EDAF95/EDAN40: Functional Programming
Standard Prelude Overview

Jacek Malec
Dept. of Computer Science, Lund University, Sweden

March 25th, 2024

Indentation

```
-- the first 'f' defines THE column
main = do foo 1
          foo 2
          if pizza
            -- indented further
            then foo 3
            else foo 4
          bar 5
   baz = fafafa -- first line indented less than 'f'
```

Indentation

```
module Main where

{main = do {foo 1
           ;foo 2
           ;if pizza

              then foo 3
              else foo 4
           ;bar 5
}    ;baz = fafafa
}
```

Basic I/O

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

`()` is the empty tuple (a.k.a. *unit*). It's type is also `()`!

```
getChar          :: IO Char
-- eof generates an IOError
```

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

Check Chapter 7 in Haskell 2010 report!

Sequencing I/O

The type constructor `IO` is an instance of the `Monad` class. There are two monadic binding functions used to sequence operations.
`>>` is used when the result of the first operation is uninteresting (e.g. is `()`).
`>>=` passes the result of the first operation as an argument to the second.

```
(>>=) :: IO a -> (a -> IO b) -> IO b
(>>)  :: IO a -> IO b        -> IO b
```

Example:

```
main = readFile "infile"                         >>= \ s ->
       writeFile "outfile" (filter isAscii s) >>
       putStr "Filtering successful\n"
```

Sequencing I/O

Do-notation: syntactic sugar for bind (>>=) and then (>>)

```
main = do
        putStr "Input file: "
        ifile <- getLine
        putStr "Output file: "
        ofile <- getLine
        s <- readFile ifile
        writeFile ofile (filter isAscii s)
        putStr "Filtering successful\n"
```

Sequencing I/O

```
echoReverse = do
    aLine <- getLine
    putStrLn (reverse aLine)
```

is just

```
echoReverse =
    getLine >>= \aLine ->
    putStrLine (reverse aLine)
```

or

```
echoReverse = getLine >>= (\aLine -> putStrLine (reverse aLine))
```

Random numbers

```
pick :: RealFrac r => r -> [a] -> a
pick u xs = xs !! (floor.(u*).fromIntegral.length) xs
```

How to randomise `r`?

```
somethingRandom rs = do
  r <- randomIO :: IO Float
  return (pick r rs)
```

Modules

Each Haskell program is a collection of *modules*

Module is an organizational unit, controlling the name space

One module **must** be called `Main` and must export value `main`.

```
module A (x,y) where
x,y :: Int -> Int
x = (+1)
y = (*2)
```

Entity export and import

A module declares which *entities* (values, types and classes) it *exports* (implicitely all). `import` expression makes exported entities available in another module. E.g. assume `A` exports x and y:

```
import A
import A()
import A(x)
import qualified A
import qualified A()
import qualified A(x)
-- import A hiding ()
import A hiding (x)
import qualified A hiding (x)
import A as B
import A as B(x)
import qualified A as B
```

Module Prelude

Standard Prelude is a module available in every language implementation and implicitely imported always into all modules (unless there is an explicit import!)

- **The Haskell 2010 Report: Chapters 5, 6 and 9**

Described as core type definitions and three parts: `PreludeList`, `PreludeText` and `PreludeIO`. Purely presentational.

Library organization

1. Standard Prelude
2. Haskell 2010 Language definition (part II)
3. GHC
4. The Haskell platform
5. Hackage

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)
```

A word on style

```
f $ x = f x
(f . g) x = f (g x)
```

Implications:

```
putStrLn (take 8 (map foo (bar ++ "ack")))
```

can be rewritten as

```
putStrLn $ take 8 $ map foo $ bar ++ "ack"

(putStrLn . take 8 . map foo) (bar ++ "ack")

putStrLn . take 8 . map foo $ bar ++ "ack"
```

The last one is most preferable!
NB, `($)` has precedence 0 (lowest).

Operator precedence

```
infixr 9  .
infixr 8  ^, ^^, ..
-- (..) is a built-in syntax!
infixl 7  ., /, `quot`, `rem`, `div`, `mod`
infixl 6  +, -
-- The (:) operator is built-in syntax, and cannot legally
-- be given a fixity declaration; its fixity is given by:
--   infixr 5  :
infix  4  ==, /=, <, <=, >=, >
infixr 3  &&
infixr 2  ||
infixl 1  >>, >>=
infixr 1  =<<
infixr 0  $, $!, `seq`
```

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
foldr1 (+) [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
```

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`, `Foreign.C`, `Foreign.C.Error`, `Foreign.C.String`, `Foreign.C.Types`, `Foreign.ForeignPtr`, `Foreign.Marshal`, `Foreign.Marshal.Alloc`, `Foreign.Marshal.Array`, `Foreign.Marshal.Error`, `Foreign.Marshal.Utils`, `Foreign.Ptr`, `Foreign.StablePtr`, `Foreign.Storable`
- `Numeric`
- `System.Environment`, `System.Exit`, `System.IO`, `System.IO.Error`