EDAF95/EDAN40 Standard Prelude

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
}

EDAF95/EDAN40: Functional Programming Standard Prelude Overview

Jacek Malec
Dept. of Computer Science, Lund University, Sweden
March 30th, 2020

EDAF95/EDAN40 Standard Prelude

Indentation

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* (implicitly 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 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`
- `f x $ g y = f x (g y)`
**A word on style**

\[ f \; \$ \; x = f \; x \]

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

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")`

The last one is most preferable!

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

**Precedence**

- `infixr 9 .`
- `infixr 8 ^, ^^, ..`
- `infixl 6 +, -`

**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`

- `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 -> [a] [m..n]`
Pairs

\[
\begin{align*}
\text{fst} & : (a, b) \to a \\
\text{snd} & : (a, b) \to b
\end{align*}
\]

Note: pairs only!

Union types

\[
data \text{ Either } a \ b = \text{ Left } a \mid \text{ Right } b
\]

\[
either : (a \to c) \to (b \to c) \to \text{ Either } a \ b \to c
\]

\[
either f g \text{ (Left } x) = f x
\]

\[
either f g \text{ (Right } y) = g y
\]

Example:

\[
\text{isNull} : \text{ Either } \text{ String } \text{ Integer} \to \text{ Bool}
\]

\[
\text{isNull} = \either (==") (==0)
\]

Types with failure

\[
data \text{ Maybe } a = \text{ Nothing } \mid \text{ Just } a
\]

\[
\text{maybe} : b \to (a \to b) \to \text{ Maybe } a \to b
\]

\[
\text{maybe} 0 (+1) \text{ (Just } 1) = 2
\]

\[
\text{lookup} : \text{ Eq } a \to a \to [(a, b)] \to \text{ Maybe } b
\]

Lists

\[
\text{length} : [a] \to \text{ Int}
\]

\[
\text{length } "\text{ Abc"} = 3
\]

\[
\text{elem} : (\text{ Eq } a) \to a \to [a] \to \text{ Bool}
\]

\[
\text{notElem} : (\text{ Eq } a) \to a \to [a] \to \text{ Bool}
\]

\[
\text{"a" \ 'elem' } "\text{ abc"} = \text{ True}
\]

\[
(!!) : [a] \to \text{ Int } \to a
\]

\[
[0,1,2] !! 1 = 1
\]

\[
(++) : [a] \to [a] \to [a]
\]

\[
"\text{ abc"} ++ "\text{ def"} = "\text{ abcdef"
\]

\[
\text{concat} : [[a]] \to [a]
\]

\[
\text{concat } ["a","bc","d"] = "\text{ abcd"
\]

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

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)

zip :: [a] -> [b] -> [(a, b)]
unzip :: [(a, b)] -> ([a], [b])
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
Lists

\[
\begin{align*}
\text{repeat} & : a \rightarrow [a] \\
\text{repeat } 'a' & = "aaaaaaaaa..." \\
\text{replicate} & : \text{Int} \rightarrow a \rightarrow [a] \\
\text{replicate } 4 'a' & = "aaaaa" \\
\text{cycle} & : [a] \rightarrow [a] \\
\text{cycle } "abc" & = "abcabcabc..." \\
\text{iterate} & : (a \rightarrow a) \rightarrow a \rightarrow [a] \\
\text{iterate } (++ " ") & = ["", ",", ",", ...] \\
\text{until} & : (a \rightarrow \text{Bool}) \rightarrow (a \rightarrow a) \rightarrow a \rightarrow a \\
\text{until } (> 3) (\rightarrow 2) & = 4 \\
\end{align*}
\]

Lists (Strings)

\[
\begin{align*}
\text{span} & : (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow ([a], [a]) \\
\text{span } \text{isAlpha } "ab cd" & = ("ab"," cd") \\
\text{break} & : (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow ([a], [a]) \\
\text{break } (>2) [1,2,3] & = ([1],[2,3]) \\
\end{align*}
\]

\[
\begin{align*}
\text{take} & : \text{Int} \rightarrow [a] \rightarrow [a] \\
\text{take } 3 "abcde" & = "abc" \\
\text{drop} & : \text{Int} \rightarrow [a] \rightarrow [a] \\
\text{drop } 2 "abcd" & = "cd" \\
\text{splitAt} & : \text{Int} \rightarrow [a] \rightarrow ([a], [a]) \\
\text{splitAt } 2 "abcdef" & = ("ab", "cdef") \\
\text{takeWhile} & : (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \\
\text{takeWhile } (>3) [3,2,1] & = [3] \\
\text{dropWhile} & : (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \\
\text{dropWhile } (>3) [5,3,5] & = [3,5] \\
\end{align*}
\]

\[
\begin{align*}
\text{words} & : \text{String} \rightarrow [\text{String}] \\
\text{words } "ab d as+3" & = ["ab","d","as+3"] \\
\text{unwords} & : [\text{String}] \rightarrow \text{String} \\
\text{lines} & : \text{String} \rightarrow [\text{String}] \\
\text{unlines} & : [\text{String}] \rightarrow \text{String} \\
\end{align*}
\]
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

To and from text

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

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.

\[
(\gg\gg) :: IO a \to (a \to IO b) \to IO b \\
(\gg) :: IO a \to IO b \to IO b
\]

Example:

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

Sequencing I/O

Do-notation: syntactic sugar for bind (\gg\gg) and then (\gg)

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

Random numbers

`pick :: RealFrac r => r -> [a] -> a`

```haskell
pick u xs = xs !! (floor.(u*).fromIntegral.length) xs
```

How to randomise `r`?

```haskell
somethingRandom rs = do
               r <- randomIO :: IO Float
               return (pick r rs)
```
• Control.Monad
• Numeric
• System.Environment, System.Exit, System.IO, System.IO.Error