# Introduction to Haskell III

Rolf Fagerberg

Spring 2005

# More Haskell Syntax

## List comprehensions

Math: $\{x \in S \mid x \geq 1,\ x \text{ even}\}$

Haskell: `[ x | x <- S, x >= 1, isEven x ]`

General form: `[ exp | generators, guards]`

Examples:

`[ x+y | (x,y) <- [(1,2),(7,8)], y > 5 ]` $\leadsto$ `[15]`

`[ (i,j) | i<-[1,2,3,4], j<-[8,9], isEven i ]`
$\leadsto$ `[(2,8),(2,9),(4,8),(4,9)]`

`[ j^2 | i<-[[1,2],[10,20]], j<-i ]` $\leadsto$ `[1,4,100,400]`

`[ [j^2|j<-i] | i<-[[1,2],[10,20]] ]` $\leadsto$ `[[1,4],[100,400]]`

# More Haskell Syntax

## Lambda definitions

Nameless functions defined inline:

```
zipWith (\x y -> x^2 + y^2) [1,2,3] [2,3,4]
```

$\leadsto$ `[5,13,25]`

```
compose2 f g = \x y -> g (f x) (f y)
```

## Enumeration expression

Easy generation of lists of certain types (types in the `Enum` class, see later).

```
[3 .. 10]  ⇝  [3,4,5,6,7,8,9,10]
[3, 3.3 .. 4]  ⇝  [3.0,3.3,3.6,3.9]
['a', 'c' .. 'i']  ⇝  "acegi"
[False ..]  ⇝  [False,True]
```

# More Haskell Syntax

## Local definitions

`where`:

```
f x y
  | x  < 0 = -(sqx*sqy + sqx + sqy) + g y
  | x >= 0 =   sqx*sqy + sqx + sqy
    where
      sqx = x*x
      sqy = y*y
      g z = (max x z) + t
        where t = x*y*z
```

`let`:

```
f x = let y=x^3; z=log x in y*z + z^2
```

# More Haskell Syntax

`case`:

```
isOdd x
  = case (x `mod` 2) of
    0 -> False
    1 -> True
```

or

```
empty l = case (x `mod` 2) of {0 -> False; 1 -> True}
```

`if then else`:

```
isOdd x = if (x `mod` 2)==0 then "Even" else "Odd"
```

# Haskell Classes

Class = set of types

Classes defined by giving their signature = the set of functions required to be defined on the types in the class (so signature $\approx$ interface in Java).

```
class Eq a where
(==) ::  a -> a -> Bool
```

Adding types to the class:

```
instance Eq MyBool where
  (==) MyTrue  MyTrue  = True
  (==) MyFalse MyFalse = True
  (==) _       _       = False
```

# Context

Classes can be used as context, i.e. requirements on the parametric types used:

```
elem ::  Eq a => a -> [a] -> Bool
elem x [] = False
elem x (y:ys) = (x == y) || (elem x ys)
```

Can also be used in instance declarations:

```
instance Eq a => Eq [a] where
  (==) []     []     = True
  (==) (x:xs) (y:yx) = (x == y) && (xs == ys)
  (==) _      _      = False
```

Note: not all types are in the (built-in) class `Eq`. E.g. function types are not (it seems difficult to give an operational feasible definition of function equality).

# Overloading vs. Polymorphism

Polymorphism

One definition of function works for many types.

Overloading

Several definitions of the same function (i.e. same identifier), one for each type.

OO languages like Java normally have overloading but not polymorphism.

In Haskell, overloading eases coding (imagine naming a version of == for each type) and makes the notion of polymorphism stronger (more functions can be defined with the same code).

# Default Definitions

Class declarations can contain default definitions:

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

Now, instance declarations only need to define `/=` or `==`.
Defining (overriding) both is OK.

# Derived Classes

Classes can be derived from other classes (again using the context notation):

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

When declaring a type an instance of `Ord`, the methods of `Eq` are inherited.

Thus, type classes form a hierarchy rather like the class hierarchy in OO languages.

# Built-In Classes

The standard prelude contains many predefined type classes.

E.g. for equality (`Eq`), ordering (`Ord`), enumeration(`Enum`), serialization (`Show`, `Read`), numeric types (`Int`, `Integer`, `Float`, `Double`, `Rational`, `Complex`).

Literals may be overloaded, which can lead to ambiguities for Haskell. Of what type is e.g. `2+3`? It may be necessary to resolve explicitly:

```
(2+3)::Int
```