Advanced Programming

Introduction to Haskell

Ken Friis Larsen
kflarsen@diku.dk

Department of Computer Science
University of Copenhagen

September 4, 2012

Today's Menu

- General course information
- Course content and motivation
- Introduction to Haskell

Learning Objectives

After taking this course the student should be able to:

- Use programming structuring principles and design patterns, such as monads, to structure the code so that there is a clear separation of concerns.
- Use a parser combinator library to write a parser for medium-sized language with a given grammar, including changing the grammar so that it is on an appropriate form.
- Use parallel algorithm skeletons such as map-reduce to write data exploring programs.
- Implement simple concurrent/distributed servers using message passing, with appropriate use of synchronous and asynchronous message passing.
- Use programming structuring principles and design patterns for making reliable distributed systems in the presence of software errors.
- Write idiomatic programs in a logic programming language.
- Give an assessment based on a systematic evaluation of correctness, selection of algorithms and data structures, error scenarios, and elegance.

Course Goals, Rephrased

- Learn about advanced programming techniques for realistic, useful program designs.
- Practice using these techniques in realistic code.
- Practise to read a research paper.
  - Bring concepts and ideas from one language/paradigm to another.

Teachers

Ken Friis Larsen
Haskell, Monads

Michael Kirkedal Carøe
Prolog, Erlang

Marcho Markov

Oleksandr Shturmov

Simon Shine

Troels Henriksen

Online Information

- The course home page can be found in Absalon
- The home page for the course contains a detailed lecture plan, exercises, latest news, and other important course information.
- The lecture plan contains links to slides
- **Keep an eye** on the course home page throughout the block.
- Lectures Tuesday 10:15–12:00 and Thursday 13:15–15:00
- TA's clinic: Thursday 15:15-17:00 in rooms 3-1-25, 1-0-22, 1-0-04, 1-0-18.

How Should You Spend Your Time

- A typical week:

Attend lectures:	4 hours
Read articles:	6 hours
Coding and write up solutions:	10 hours

- We will try to provide open-ended exercises as inspiration for how to work with the topics.
- If you spend significantly less or more time on the course, please let us know.

To Pass The Course

- Pass 2 out of 3 mandatory assignments (we recommend that you pass them all). Groups are allowed, and recommended.
  - Maximum group size is two members
- Pass a one week take-home exam (typically consisting of 3-4 questions, each roughly the size of an assignment).

Languages In This Course

- Haskell
  - `http://haskell.org`
  - Haskell Platform (`http://hackage.haskell.org/platform/`) with GHC (`http://haskell.org/ghc`)
- Erlang
  - `http://erlang.org`
- Prolog
  - SWI-Prolog (`http://www.swi-prolog.org/`)
  - GNU-Prolog (`http://www.gprolog.org/`)

What You'll Build

- An environment for maze solving robots
- Programs only described by rules. That is, without explicit control-flow
- Algorithms that can be massively distributed

Haskell

Haskell

- is a lazy, pure, statically typed functional programming language
- is often used as a vehicle for programming language research

How do we learn a new programming language?

Haskell Basics

- A Haskell value:

```
[("Homer", 42), ("Bart", 8)]
```

- It has type:

```
[(String, Int)]
```

- We can declare a name for it:

```
maleSimpsons :: [(String, Int)]  -- type signature
maleSimpsons = [("Homer", 42), ("Bart", 8)]
```

- A functional value:

```
\ x y -> x+y
```

- We can declare a name for it:

```
add :: Num n => n -> n -> n
add = \ x y -> x+y
add' x y = x+y
```


More Haskell Fun

- Haskell has list comprehensions:

```
digits = [0..9]
evenDigits = [x | x <- digits, x `mod` 2 == 0]
```

- Even infinite lists:

```
nats = [0 ..]
evenNats = [x | x <- nats, x `mod` 2 == 0]
```

- Functions that works on lists:

```
startFrom s = s : startFrom (s+1)
len [] = 0
len (_ : t) = 1 + len t
```

- An old friend:

```
q [] = []
q (x:xs) = q sxs ++ [x] ++ q lxs
    where sxs = [a | a <- xs, a <= x]
          lxs = [b | b <- xs, b > x]
```


