`interact` のすすめ

それでも関数的に書きたいあなたに

2025-06-14

関数型まつり2025

山下伸夫 @ SampouOrg

目次

1. モチベーション
    - 「関数的」に書きたい
    - 「関数的」のお気持ち
2. お題：
    - AtCoder提出プログラムの雛形
    - Wordleパズルのチートプログラム
    - Interactive Sorting
3. まとめ：
    - I/O 分離
    - 明示的データ依存のない順序依存
    - 命令的記述に見えてしまうもの

モチベーション

「関数的」に書きたい

```
main :: IO ()
```

> おおもとが「関数」ではなく「命令」！？
>
> 「関数的」プログラミングがしたい

「命令的」への偏見

```
main :: IO ()
main = do
    {     x <- instructio0
    ; let y =  function0 x
    ;     z <- instruction1 z
    ;     instruction2
    ; let a =  function x z
          ..
    ;     instructionZ xx yy
    }
```

「関数的」のお気持

関数（function）と **関数型**（function type）

関数（function）

(skip)

> ここでの関数は数学で関数として扱える純粋関数を指すものとする。 純粋関数の「純粋」は、関数的ではないものは含まれないの意。 （出典不詳）

関数的（functional）

(skip)

> **Functional**(also called **right-unique** or **univalent**): for all $x \in X$ and all $y, z \in Y$, if xRy and xRz then $y = z$.
>
> WikiPedia: Binary relation

「関数的」のお気持

(skip)

	関数的	命令的
思考	トップダウン	ボトムアップ
時間	静的	動的
計算	関数適用	命令実行
区切	関数合成演算子 `.`	命令区切り子 `;`

- フォン・ノイマン型計算機を前提としない「お花畑」
- 対象はアイデンティティに影響しない属性をもたない

関数、関数型（お気持）

関数は関数型（function type）の値

`a -> b` は `a` 型の値に**適用すると** `b` 型の値になるような関数（の値）の型である。

`a` および `b` が型ならば `a -> b` は型である

このとき `a` を域（domain）の型、 `b` を余域（codomain）の型と（ここでは）呼ぶ

高階関数

域の型または余域の型が関数型であるような関数型の値

```
(a -> b) -> c
--------   ---
   域       余域

a -> (b -> c)
--   --------
域      余域
```

型構成子 `->` は右結合なので、後者の場合は

```
a -> b -> c
```

と書くことが多い。

```
f :: a -> b -> c
```

で

```
uncurry f :: (a,b) -> c
```

を想起することがよくあるが、

```
f :: a -> (b -> c)
```

も想起できると、見え方がひろがる。たとえば、二項演算子は高階関数！？

型のお気持

- σ、τ が型なら $\sigma \to \tau$ も型
- σ、τ が型なら (σ, τ) も型
- σ が型なら $[\sigma]$ も型
- T が型構成子（:: Type -> Type）、σ が型なら $\mathrm{T}\ \sigma$ は型
- **正しく型付けされた式は値を表示する**

命令

命令は命令型（instruction type）の値

`IO a` は**実行すると** `a` 型の値にが得られるはずの命令（の値）の型である。

プログラムで命令を表示することはできる。

N.B.「命令型」は、この資料独自の用語法です。

Haskellプログラミング

Haskellプログラミング ＝ 関数プログラミング ＋ 命令プログラミング

Haskellでは、命令的に書ける。そもそも、`main` は関数ではなく命令（2回目）

```
main :: IO ()
main = do
    { inp <- getContents
    ; let xs = map (read @Int) $ words inp
    ; let ys = scanl1 (+) xs
    ; forM_ ys print
    }
```

`interact`

「関数」を「命令」にしてくれる `Prelude` （Haskellの標準モジュール）関数

```
interact :: (String -> String) -> IO ()
```

