関数・論理型プログラミング実験
Haskell演習第1回
（通算第8回）

松田 一孝
TA: 武田広太郎 寺尾拓

今日からHaskell演習

- 6/3
- 6/10
- 6/17

} Haskell演習

- 6/24
- 7/1
- 7/8

} Prolog演習

- 7/15 } Curry演習
- 7/22 } 最終課題

Haskell？

- 遅延評価純粋関数型言語
  - 遅延評価（具体的にはcall by need)
    - if等を式として表現化
    - 評価ステップ数は高々値呼び程度
  - 純粋関数型
    - 「副作用」がない
      - I/O，参照は別の方法で実現

MLとの主な共通点

- 関数型
  - 高階関数を用いた簡潔で柔軟なプログラミング
    - map, filter, foldr, …
- 静的型付け
  - Haskellの型の基本的な部分はMLと同じlet多相
    - が，副作用がないのでvalue restrictionはない

なぜHaskell演習？

- MLと異なる関数型言語を学び，関数プログラミング/関数型言語への理解をより深める
  - MLとの共通点
    - 関数型言語の全般の性質
  - MLとの相違
    - 抽象化の違いと得失

参考図書・URL

- Learn You Haskell for Great Good!
  - http://learnyouahaskell.com/
  - チュートリアル
  - 書籍版もある
- Graham Hutton: Programming in Haskell, Cambridge University Press
  - ISBN: 978-0521692694
  - 定番の入門書

参考図書・URL

- Haskell 98 Language Report
  - http://www.haskell.org/onlinereport/
  - Haskell言語の構文・非形式的な意味
  - Haskell言語の仕様としては古いが，シンプルにまとまっている
    - 最新版はHaskell 2010 Language Report
      - 主な違いは階層的モジュール

参考図書・URL

- Jeremy Gibbons and Oege de Moor (ed.): The Fun of Programming
  - 関数プログラミングでは
    こんな面白いことができるぞ！

Haskell处理系ghc

Glasgow Haskell Compiler

- Haskellの事実上標準の処理系
  - さかんな研究・開発・進化
    - 研究→実装が速い
    - その分最新の機能は枯れていない
- 強力な最適化
  - (意識して書けば)
    遅延評価のオーバヘッドは消える
    - Computer Language Benchmarks Game参照
- 注：ocamlと違い他にも処理系が存在
  - jhc, uhc, …

ghcとghci

- ghc
  - コンパイラ
    - ネイティブコード作成をサポート
    - ソース間の依存性の解析もしてくれる
      - ocamlcと違いコンパイルの順番は通常は意識する必要はない

- ghci
  - インタプリタ
    - コードを読みこんだり，式を評価したり

ghciの起動

```
$ ghci
GHCi, version 7.4.1: http://www.haskell.org/
ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

プロンプトが表示
(Preludeは標準モジュール)

式の評価

式を入力しEnterで評価

```
Prelude> 1
1
Prelude> 1+2
3
Prelude> "a"
"a"
Prelude> :t "a"
"a" :: [Char]
Prelude> :t (let loop () = loop () in loop ())
(let loop () = loop () in loop ()) :: t
```

式の型を知るには:type
(:tや:tyや:typでもよい)

その際，式は評価されない

コードの読み込み

- :load ModuleName
- :load ファイル名

ファイル名の先頭は慣習的に大文字

