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'

module Main where
{main = do {foo 1
       ;foo 2
       ;if pizza
          then foo 3
          else foo 4
       ;bar 5
} ;baz = fafafa
}
Entity export and import

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

```
import A
import A()
import qualified A
import qualified A()
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
```

Standard Prelude is a module available in every language implementation and implicitly 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

- Standard Prelude
- Haskell 2010 Language definition (part II)
- GHC
- The Haskell platform
- 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
fx $ gy = fx (g y)
```
A word on style

\[ f \times x = f x \]

\[(f \cdot g) x = f (g x)\]

Implications:

\[
\text{putStrLn (take 8 (map foo (bar ++ "ack")))}
\]
can be rewritten as

\[
\text{putStrLn $ take 8 $ map foo $ bar ++ "ack"}
\]

\[(\text{putStrLn . take 8 . map foo}) (\text{bar ++ "ack"})\]

\[
\text{putStrLn . take 8 . map foo $ bar ++ "ack"}
\]
The last one is most preferable!

NB, \(($)\) has precedence 0 (lowest).

EDAN40 Standard Prelude

Precedence

\[\text{infixr 9 .}\]
\[\text{infixr 8 ^, ^^, ..}\]
\[\text{infixl 6 +, -}\]

-- The (::) operator is built-in syntax, and cannot legally
-- be given a fixity declaration;
-- but its fixity is given by:
\[\text{infixr 5 :}\]
\[\text{infix 4 ==, /=, <, <=, >=, >}\]
\[\text{infixr 3 &&}\]
\[\text{infixr 2 ||}\]
\[\text{infixl 1 >>, >>=}\]
\[\text{infixr 1 =<<}\]
\[\text{infixr 0 $, $!, ‘seq’}\]

Jacek Malec, http://rss.cs.lth.se

EDAN40 Standard Prelude

Enumerated types

\[\text{fromEnum :: Enum a => a -> Int}\]
\[\text{toEnum :: Enum a => Int -> a}\]
\[\text{toEnum 0 :: Bool = False}\]
\[\text{pred :: Enum a => a -> a}\]
\[\text{pred True = False}\]
\[\text{succ :: Enum a => a -> a}\]
\[\text{succ False = True}\]

Jacek Malec, http://rss.cs.lth.se

EDAN40 Standard Prelude

Enumerated types

\[\text{enumFrom :: Enum a => a -> [a]}\]
\[\text{[n..]}\]
\[\text{enumFromThen :: Enum a => a -> a -> [a]}\]
\[\text{[m,n..]}\]
\[\text{enumFromThenTo :: Enum a => a -> a -> a -> [a]}\]
\[\text{[m,n..o]}\]
\[\text{enumFromTo :: Enum a => a -> a -> [a]}\]
\[\text{[m..n]}\]

Jacek Malec, http://rss.cs.lth.se
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
'abc' `notElem` "def" = True

(!!) :: [a] -> Int -> a

[0..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 "abcdef" = 'f'

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

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)

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]

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

Jacek Malec, http://rss.cs.lth.se

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 (> 3) [3,2,1] = [3]

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

Jacek Malec, http://rss.cs.lth.se

Lists (Strings)

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

unwords :: [String] -> String
lines :: String -> [String]
unlines :: [String] -> String

Jacek Malec, http://rss.cs.lth.se
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

max :: (Ord a) => a -> a -> a
maximum :: (Ord a) => [a] -> a
min :: (Ord a) => a -> a -> a
minimum :: (Ord a) => [a] -> a

show :: (Show a) => a -> String
read :: (Read a) => String -> a

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

getChar :: IO Char
-- eof generates an I0Error

getLine :: IO String
-- eof generates an I0Error

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"

Jacek Malec, http://rss.cs.lth.se

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"

Jacek Malec, http://rss.cs.lth.se

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)
- Control.Monad
- Numeric
- System.Environment, System.Exit, System.IO, System.IO.Error