があって

```
main :: IO ()
main = interact someFunc
```

と書けるので、**関数**

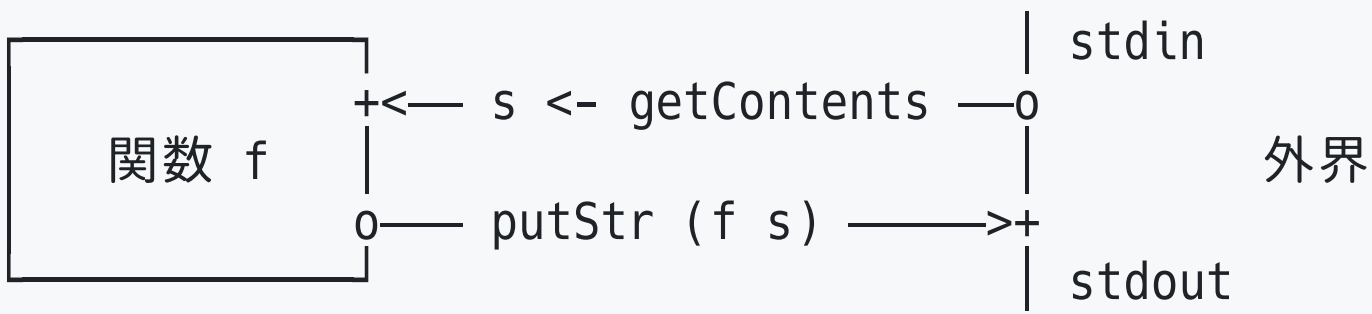
```
someFunc :: String -> String
```

関数プログラミングができて嬉しいよ。

interact

```
interact :: (String -> String) -> IO ()
interact f = do s <- getConetnts
                putStr (f s)

getContents :: IO String         -- 標準入力にある文字列全体をえる命令
putStr      :: String -> IO ()   -- 指定した文字列を標準出力に置く命令
```



お題（１）： AtCoder提出プログラムの雛形

Welcome to AtCoder

> **入力**
>
> 入力は以下の形式で与えられる
>
> ```
> a
> b c
> s
> ```
>
> **出力**
>
> $a + b + c$ と s を空白区切りで1行に出力せよ。

命令的

命令的に考えてしまうと抽象化しにくい？ボトムアップに考えてしまう？

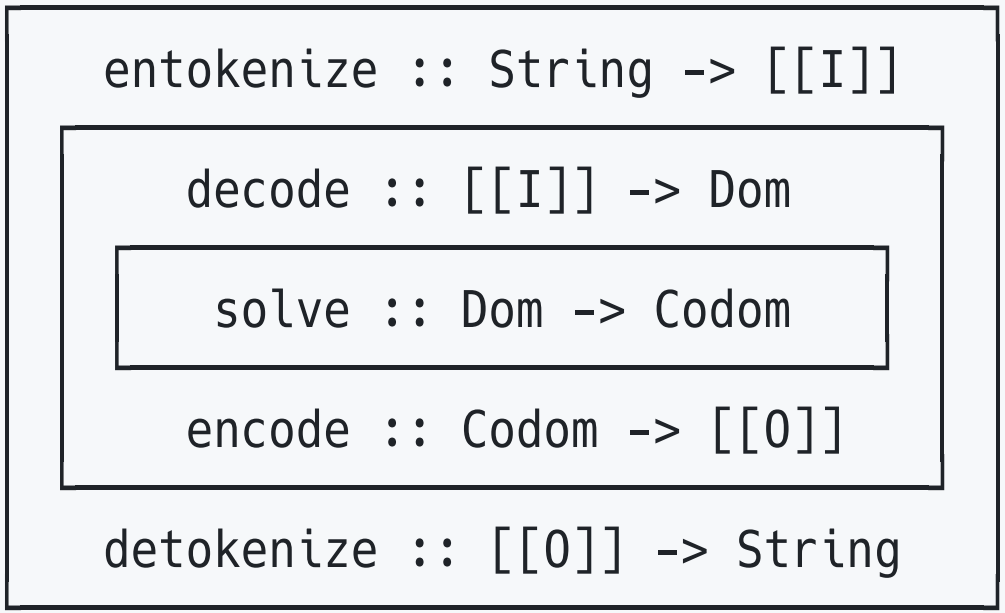
```
main :: IO ()
main = do
  { a       <- read  <$> getLine
  ; [b, c] <- map read . words <$> getLine
  ; s       <- getLine
  ; solve [a,b,c] s
  }

solve :: [Int] -> String -> IO ()
solve as s = do
  { putStr (show (sum as))
  ; putStr " "
  ; putStrLn s
  }
```

関数的

トップダウンに考えられる？

```
main :: IO ()
main = interact (detokenize . (encode . solve . decode) . entokenize)
```



雛形の固定部

- 入力トークン `I`、出力トークン `O` それぞれ、高々 `Int`、 `String` の2つ
- `entokenize`、 `detokenize` を `AsToken` クラスのメソッドとする
- `Int`、 `String` をそれぞれ `AsToken` クラスのインスタンスとして宣言

```
class AsToken a where
  entokenize :: String -> [[a]]
  detokenize :: [[a]] -> String

instance AsToken String where
  entokenize = map words . lines
  detokenize = unlines . map unwords

instance AsToken Int where
  entokenize = map (map read . words) . lines
  detokenize = unlines . map (unwords . map show)
```