```
$ cat Test.hs
inc x = x + 1
dec x = x - 1
$ ghci
...
Prelude> :load Test
[1 of 1] Compiling Main                ( Test.hs, interpreted )
Ok, modules loaded: Main.
*Main> inc 1
2
*Main>
```

Test.hsのロード

モジュール名は指定しなければMain
(が，通常モジュールXはX.hsで定義)

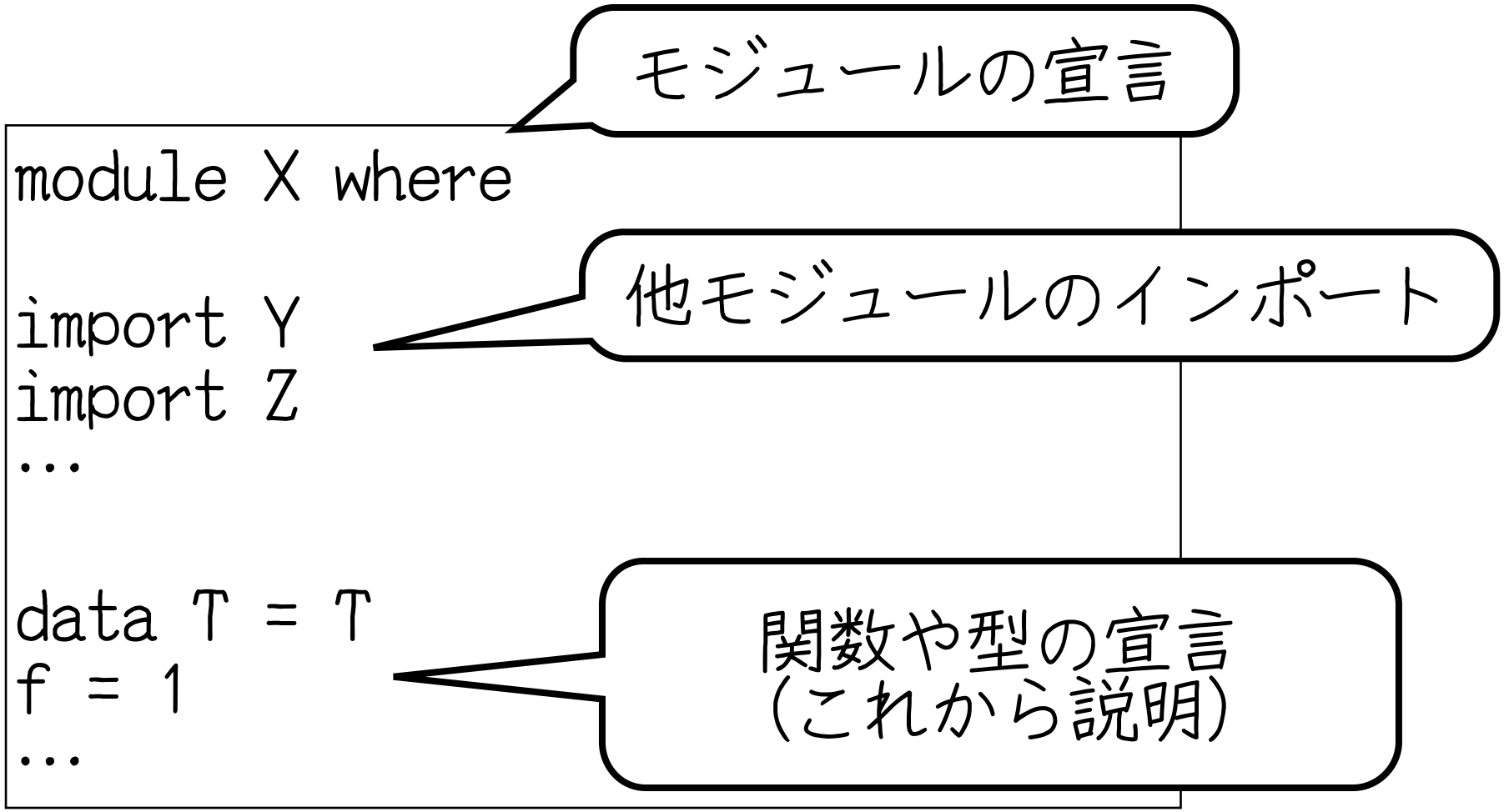
注意

- インタプリタには基本的に「式」か:loadなどの「コマンド」しか書けない
  （と思っておいたほうがよい）
  - 実際は，do記法（今週は紹介せず）中に書けるものと型の定義が書ける
  - どのみち改行がつかえないので大きなものは入力不可

Haskellの基本的な構文

Haskell 98 Language Reportや
Haskell 2010 Language Reportも参考のこと

Haskellコードの構造



（再帰）関数の宣言

- 関数名 パターン1_1 ... パターン1_n = 式$_1$
  関数名 パターン2_1 ... パターン2_n = 式$_2$

関数の型の宣言（必須ではないがあると読みやすい）

```
append :: [a] -> [a] -> [a]
append []     ys = ys
append (x:xs) ys =
    x:append xs ys
```

インデントでコードのまとまりを表現

相互再帰

- 特に特別な構文はない