Working With Types

- We can declare type aliases:

```
type Pos = (Int, Int)
```

- Record types:

```
data Student = Student { name :: String
                       , knowsHaskell :: Bool}
```

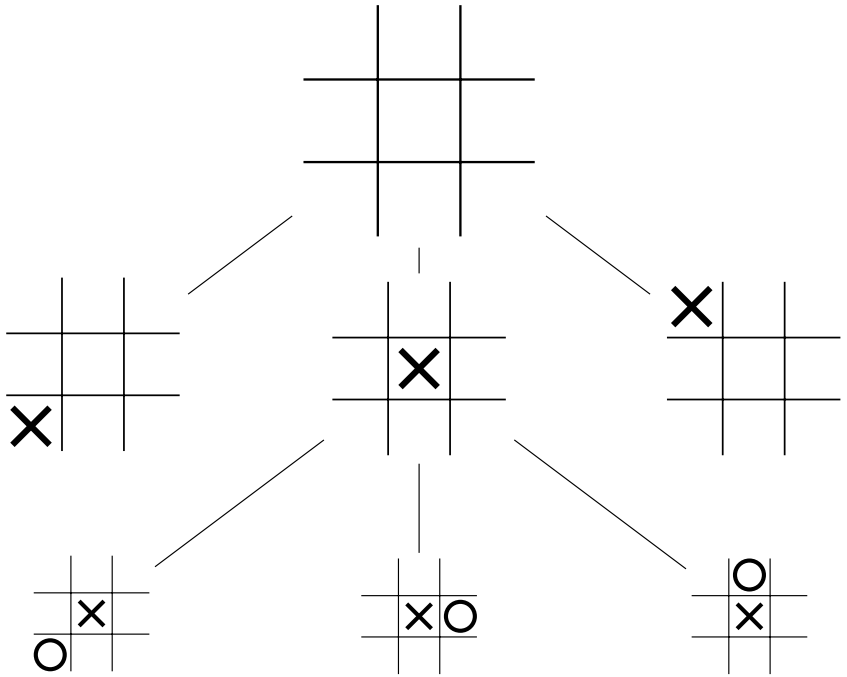
- Algebraic data types:

```
data Direction = North | South | East | West
```

- Functions on all of these

```
followAP :: Student -> Student
followAP stud = stud{knowsHaskell = True}

move :: Direction -> Pos -> Pos
move North (x,y) = (x, y+1)
move West  (x,y) = (x-1, y)
```

Polymorphic Types

- Some polymorphic types:

```
type Assoc a = [(String, a)]
{- The following two types are part of the prelude -}
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
```

- A useful function:

```
findAssoc :: String -> Assoc a -> a
findAssoc key assoc = head bindings
   where bindings = [val| (k,val) <- assoc, k == key]
```


Recursive Types

- A data type for modelling natural numbers

```
data Nat = Zero | Succ Nat
     deriving (Eq, Show, Read, Ord)
```

- A function for adding natural numbers:

```
add x Zero = x
add x (Succ n) = add (Succ x) n
```

- We can declare our own list type, if we want:

```
data List a = Nil | Cons a (List a)
```


Type Classes

- Haskell use *type classes* for managing ad-hoc overloading.
- For example, the `Eq` class from the prelude:

```
Class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not(x == y)
```

- We could have declared `Nat` to be an instance of `Eq`, instead of using `deriving`:

```
instance Eq Nat where
  Zero == Zero     = True
  Succ n == Succ m = n == m
  _ == _           = False
```

- Type classes in Haskell are similar to interfaces in Java

Abstract Syntax Trees

- Algebraic data types are excellent for modelling abstract syntax trees.
- For instance for arithmetic expressions:

```
data Expr = Con Int
          | Add Expr Expr
     deriving (Eq, Show, Read, Ord)

value :: Expr -> Int
value (Con n) = n
value (Add x y) = value x + value y
```

Tasks For The Week

- Install Haskell on your computer
- Talk to your fellow students about forming a group (max two members)
- Work on exercise set 1.
- Ken's email: `kflarsen@diku.dk`