Haskell の演習

16:20～17:30 (70分)

演習の流れ

- 連想リストを検索する関数を作る
- 連想リストを検索する関数をテストする

- 二分木を作る
- 二分木を検索する関数を作る
- 連想リストを検索する関数をモデルとして
  二分木を検索する関数をテストする

リスト

- Haskell のリストは `[1,2,3,4]` のように書く
  - このリストの型は `[Int]`
- 最後の要素の後にカンマを書いてはならない
  - × `[1,2,3,4,]`
- 同じ型の要素だけを格納できる
  - ○ `['a','b','c'] :: [Char]`
  - × `['a',1,'b',2]`
- 空リストは `[]`
- リストの先頭に要素を追加するには：(コンス)を使う
  - `3:[]` → `[3]`
  - `2:[3]` → `[2,3]`
  - `1:[2,3]` → `[1,2,3]`

タプル

- 異なる型のデータをまとめる一番簡単な方法はタプル
- タプルは `('a',1)` のように書く
  - このタプルの型は `(Char,Int)`

連想リスト

- キーに対して値が収められているリストを連想リスト(association list)という
  - 機能はハッシュと同じで検索速度は遅い O(n)
  - 言語によっては辞書
- 連想リストはタプルとリストを組み合わせると作れる
  - `[('a',1),('b',2),('c',3)] :: [(Char,Int)]`

連想リストの検索

- 標準で lookup という関数が提供されている

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

  - 型を固定してみると

```
lookup :: Char -> [(Char, Int)] -> Maybe Int
```

- 対話環境 ghci で使ってみる

```
% ghci
> lookup 'b' [('a',1),('b',2)]
Just 2
> lookup 'c' [('a',1),('b',2)]
Nothing
```

- ここで Haskell のリテラルのすごさに感動すること！
  - ほとんどのデータの入力と出力が同じ形！！！

連想リストの検索(2)

- 「ファイル名.hs」に以下を書く

```
module Main where

main :: IO ()
main = print $ lookup 'b' [('a',1),('b',2)]
```

- コンパイラ ghc を使う

```
% ghc ファイル名
% ./a.out
```

- スクリプト実行コマンド runghc を使う

```
% runghc ファイル名
```

- Web を使う
  - フォームに貼付けて「実行」

```
http://mew.org:8080/runhaskell/
```

lookup の再発明

- lookup を search という名前で実装する
- まず型を考える

```
search :: Eq k => k -> [(k,v)] -> Maybe v
search = undefined
```

- 本体を undefined にしておけば
  型検査を通る

```
> :load file.hs
[1 of 1] Compiling Main ( file.hs, interpreted )
Ok, modules loaded: Main.
>
```

連想リストの検索

- search の型

```
search :: Eq k => k -> [(k,v)] -> Maybe v
```

- 基底部

```
search _ [] = □□□□□□□
```

- 再帰部

```
search k ((xk,xv):xs)
  | k == xk   = □□□□□□□
  | otherwise = □□□□□□□
```

- ヒント

```
data Maybe a = Nothing | Just a
```

```
data [a] = [] | a : [a]
```

```
[('a',1),('b',2),('c',3)]
```
は、以下の別表現

```
('a',1) : ('b',2) : ('c',3) : []
```

注意

インデントすると式が継続する

インデントには SPC を使う！
TAB を使ってはいけない！

search の答え

```
search :: Eq k => k -> [(k,v)] -> Maybe v
search _ [] = Nothing
search k ((xk,xv):xs)
  | k == xk   = Just xv
  | otherwise = search k xs
```

search をテストする

- lookup をモデルとして search をテストする

```
module Main where

import Test.QuickCheck

main :: IO ()
main = quickCheck prop_model0

prop_model0 :: Char -> [(Char, Int)] -> Bool
prop_model0 k xs = search k xs == lookup k xs

search :: Eq k => k -> [(k,v)] -> Maybe v
search _ [] = Nothing
search k ((xk,xv):xs)
  | k == xk   = Just xv
  | otherwise = search k xs
```

- テストしてみる

```
> quickCheck prop_model0
+++ OK, passed 100 tests.
```

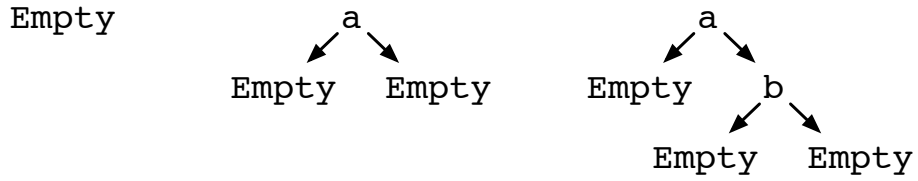
探索木

- 探索木のデータ構造を定義する

```
data Tree k v = Empty
              | Node k v (Tree k v) (Tree k v)
                deriving Show
```

- 例

```
Empty
Node 'a' 1 Empty Empty
Node 'a' 1 Empty (Node 'b' 2 Empty Empty)
```



探索木を生成するための補助関数

- 要素が0個の木を生成

```
empty :: Tree k v
empty = □□□□□
```

- 要素が1個の木を生成

```
singleton :: k -> v -> Tree k v
singleton k v = □□□□□
```

empty と singleton の答え

```
empty :: Tree k v
empty = Empty

singleton :: k -> v -> Tree k v
singleton k v = Node k v Empty Empty
```