```
isEvenLen []     = True
isEvenLen (x:xs) = isOddLen xs

isOddLen []     = False
isOddLen (x:xs) = isEvenLen xs
```

注意

- 以下はエラー
  - 同じ関数の宣言は連続して書かないとダメ
  - 同じ名前の関数は二個定義できない

```
f x = x + 1
g x = x - 1
f x = x + 1
```

Haskellの式

- コメント
- 整数
- 四則演算
- 論理演算
- 条件分岐
- 関数抽象・適用
- リスト・組
- パターンマッチによる分岐

コメント

- {- コメント -}
  - OCamlと同じくネスト可
- -- 行コメント
  - C++の//や，Perlの#のようなもの
  - なお，--|や--*等は
    演算子と解釈されるので注意

定数

- 整数：0, 1, 0xAb, 0o777
- 小数：2.0, 2.0e-3
  - 2.や.0はダメ
- 文字：'a'
- …

四則演算

- +, *, -, /, div, modなど
  - +, *, - : 和, 積, 差
  - /: 小数除算
  - div: 整数除算
  - mod: 整数剰余

- Camlと違い 2 + 3.0 は正しい式
  - 型クラスによる (後述)

論理演算

- if e_1 then e_2 else e_3
  - 条件分岐
- True, False
  - 真偽値
- (&&), (||), not
  - 論理積, 論理和, 否定
- (==), (/=)
  - 等価性検査

```
isLeapYear x =
  (x `mod` 4 == 0) &&
  ( (x `mod` 400 == 0) || (x `mod` 100 /= 0) )
```

関数抽象・適用

- 関数抽象：\x -> e
  - OCamlのfun x -> eと同じ
    - xの部分が変数でなくてパターンでよいところも
  - \はλに似ている？

- 関数適用：e_1 e_2
  - OCamlと同じ
    - 左結合で結合順位が最強なところも

リスト・組

- リスト
  - []
  - e_1 : e_2
    - OCamlと「:」と「::」の使いかたが逆
- 組
  - $(e_1, \ldots, e_n)$
  - OCamlと違い括弧は必須

リストに関する糖衣構文

- $[e_1, e_2, \cdots, e_n]$
  - e1 : e2 : … : en : []に同じ
    - OCamlとの違いは「;」が「,」なだけ
- [1..4]や[1,3..9]
  - それぞれ[1,2,3,4]と[1,3,5,7,9]
    - 「1」や「4」の部分は定数でなくてよい
- "abc"
  - ['a','b','c']に同じ
    - Haskellでは文字列は文字のリスト

パターンマッチによる分岐

- case e of
  p_1 -> e_1
  ...
  p_n -> e_n
  - OCamlのmatch-with式に相当

```
null x =
 case x of
   []    -> True
   _:_   -> False
```

捕捉

- case … of の入れ子関係は
  （基本は）インデントで表現
  - 以下はそれぞれ意味が違う

```
case x of
  A ->
    case y of
      B ->
      C ->
```

vs

```
case x of
  A ->
    case y of
      B ->
  C ->
```

便利な演算

- 関数合成 .

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(.) f g x = f (g x)
```

- 関数適用 $
  - 通常の適用と違い結合力最弱・右結合

```
($) :: (a -> b) -> a -> b
($) f x = f x
```

```
Prelude> succ $ succ $ 2 + 3
7
```

便利な記法

- 演算子のセクショニング
  - (+)は　\x y -> x + y と同じ
  - (+e)は \x -> x + e　と同じ
  - (e+)は \y -> e + y　と同じ

  - 注意：
    -については二番目の記法ができない
    * 単項マイナスになる

ユーザ定義型

- 型シノニム
  (type synonym)
  - ただの別名
- 代数的データ型
  (algebraic datatype)
  - OCamlでいうヴァリアント型

  - OCamlだとどっちもtypeだが,
    Haskellでは違う

型シノニム

- type 型名 型変数$_1$ … 型変数$_n$ = 型
  - 型に別の名前を付ける
    - 例：type String = [Char]
  - OCamlと違い型名は大文字スタート

代数的データ型

- data 型名 型変数$_1$ … 型変数$_n$
  = 構成子 型$_1$ … 型$_m$
  …
  | 構成子’ 型$_1$’… 型$_k$’
  - OCamlでいうヴァリアント型
  - 例
    - data Bool = True | False
      - OCamlと違い，本当にこういう風に真理値が定義されている（と考えてよい）
    - data BT a = Lf
      | Nd a (BT a) (BT a)

