# Lecture 11: Functional programming

## Haskell

December 20, 2017

# What is Haskell?

An advanced language for lazy functional programming

- named after logician Haskell Curry
- standardized in 1990 (Haskell 1), 1998, 2010
- freely available from `https://www.haskell.org`
  $\Rightarrow$ download the Glasgow Haskell Compiler (GHC) which has two main components
  - a batch compiler
  - GHCi: an interactive interpreter

  \+ a large number of libraries

- Purely functional: functions in Haskell are like mathematical functions: output depends only on input
  - there are no statements or instructions
  - there are only expressions which can not mutate variables
- Statically typed: Every expression has a type which is determined at compile time
  - all Haskell values have a type
  - types composed together by function application must match up, otherwise the compiler/interpreter will complain
- There is a built-in type inference system which can compute the types omitted by the programmer $\Rightarrow$ you don't have to write out every type.
- Lazy: functions evaluate their arguments only as much as it is needed $\Rightarrow$ control constructs like `if/else` can be defined as lazy functions

- Concurrent: GHC has
  - a built-in high-performance parallel garbage-collector
  - A library with useful primitives and abstractions for concurrency
- A wide range of open-source packages from a very active community.

- Direct interaction with GHCi in the browser, at website `https://www.haskell.org`
- Install it on your own computer (recommended):
  - download and run the installer for your platform (Windows or Linux or OS X) ; it is recommended to get the installer for the Haskell Platform
  - start the Haskell interpeter with the command `ghci`:

```
> ghci
GHCi, version 8.2.2: http://www.haskell.org/ghc/ :? for help
Prelude>
```

NOTES:

- GHCi loads the prelude and standard libraries, after which the prompt is shown. Some useful commands:
  `:set +t` turns on GHCi to show the type of each variable bound by a statement
  `:load` *sourcefile* loads the code written in file *sourcefile*
  `:quit` exits GHCi
  `:?` lists the commands available

# Interacting with the Haskell interpreter

There are 2 ways to interact with the interpreter:

1. Load the content of a Haskell source file *srcfile* with command

   `:load` *srcfile* or just `:l` *srcfile*

   - *srcfile* can contain type declarations, and definitions of: type classes (`class`), types (`data`, `instance`), functions, etc.
   - In a source file, expressions can be written on many lines

2. Type an expression on a single line after the input prompt, and press ENTER
   $\Rightarrow$ the interpreter will print the value (and type) of the result.
   - If we want to write a single expression on more than one line, we must write it between the delimiters `:{` and `:}`

# Types in Haskell
## Examples

All Haskell expressions have a **type**. A type is a set of values with common properties. `Int` is the type of integer values, and `Float` is the type of floating-point numbers.

```
Prelude>:set +t
```

```
Prelude> 1
1
it :: Num p => p
Prelude> 'a'
'a'
it :: Char
Prelude> True
True
it :: Bool
```

```
Prelude> ['a','b','c']
"abc"
it :: [Char]
Prelude> "abc"
"abc"
it :: [Char]
Prelude> 1+2.4
3.4
it :: Fractional a => a
Prelude> (True,'c')
(True,'c')
it :: (Bool, Char)
```

Expressions which can rot be typed are rejected:

```
Prelude> [True 1]
   error: ...
```

```
Prelude> True+2
   error: ...
```

# Function definitions
## Some simple examples

```
Prelude> :set +t
Prelude> 1+2
3
it :: Num a => a
Prelude>  f::Int->Int->Int; f x y = 2*x+3*y
f :: Int -> Int -> Int
Prelude> f 3 4
18
it :: Num a => a
Prelude>  g x y = sqrt (x*x+y*y)
g :: Floating a => a -> a -> a
Prelude> g 3 4
5.0
it :: Floating a => a
Prelude> u = \x y-> x+y
u :: Num a => a -> a -> a
Prelude> u 1 2
3
it :: Num a => a
Prelude> (\x y-> x+y) 1 2
3
it :: Num a => a
```

# Sample session
## Explanations

1. `\x y->x+y` is an **anonymous function**, that is, a function without a name. Anonymous functions are also called **lambda-abstractions**. The backslash `\` is Haskell's way of expressing the Greek letter $\lambda$
2. In Haskell, we can define **polymorphic functions**: they can operate on values from a family of types. For example, if we define the function

   ```
   Prelude> ii = \x y->x
   ii :: p -> q -> p
   ```

   the interpreter infers that `ii` is a function that takes as inputs arguments `x` of arbitrary type `p` and `y` of arbitrary type `q`, and returns a result of type `p`. This fact is indicated by the expression `ii :: p -> q -> p`.
3. Polymorphic types can be **qualified**. A qualified type is a type constrained to be a member of one or more **type classes**.
   - A **type class** (or just **class**) is a description of a colection of types.
   - **Types** are instances of a type class.

