EDAN40: Functional Programming
Standard Prelude Overview

Jacek Malec
Dept. of Computer Science, Lund University, Sweden
November 7th, 2012

Infromation sources

- A Tour of the Haskell Prelude;
- The Haskell Report:
  - Chapter 8 in Haskell 98
  - Chapter 9 in Haskell 2010

Links on the *links* page.

Below come only **selected** functions.

Basics

```
id      :: a -> a

const   :: a -> b -> a

(.)     :: (b -> c) -> (a -> b) -> a -> c

curry   :: ((a, b) -> c) -> a -> b -> c
uncurry :: (a -> b -> c) -> ((a, b) -> c)

($)     :: (a -> b) -> a -> b
f x $ g y = f x (g y)
```


Enumerated types

```
fromEnum         :: Enum a => a -> Int

toEnum           :: Enum a => Int -> a
toEnum 0 :: Bool = False

pred             :: Enum a => a -> a
pred True = False

succ             :: Enum a => a -> a
succ False = True
```

Enumerated types

```
enumFrom          :: Enum a => a -> [a]
[n..]

enumFromThen      :: Enum a => a -> a -> [a]
[m,n..]

enumFromThenTo    :: Enum a => a -> a -> a -> [a]
[m,n..o]

enumFromTo        :: Enum a => a -> a -> [a]
[m..n]
```


Pairs

```
fst               :: (a, b) -> a

snd               :: (a, b) -> b
```

Note: pairs only!

Union types

```
data Either a b  =  Left a | Right b

either :: (a -> c) -> (b -> c) -> Either a b -> c
either f g (Left x)  =  f x
either f g (Right y) =  g y
```

Example:

```
isNull :: Either String Integer -> Bool
isNull = either (=="") (==0)
```


Types with failure

```
data Maybe a  =  Nothing | Just a

maybe      :: b -> (a -> b) -> Maybe a -> b
maybe 0 (+1) (Just 1) = 2

lookup     :: Eq a => a -> [(a, b)] -> Maybe b
```

Lists

```
length          :: [a] -> Int
length "Abc" = 3

elem            :: (Eq a) => a -> [a] -> Bool
notElem         :: (Eq a) => a -> [a] -> Bool
'a' `elem` "abc" = True

(!!)            :: [a] -> Int -> a
[0,1,2] !! 1 = 1

(++)            :: [a] -> [a] -> [a]
"abc" ++ "def" = "abcdef"

concat          :: [[a]] -> [a]
concat ["a","bc","d"] = "abcd"
```


Lists

```
(:)             :: a -> [a] -> [a]
'a':"bc" = "abc"

head            :: [a] -> a
head "abc" = 'a'

tail            :: [a] -> [a]
tail "abc" = "bc"

init            :: [a] -> [a]
init "abcd" = "abc"

last            :: [a] -> a
last "abcde" = 'e'

reverse         :: [a] -> [a]
reverse "abc" = "cba"
```


Lists

```
filter          :: (a -> Bool) -> [a] -> [a]

map             :: (a -> b) -> [a] -> [b]

foldl           :: (a -> b -> a) -> a -> [b] -> a
foldl (+) 0 [a,b,c] = ((0+a)+b)+c

foldl1          :: (a -> a -> a) -> [a] -> a
foldl1 (+) [a,b,c] = (a+b)+c

foldr           :: (a -> b -> b) -> b -> [a] -> b
foldr (+) 0 [a,b,c] = a+(b+(c+0))

foldr1          :: (a -> a -> a) -> [a] -> a
foldr (+) [a,b,c] = a+(b+c)
```


Lists

```
scanl           :: (a -> b -> a) -> a -> [b] -> [a]
scanl (+) 0 [1,2,3] = [0,1,3,6]

scanl1          :: (a -> a -> a) -> [a] -> [a]
scanl1 (+) [1,2,3] = [1,3,6]

scanr           :: (a -> b -> b) -> b -> [a] -> [b]
scanr (+) 0 [1,2,3] = [6,5,3,0]

scanr1          :: (a -> a -> a) -> [a] -> [a]
scanr1 (+) [1,2,3] = [6,5,3]
```