注意

- 相互再帰的な型を定義する
  特別な構文はない

```
data Value = VFunc String Exp Env
data Exp = EConst Value
         | …
data Env = [ (String, Value) ]
```

OCamlとの細かい差

- 構成子はそのまま関数として利用可
  - map Just xsはHaskellの正しい式
    - data Maybe a = Nothing | Just a
  - map Some xsはOCamlの誤った式
    - type 'a option = None | Some of 'a

Haskellのモジュール

- Haskellにもモジュールがある
  - 名前空間を提供
  - 実装と使用の分離
  - 分割コンパイルには必須

- MLのように構文で定義し束縛したり，変換したりはできない
  - が，課題でやったようなことは他の方法で可能

モジュールの利用

- import Module

importは関数や型の宣言の前

```
import Data.List
…
snub = map head . group . sort
```

モジュールData.Listの関数
groupやsortを利用

より細かな制御

- import Module (name1, name2, …)
  - 一部の名前だけをインポート
- import qualified Module
  - 修飾子付きの名前でインポート
    - Module.nameでアクセス
- import qualified Module as Q
  - 修飾子Qでインポート
    - Q.nameでアクセス

詳細はLanguage Reportを参照

モジュールの定義

- 基本，X.hsでXを定義
  - そうでないとコンパイラが探せない

```
{- Main.hs -}
module Main where
import Foo
main = putStrLn foo
```

```
{- Foo.hs -}
module Foo where
foo = "Hello World"
```

補足

- module M（関数名や型名等）where でエクスポートする関数等を制御可
  - 詳細はLanguage Report参照

分割コンパイル

- ghc --makeで勝手にやってくれる

```
$ ls
Main.hs  Foo.hs
$ ghc --make Main.hs
...
$ ./Main
Hello World
```

依存性を自動で解析し
分割コンパイル+リンク

```
{- Main.hs -}
module Main where
import Foo
main = putStrLn foo
```

```
{- Foo.hs -}
module Foo where
foo = "Hello World"
```

型クラス

型クラス

- 型等に対して可能な操作を抽象化
  - 型クラスの宣言の基本的な構文
    class 型クラス名 型変数 where
        宣言

Eqのインスタンスである型には(==)を使用できる

```
class Eq a where
    (==) :: a -> a -> Bool
```

- Javaのインタフェースに近い

型クラスの利用

コンテキスト：
Eqのインスタンスaについて…

```
{- Test.hs -}
allEqual :: Eq a => [a] -> Bool
allEqual []         = True
allEqual [x]        = True
allEqual (x:y:xs) = (x == y) && allEqual (y:xs)
```

```
$ ghci Test.hs
…
*Main> allEqual [1,1,1]
True
*Main> allEqual [not,not]
エラー
```

OK. IntはEqのインスタンス

NG. Bool->BoolはEqのインスタンスでない

インスタンス宣言

- 基本的な構文
  instance 型クラス 型 where
    宣言
  - 正確には型でなく型構成子でもOK

```
instance Eq Bool where
    True  == True  = True
    False == False = True
    _     == _     = False
```

インスタンス宣言

- コンテキストの利用

aがEqのインスタンスなら[a]も

```
instance Eq a => Eq [a] where
    []     == []     = True
    (a:x)  == (b:y)  = (a==b) && (x==y)
    _      == _      = False
```

複数指定するときは括弧と「,」で

```
instance (Eq a,Eq b) => Eq (a,b) where
    (x1,y1) == (x2,y2) = (x1==y1)&&(x2==y2)
```

補足

- 型クラスは「型」以外も扱える

```
class Functor where
    fmap :: (a -> b) -> f a -> f b
```

```
data MyList a = Nil | Cons a (MyList a)

instance Functor MyList where
     fmap f Nil = Nil
     fmap f (Cons x xs) =
          Cons (f x) (fmap f xs)
```

MyList aは型だが，MyListは型ではないことに注意

組み込みの型クラス

- Eq：等価性検査できる
  - (==), (/=)が利用可
- Ord：比較できる
  - (<=), (>=), (<), (>), max, min等
- Show：文字列に変換できる
  - class Show a where
    show :: a -> String

```
Prelude> show 1
"1"
Prelude> show (1, 2.00)
"(1,2.0)"
```