   (see next slide)

# Type classes and types
## Examples

The class *Eq* for types with equality (it is predefined in Haskell):

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

Type *Integer* is an instance of class *Eq* (it is predefined in Haskell):

```
instance Eq Integer where
   x == y = floatEq x y
```

The type inference system can infer qualified types:

```
Prelude> ff x y = if x == y then "yes" else "no"
ff :: Eq a => a -> a -> [Char]
```

This means, `ff` has qualified type $\forall(a \in Eq).a \to a \to [Char]$.

REMARKS:

- In general, [*a*] is the type of lists of type *a*. In particular, [*Char*] is the type of lists of characters. Strings are encoded as lists of characters.

# Type aliases and data types
## Examples

In Haskell we can define recursive datatypes:

- For convenience, we can use `type` to define aliases for given types:

```
type Radius = Float
type Side   = Float
type Vertex = (Float,Float)
```

- We can use `data` to define our own datatypes. E.g., we can define the data type `Shape` for geometric figures:

```
data Shape = Rectangle Side Side
           | Ellipse Radius Radius
           | RtTriangle Side Side
           | Polygon [Vertex]
     deriving Show
```

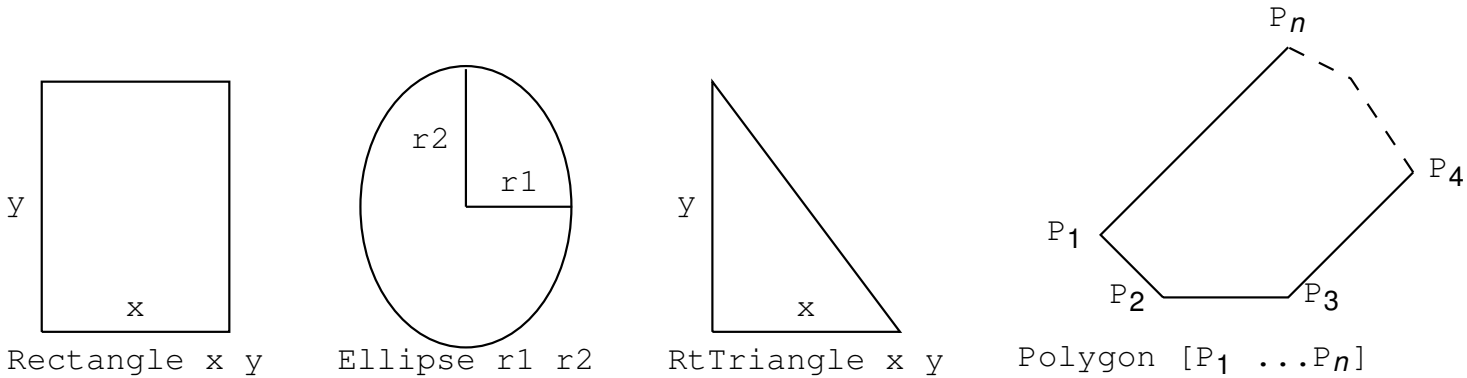
REMARKS:

- the datatype `Shape` has four type constructors: `Rectangle`, `Ellipse`, `RtTriangle`, and `Polygon`
- `deriving Show` indicates that type `Shape` is an instance of type class `Show`
  - class `Show` is a predefined class of types for which we can print the values.

# Data types
## Examples of `Shape` values

```
Prelude> s1 = Rectangle 3 4
it :: Shape
Prelude> s2 = Polygon [(0,0),(2,0),(0,3)]
it :: Shape
Prelude> [s1,s2]
[Rectangle 3.0 4.0,Polygon [(0.0,0.0),(2.0,0.0),(0.0,3.0)]]
it :: [Shape]
```

The intended meaning of geometric shapes:



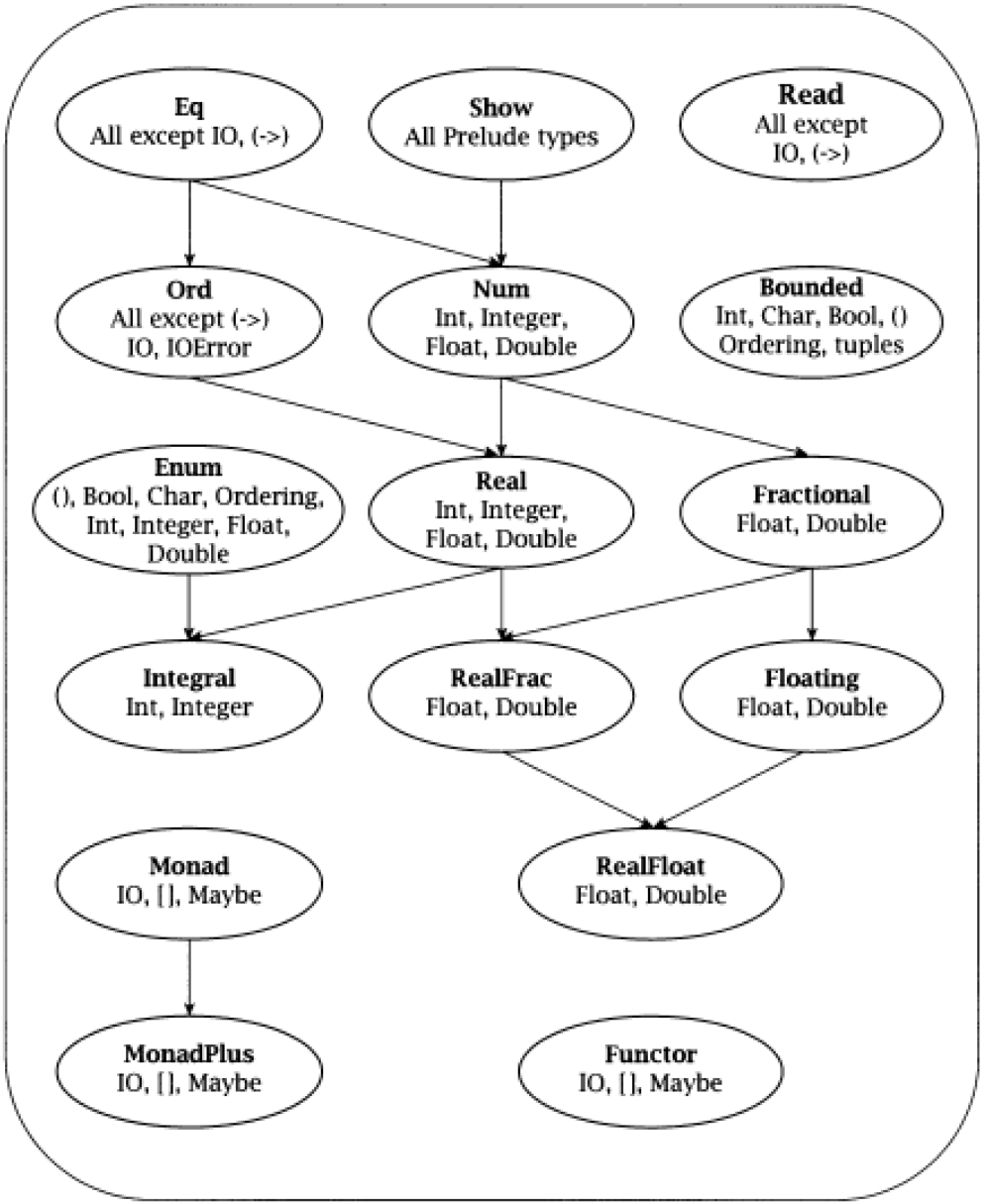
Example: The class *Ord* of ordered types inherits all the operations from the class *Eq* of types with equality:

```
class Eq a => Ord a where
    (<),(<=),(>),(>=) :: a -> a -> Bool
    max, min          :: a -> a -> a
```

Haskell permits multiple inheritance. E.g., the class of numeric types

```
class (Eq a, Show a) => Num a where
    (+),(-),(*) :: a -> a -> a
    negate      :: a -> a
    abs,signum  :: a -> a
    fromInteger :: Integer -> a
```

# Haskell classes for numeric types



# Recursive data types
## Example: a recursive data type for binary trees

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

- If we want to compare trees for equality, we assume `a` is an instance of class `Eq`, and make `Tree a` an instance of `Eq`:

```
instance Eq a => Eq (Tree a) where
    Leaf x == Leaf y                = x == y
    Branch l1 r1 == Branch l2 r2    = l1 == l2 && r1 == r2
    _ == _                          = False
```

- If we want to order trees in some increasing order, we should also assume that `a` is an instance of class `Ord`, and make `Tree a` an instance of class `Ord`:

```
instance Ord a => Ord (Tree a) where
    Leaf _ < Branch _ _           = True
    Leaf x < Leaf y               = x < y
    Branch _ _ < Leaf _           = False
    Branch l1 r1 < Branch l2 r2 = l1<l2 || (l1==l2 && r1<r2)
    t1 <= t2                      = t1==t2 || t1<t2
                                  ...
```

