I217E: Functional Programming

Nao Hirokawa

JAISTT

Term 2-1, 2022

http://www.jaist.ac.jp/~hirokawa/lectures/fp/

Schedule

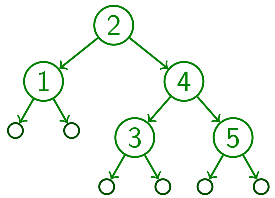
10/12	introduction	11/9	interpreters
10/14	algebraic data types I	11/11	compilers
10/19	algebraic data types II	11/16	termination
10/21	program reasoning	11/18	confluence
10/26	applications	11/25	verification
10/28	data structures I	11/30	review
11/2	data structures II	12/5	exam
11/4	computational models		

Evaluation

exam (60) + reports (40)

Advanced Types

Binary Trees

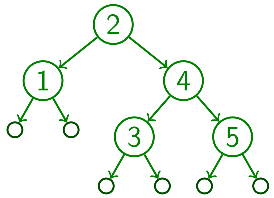


data Tree = Leaf | Node Tree Int Tree

Node (Node Leaf 1 Leaf) 2
(Node (Node Leaf 3 Leaf) 4 (Node Leaf 5 Leaf))

```
member :: Int -> Tree -> Bool
member x Leaf          = False
member x (Node l y r) =
  x == y || member x l || member x r
```

Tree Traversal



preorder	2	1	4	3	5
in-order	1	2	3	4	5
postorder	1	3	5	4	2

```
inorder :: Tree -> [Int]
inorder Leaf          = []
inorder (Node l x r) =
  inorder l ++ [x] ++ inorder r
```

Exercise

define `preorder`

Type Classes for Function Overloading

```
class Size a where
  size :: a -> Int

instance Size [a] where
  size xs = length xs

instance Size Tree where
  size Leaf         = 0
  size (Node l x r) = 1 + size l + size r

totalSize :: Size a => [a] -> Int
totalSize xs = sum [ size x | x <- xs ]
```


Pretty-Printing Trees

Note

Haskell interpreter uses `show` to output values

```
instance Show Tree where
  show Leaf         = "Leaf"
  show (Node l x r) =
    "(Node " ++ show l ++ " " ++
    show x ++ " " ++ show r ++ ")"
```


Equality on Trees

Note

== belongs to type class `Eq`

```
instance Eq Tree where
  Leaf         == Leaf         = True
  Node l1 x1 r1 == Node l2 x2 r2 =
    x1 == x2 && l1 == l2 && r1 == r2
```

Automatic Generation

```haskell
data Tree = Leaf | Node Tree Int Tree
  deriving Show
```

```haskell
data Tree = Leaf | Node Tree Int Tree
  deriving Eq
```

```haskell
data Tree = Leaf | Node Tree Int Tree
  deriving (Show, Eq)
```


Data Types vs Type Aliases

```haskell
data Tree = Leaf | Node Tree Int Tree
type Forest = [Tree]

-- data [a] = [] | a : [a]
-- data Bool = False | True
-- data Maybe a = Nothing | Just a

-- type String = [Char]
```

Remark

`instance Show String` is unavailable (why?)

Exercise for Type Classes

for $a = [(x_1, y_1), \ldots, (x_n, y_n)]$

$$\texttt{myLookup}\ x\ a = \begin{cases} \text{Just } y_k & \text{if } x = x_i \text{ for some } i \\ \text{Nothing} & \text{otherwise} \end{cases}$$

$$\text{where } k = \min\{i \mid x = x_i\}$$

for example

$$\texttt{myLookup}\ 2\ [(1, \texttt{"Jan"}), (2, \texttt{"Feb"})] = \text{Just } \texttt{"Feb"}$$
$$\texttt{myLookup}\ 3\ [(1, \texttt{"Jan"}), (2, \texttt{"Feb"})] = \text{Nothing}$$

1. declare type of `myLookup`
2. implement `myLookup`

Pattern Matching

Case Expressions and Pattern Guards

```
monthNames = [(1, "Jan"), (2, "Feb")]

monthName1 n =
  case lookup n monthNames of
    Just s  -> s
    Nothing -> "not found"

monthName2 n
  | Just s <- lookup n monthNames = s
  | otherwise = "not found"
```


Pattern Guards

```
partition p [] = ([], [])
partition p (x : xs)
  | p x       = (x : ys, zs)
  | otherwise = (ys, x : zs)
  where (ys, zs) = partition p xs

partition p [] = ([], [])
partition p (x : xs)
 | True <- p x, (ys,zs) <- partition p xs =
     (x : ys, zs)
 | (ys,zs) <- partition p xs =
     (ys, x : zs)
```


Pattern Matching in List Comprehensions

$$[\, x + y \mid (x, y) \texttt{<-} [(1, 10), (2, 20)]\,] = [11, 22]$$

$$[\, x \mid \text{Just } x \texttt{<-} [\text{Just } 1, \text{Nothing}, \text{Just } 2]\,] = [1, 2]$$

I/O

Making Stand-Alone Programs

Program

```
-- Hello.hs
main :: IO ()
main = putStr "Hello, World\n"
```

Compilation

`ghc Hello.hs` or `stack ghc Hello.hs`
generates `Hello.exe`

Do Notation

```
main =
  do { putStr "Hello "; putStr "World\n" }
```

equivalently,

```
main = do
  putStr "Hello"
  putStr "World!\n"
```


Homework

1. Visit `https://www.tryhaskell.org/` to go thourgh all lessons (1–5).
2. Implement `postorder` :: Tree $\to$ [Int].
3. Implement `nub` :: Eq a => [a] -> [a] that eliminates duplicating elements from a given list.

$$\texttt{nub}\ [1,2,3,3,3,2,4,1] = [1,2,3,4]$$
$$\texttt{nub}\ \texttt{"apple"} = \texttt{"aple"}$$

4. Install the graphic library **Gloss** by executing:

```
cabal update
cabal install gloss
```

Make sure that `GlossSample.hs` runs on your PC.