組み込みの型クラス

- Functor：「全要素に関数を適用する」関数が定義できる
  - fmap

```
Prelude> fmap (+1) [1,2]
[2,3]
Prelude> fmap (+1) (Just 1)
Just 2
```

- 他にも多数

deriving

- 一部の組み込み型クラスについては derivingでインスタンス宣言を 自動生成できる

```
{- Test.hs -}
data MyBool = MyFalse | MyTrue
    deriving (Eq, Ord, Show)
```

```
$ ghci
...
Prelude> show MyFalse
"MyFalse"
Prelude> MyFalse == MyTrue
False
```

注意

- インタプリタ上で式の結果を表示してくれるのはその式の型がShowのインスタンスであるときだけ

```
Prelude> (+1)

<interactive>:1:1:
    No instance for (Show (a0 -> a0))
      arising from a use of `print'
    Possible fix: add an instance declaration for
(Show (a0 -> a0))
    In a stmt of an interactive GHCi command: print it
```

注意の補足

- が，逆に適切にインスタンス宣言することで整形出力が可能

```
data Val = VInt  Int
         | VBool Bool
         …
instance Show Val where
    show (VInt i)  = show i
    show (VBool b) = show b
    …
```

```
Prelude> (VInt 1, VBool False)
(1,False)
```