# Recursive datatypes
## Lists

Are predefined in Haskell:

- `[]` is the empty list
- `x:y` is the list with head `x` and tail `y`
- we can write `[x1,x2,...,xn]` instead of `x1:x2:...:xn:[]`

```
data List a = [] | a:(List a)
```

# Recursive datatypes
## Lists

Are predefined in Haskell:

- `[]` is the empty list
- `x:y` is the list with head `x` and tail `y`
- we can write `[x1,x2,...,xn]` instead of `x1:x2:...:xn:[]`

```
data List a = [] | a:(List a)
```

Functions on data types can be defined by case distinction. E.g.

```
Prelude> listLength [] = 0 ; listLength (_:xs) = 1+length xs
listLength :: [a] -> Int
Prelude> listLength [1,2,3]
3
it :: Int
Prelude> :{
Prelude| listSum [] = 0
Prelude| listSum (x:xs) = x + listSum xs
Prelude| :}
listSum :: Num p => [p] -> p
Prelude> listSum [1,2,3]
6
it :: Num p => p
```

# Function definitions
## More examples

A function definition with a guard:

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n | n > 0 = n * factorial (n-1)
```

# Function definitions
## More examples

A function definition with a guard:

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n | n > 0 = n * factorial (n-1)
```

All functions are lazy by default:

```
Prelude> nats n = n:nats (n+1)
nats :: Num t => t -> [t]
Prelude> take 7 (nats 0)
[0,1,2,3,4,5,6]
it :: Num a => [a]
```

# Function definitions
## More examples

A function definition with a guard:

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n | n > 0 = n * factorial (n-1)
```

All functions are lazy by default:

```
Prelude> nats n = n:nats (n+1)
nats :: Num t => t -> [t]
Prelude> take 7 (nats 0)
[0,1,2,3,4,5,6]
it :: Num a => [a]
```

REMARKS:

- `nats` is a lazy functions: the evaluation of `nats 0` with start displaying indefinitely the list of natural numbers, starting from 0
- `take k lst` is a predefined function: it returns the first `k` elements of list `lst`:

  The evaluation of `7 (nats 0)` will demand the evaluation of `(nats 0)` to `0:1:2:3:4:5:6:(nats 7)`

# Lazy functions
## A lazy version of Newton's method to compute $\sqrt{x}$

### Main ideas

- Create an infinite list `sqrtGuesses` of approximations of $\sqrt{x}$, starting with first approximation `1`.
- Traverse `sqrtGuesses` until we find the first approximation which is accurate enough and return it.

```
sqrt x = head (dropWhile (not . goodEnough) sqrtGuesses)
   where
      goodEnough guess = (abs (x - guess*guess))/x < 0.00001
      improve guess = (guess + x/guess)/2.0
      sqrtGuesses = 1:(map improve sqrtGuesses)
```

### Remarks

- `dropWhile (not . goodEnough) sqrtGuesses` drops the approximations from the front of the list that are not close enough.
- `(not . goodEnough)` is a **function composition**. It applies `goodEnough` to the approximation and then applies the boolean function `not` to the result. So `(not . goodEnough)` is a function that returns true if `goodEnough` returns false.

# Currying

Currying is a method `curry` which transforms a function

`curry` $f = \backslash x_1 \ \ldots \ x_n$ `->` *expr*

which expects $n$ arguments $x_1, \ldots, x_n$ into a composition of $n$ functions $f_1, f_2, \ldots, f_N$ which take 1 argument:

$$f \ = \ f_1 \ . \ f_2 \ . \ \cdots \ . \ f_n$$

- $f_i = \backslash x_i$ `->` `curry` $\backslash x_{i+1} \ \ldots \ x_n$ `->`*expr*
  for all $1 \leq i \leq n$

In Haskell, all functions are curried.

# Currying
## Example

- `map :: (a->b) -> [a] -> [b]` is a predefined higher-order function:

  `map f lst`

  takes as input a function `f:a->b` and a list `lst` of elements of type `a`, and returns the list produced by applying `f` to each element of `lst`

- ```
  Prelude>
  Prelude> doubleList = map (\x -> 2*x)
  doubleList :: Num b => [b] -> [b]
  ```

  `\x -> 2*x` is an anonymous function of type `Num a => a->a`

  $\Rightarrow$ `map (\x -> 2*x)` has type `Num a => [a]->[a]`.
  This function expects a list of elements of numeric type `a`, and returns the list obtained by applying the anonymous function `\x -> 2*x` to each element of the input list:

  ```
  Prelude> doubleList [1,2,3]
  [2,4,6]
  it :: Num b => [b]
  ```
  ```