Lists

```
zip      :: [a] -> [b] -> [(a, b)]
zip "abc" "de" = [('a','d'), ('b',e')]

unzip    :: [(a, b)] -> ([a], [b])
unzip [('a','b'),('c','d')] = ("ac",bd")

zipWith  :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith (+) [1,2] [3,4] = [4,6]

zip3     :: [a] -> [b] -> [c] -> [(a, b, c)]
unzip3   :: [(a, b, c)] -> ([a], [b], [c])
zipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
```


Lists

```
repeat           :: a -> [a]
repeat 'a' = "aaaaaaaaa..."

replicate        :: Int -> a -> [a]
replicate 4 'a' = "aaaa"

cycle            :: [a] -> [a]
cycle "abc" = "abcabcabc ..."

iterate          :: (a -> a) -> a -> [a]
iterate (++ " ") "" = ["", " ", "  ",...]

until            :: (a -> Bool) -> (a -> a) -> a -> a
until (> 3) (+ 2) 0 = 4
```


Lists

```
take             :: Int -> [a] -> [a]
take 3 "abcde" = "abc"

drop             :: Int -> [a] -> [a]
drop 2 "abcd" = "cd"

splitAt          :: Int -> [a] -> ([a], [a])
splitAt 2 "abcdef" = ("ab","cdef")

takeWhile        :: (a -> Bool) -> [a] -> [a]
takeWhile (> 2) [3,2,1] = [3]

dropWhile        :: (a -> Bool) -> [a] -> [a]
dropWhile (>3) [5,3,5] = [3,5]
```


Lists

```
span             :: (a -> Bool) -> [a] -> ([a], [a])
span isAlpha "ab cd" = ("ab"," cd")

break            :: (a -> Bool) -> [a] -> ([a], [a])
break (>=2) [1,2,3] = ([1],[2,3])
```

Lists (Strings)

```
words             :: String -> [String]
words "ab d as+3" = ["ab","d","as+3"]

unwords           :: [String] -> String

lines             :: String -> [String]

unlines           :: [String] -> String
```


Lists

```
sum               :: (Num a) => [a] -> a
sum [1,2,3] = 6

product           :: (Num a) => [a] -> a

and               :: [Bool] -> Bool
and [True, True, True] = True

or                :: [Bool] -> Bool

all               :: (a -> Bool) -> [a] -> Bool
all (/= 'a') "cba" = False

any               :: (a -> Bool) -> [a] -> Bool
any (== 'c') "abc" = True
```


Lists

```
max               :: (Ord a) => a -> a -> a

maximum           :: (Ord a) => [a] -> a

min               :: (Ord a) => a -> a -> a

minimum           :: (Ord a) => [a] -> a
```


To and from text

```
show              :: (Show a) => a -> String

read              :: (Read a) => String -> a
```

Basic I/O

```
putChar          :: Char -> IO ()
putStr           :: String -> IO ()
putStrLn         :: String -> IO ()
--        adds also a newline

getChar          :: IO Char
-- eof generates an IOError

getLine          :: IO String
-- eof generates an IOError
```


Other Libraries

- Ratio: rational numbers
- Complex: complex numbers
- Numeric: assorted numeric function
- Ix: mapping to integer ranges
- Array: efficient implementation of arrays
- List: additional list functions
- Maybe: some functions on the Maybe type
- Char: character conversions and tests
- Monad: additional functions on monads

Other Libraries

- IO: more advanced I/O features
- Directory: operations on file system directories
- System: basic OS interaction
- Time: date and time functions
- Locale: adaptation to local conventions
- CPUTime: functions to access current CPU time
- Random: functions generating random number sequences

Haskell 2010

Changes compared to Haskell 98:

- Foreign function interface (FFI);
- Hierarchical module names (e.g. `Data.Bool`.);
- Pattern guards.

```
addLookup env var1 var2
   | Just val1 <- lookup env var1
   , Just val2 <- lookup env var2
   = val1 + val2
{-...other equations...-}
```

Libraries in Haskell 2010

- `Control.Monad`
- `Data.Array`, `Data.Bits`, `Data.Char`, `Data.Complex`, `Data.Int`, `Data.Ix`, `Data.List`, `Data.Maybe`, `Data.Ratio`, `Data.Word`
- `Foreign.*`
- `Numeric`
- `System.Environment`, `System.Exit`, `System`,`IO`, `System.IO.Error`