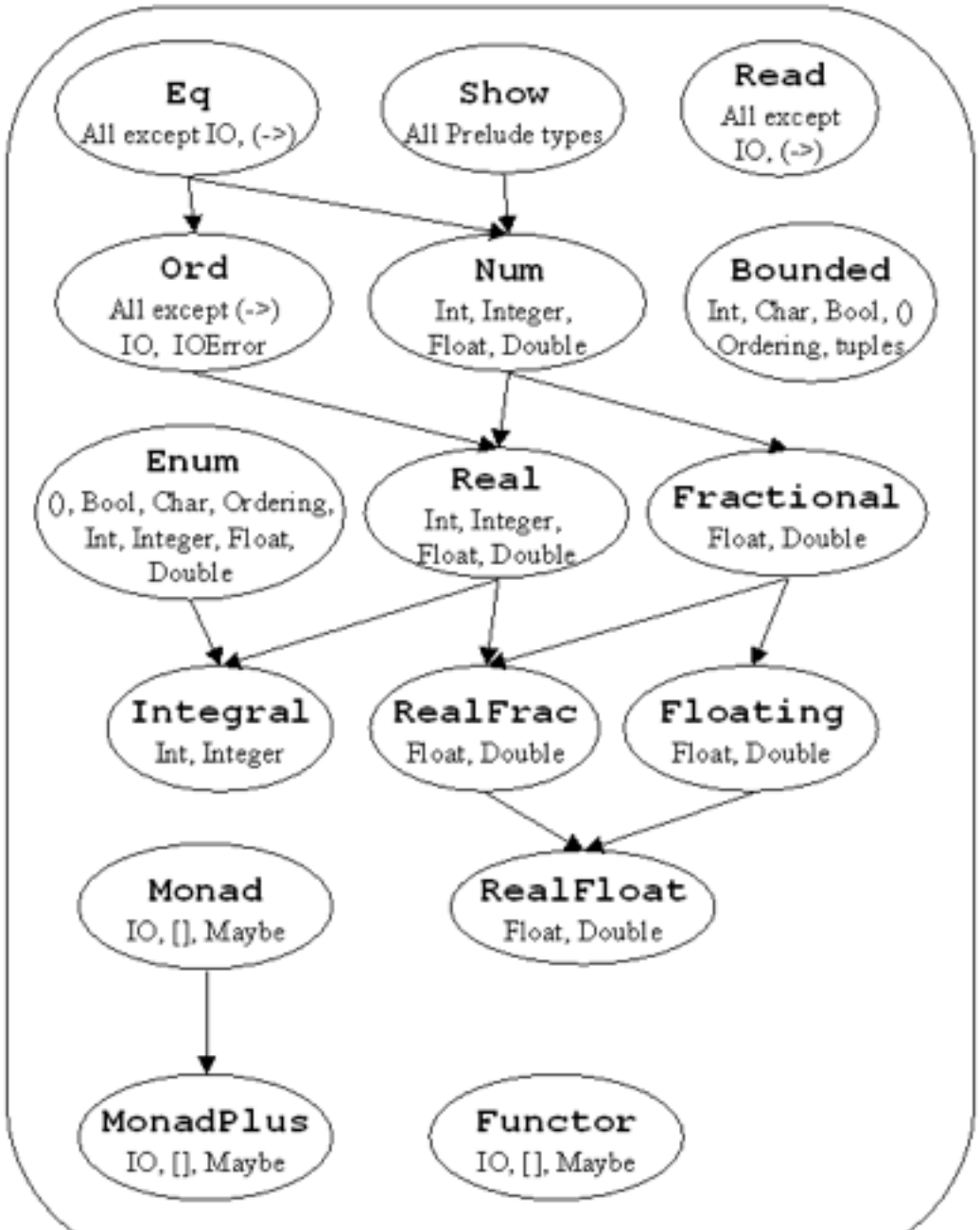
Haskellの整数定数

- Num：数を表す型クラス
- Integral：整数を〃
- Fractional：分数を〃

```
Prelude> :t 1
1 :: Num a => a
Prelude> :t (+)
(+) :: Num a => a -> a -> a
Prelude> :t (/)
(/) :: Fractional a => a -> a -> a
Prelude> :t div
div :: Integral a => a -> a -> a
```

標準の型と型クラス

- Haskell 98 Language Reportより



今回のまとめ

- Haskellの基本的な機能について簡単に説明した
  - 次回は型クラス（予定）についてもっと詳しく見る

第8回レポート課題
締切 6/17 13:00

問1

- 以下の関数をHaskellで書け
  - 挿入ソート： insSort
  - Quickソート：quickSort
  - 選択ソート： selectionSort
  - Mergeソート：mergeSort

- 型はどれも同じになるはず
  - プログラムは読み易さを重視すること
  - ただし，quickSortは平均O(nlogn)，mergeSortは最悪O(nlogn)で動作すること

問2

- 以下の無限リストを作成せよ
  - ones = [1, 1, 1, 1, …]
  - nats = [1, 2, 3, 4, …]
  - fibs = [1, 1, 2, 3, 5, 8, …]
  - primes = [2, 3, 5, 7, 11, …]

問3

- 以下の無限リストを作成せよ
  - natPairs =
    [(1,1),…,(n,m),…]
    - 任意の自然数の組が
      いずれ出現するように

問4

- 関数memoizeとmfibを以下で定義する

```
memoize f = let rs = map f [0..] in  (rs !!)
```

```
mfib = memoize
         (\x -> if x <= 2 then 1
                else mfib (x-1) + mfib (x-2))
```

  - このときmfib nの計算時間はnの多項式程度となる．なぜそうなるか説明せよ
  - memoize fを以下で定義すると上記のようには行かない．なぜか説明せよ

```
memoize f i = let rs = map f [0..] in  rs !! i
```

発展1（Bird's repmin）

- 二分木の型を以下で定義する

```
data T a = Fork (T a) (T a)
         | Tip a
```

  - この木を受け取り，その最小の要素で木の各要素を置き換えた木を返す関数repminを作成せよ
  - ただし，1-passで行うようにせよ
    - 1-pass：入力の木は再帰関数で一回だけ走査される
    - 1-passでないものは不可

補足

- 2-passなら簡単

```
repmin t = replace t (minim t)

replace (Tip x) m = Tip m
replace (Fork t1 t2) m =
   Fork (replace t1 m) (replace t2 m)

minim (Tip x) = x
minim (Fork t1 t2) =
   min (minim t1) (minim t2)
```

発展2

- 以下のコードの片方は型検査を通るがもう一方は通らない，何故か？

```
myId x = x
foo = (myId 1, myId "a")
```

```
myId x = let _ = foo in x
foo = (myId 1, myId "a")
```

- これを踏まえ，前回作成したインタプリタでHaskell風相互再帰を許すにはどうすればよいか議論せよ
  - グラフ上のある操作を使う？

発展3

- 次の二つの関数は同じではない
  - 正確にはfは組の上の恒等関数だが gが組の上の恒等関数ではない

```
f (x, y) = (x, y)
g x      = (fst x, snd x)
```

- h fとh gが異なる出力となるような hを定義せよ
  - ヒント：止まらない式を考える