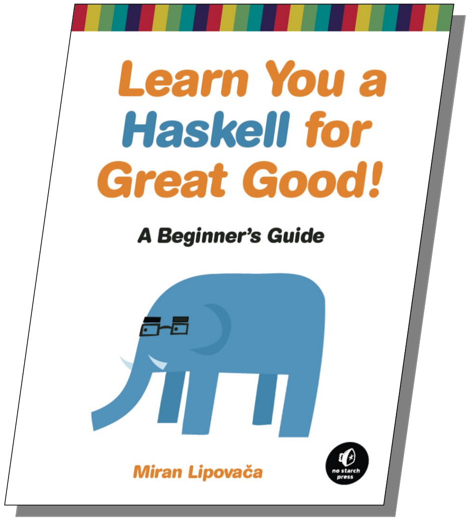
- What are the types of repli and group?

```
repli :: Integer -> String -> String
group :: Integer -> String -> [String]
```

```
repli :: Integer -> [a] -> [a]
group :: Integer -> [a] -> [[a]]
```

Material

- Book (online):
  http://learnyouahaskell.com/



- Lecture slides

- Overview for each lecture:
  http://www.cse.chalmers.se/edu/course/TDA555/lectures.html