探索木を作成するための関数

- insert の動作例

```
> insert ’b’ 2 $ insert ’a’ 1 empty
Node ’a’ 1 (Node ’b’ 2 Empty Empty) Empty
```

- insert を作成する
  - 破壊的な代入はないので、新しく木を作って返す

```
insert :: Ord k => k -> v -> Tree k v -> Tree k v
insert k v Empty = singleton k v
insert k v (Node xk xv l r)
  | k < xk    = insert □□□□□
  | otherwise = insert □□□□□
```

insert の間違った答え

```
insert :: Ord k => k -> v -> Tree k v -> Tree k v
insert k v Empty = singleton k v
insert k v (Node xk xv l r)
  | k < xk    = insert k v l
  | otherwise = insert k v r
```

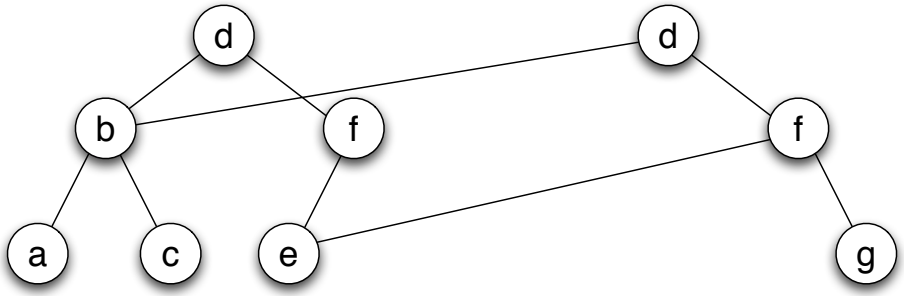
- 'a' が消えてしまう

```
> insert 'b' 2 $ insert 'a' 1 empty
Node 'b' 2 Empty Empty
```

insert の答え

```
insert :: Ord k => k -> v -> Tree k v -> Tree k v
insert k v Empty = singleton k v
insert k v (Node xk xv l r)
  | k < xk    = Node xk xv (insert k v l) r
  | otherwise = Node xk xv l (insert k v r)
```

- 木の一つのパス全体が新しく作られる
  - 他は共有される
  - 1000万ノードの平衡木なら、
    新しく作られるノードの数は13個



連想リストを木に変換する関数

- fromList の動作例

```
> fromList [('a',1),('b',2)]
Node 'a' 1 Empty (Node 'b' 2 Empty Empty)
```

- fromList の実装

```
fromList :: Ord k => [(k,v)] -> Tree k v
fromList xs = foldl insert' empty xs
  where
    insert' t (k,v) = insert k v t
```

- fold とは畳み込み関数
  - MapReduce の Reduce

木を探索する関数

- searchTree の動作例

```
> searchTree ’a’ $ fromList [(’a’,1),(’b’,2)]
Just 1
> searchTree ’c’ $ fromList [(’a’,1),(’b’,2)]
Nothing
```

- searchTree の実装

```
searchTree :: Ord k => k -> Tree k v -> Maybe v
searchTree _ Empty = □□□□□
searchTree k (Node xk xv l r)
  | k == xk   = □□□□□
  | k < xk    = □□□□□
  | otherwise = □□□□□
```

searchTree の答え

```
searchTree :: Ord k => k -> Tree k v -> Maybe v
searchTree _ Empty = Nothing
searchTree k (Node xk xv l r)
  | k == xk   = Just xv
  | k < xk    = searchTree k l
  | otherwise = searchTree k r
```

searchTree をテストする(1)

```
module Main where

import Test.QuickCheck

main :: IO ()
main = quickCheck prop_model1

prop_model1 :: Char -> [(Char, Int)] -> Bool
prop_model1 k xs = search k xs == searchTree k t
  where
    t = fromList xs
```

今まで作ったコード

- テストしてみる

```
% quickCheck prop_model1
+++ OK, passed 100 tests.
```

searchTree をテストする(2)

- prop_model1 は左から探索していた
- 右から探索しても結果は同じはず

```
prop_model2 :: Char -> [(Char, Int)] -> Bool
prop_model2 k xs = search k xs' == searchTree k t
  where
    xs' = reverse xs
    t = fromList xs
```

- 何回かテストしてみると...

```
> quickCheck prop_model2
*** Failed!
[('r',0),('r',1)]
```

- キーが重複することを考えてなかった！

searchTree を訂正する

```
insert :: Ord k => k -> v -> Tree k v -> Tree k v
insert k v Empty = singleton k v
insert k v (Node xk xv l r)
  | k == xk   = □□□□□
  | k < xk    = Node xk xv (insert k v l) r
  | otherwise = Node xk xv l (insert k v r)
```

訂正版 searchTree の答え

```
insert :: Ord k => k -> v -> Tree k v -> Tree k v
insert k v Empty = singleton k v
insert k v (Node xk xv l r)
  | k == xk   = Node k v l r   -- ここを追加
  | k < xk    = Node xk xv (insert k v l) r
  | otherwise = Node xk xv l (insert k v r)
```

- 今度はテストを通る

```
> quickCheck prop_model2
+++ OK, passed 100 tests.
```

- prop_model1 は間違ったテスト
  - lookup では、重複があると最初の方が発見される
  - fromList では、後の方が木に残る

おつかれさまでした