雛形利用時に定義

定義右辺を利用時に変更（以下は変更後）

```
type I = String                 -- 入力が「すべて」整数なら Int でなければ String
type O = String                 -- 出力が「すべて」整数なら Int でなければ String

type Dom   = ([Int], I)         -- 問題を解くためのデータ一式の型
type Codom = (Int, O)           -- 問題の解答データ一式の型

solve :: Dom -> Codom
solve = \ case
  (as,s) -> (sum as, s)         -- 選択肢の「->」の右辺をプログラムする（本丸）
```

(つづく)

(つづき)

```
decode :: [[I]] -> Dom
decode = \ case
  [a]:[b,c]:[s]:_ -> (read @Int <$> [a,b,c], s) -- パターン照合
  _               -> error "invalid input"

encode :: Codom -> [[O]]
encode = \ case
  (a,s) -> [[show a, s]]                        -- パターン照合
```

お題（２）： Wordleパズルのチートプログラム

```
$ wordle
? imply ybbyb   # imply 推量文字列、ybbyb（黄黒黒黄黒） 照合パターン
...

? their bbyyb
bilge
elide
glide

? :quit         # :quit プログラムを抜けるコマンド
$
```

プロンプトに対して推量文字列と照合パターンを入力すると候補を絞り込む

全体構造

```
main :: IO ()
main =  interactWithPrompt "? " ":quit" (wordle dict)
```

`interactWithPromt` はカスタム版 `interact`

```
interactWithPrompt :: String -> String -> ([String] -> [String]) -> IO ()
```

`wordle`の構造

```
wordle :: Dict -> ([String] -> [String])
wordle dict = mapMaybe output . eval isFinal step . initial dict
```

```
type Transition a = MachineState a -> MachineState a  -- 遷移の型
initial  :: Dict -> MachineState a
eval     :: (MachineState a -> Bool)                  -- 終了判定
         -> Transition a                              -- 遷移
         -> (MachineState a -> [MachineState a])      -- 遷移系列生成
mapMaybe :: (a -> Maybe b) -> ([a] -> [b])
```

```
type Dict = [String]
dict    :: Dict
isFinal :: MachineState Dict -> Bool
step    :: Trandition Dict
```

遷移する状態

```haskell
data MachineState a
    = MachineState
    { inChan     :: [String]       -- 以降の全入力列
    , output     :: Maybe String   -- 現在の出力
    , innerState :: a              -- 現在の内部状態
    }
type VMState = MachineState [String]
```

遷移列の生成

```haskell
eval :: (MachineState a -> Bool)                -- 終了判定
     -> Transition a                            -- 遷移関数 MachineState a -> MachineState a
     -> (MachineState a -> [MachineState a]) -- 遷移系列生成

eval isFinal_ step_ = takeWhilePlus1 isFinal_ . iterate step_

step :: Trandition Dict
step state = case state of
    MachineState { inChan = i : is
                 , innerState = dict
                 } -> state { inChan = is
                            , output = Just $ unlines dict'
                            , innerState = dict'
                            }
                       where
                           guess:pattern:_ = words i
                           dict' = buildFilter guess pattern dict
```

```
{ inChan = is0@(i1 : _) , outPut = Nothing     , innerState = dict0 }
```

↓ `drop 1 is0` `bildFilter g0 p0 dict0`

```
{ inChan = is1@(i2 : _) , outPut = Just dict1' , innerState = dict1 }
```

↓ `drop 1 is1` `bildFilter g1 p1 dict0`

```
{ inChan = []           , outPut = Just dictN' , innerState = dictN }
```

モジュラリティ

$$f = f_0 \circ f_1 \circ \cdots \circ f_n$$

- **複数の独立したコンポーネントの合成による構成**

抽象化

多相型、多相関数による抽象化

汎用ライブラリとして有用、ライブラリ使用者は、

- 内部状態の型
- 初期化関数、`initial`、遷移関数`step`
    - 場合により、終了判定述語`isFinal`

を具体化するだけで対話プログラムを構成できる

お題（３） Interactive Sorting

> 最初のN個の大文字でラベルの付いたN個のボールがあります。 どの二つのボールの重さも異なります。
>
> あなたはQ回クエリを質問することができます。各クエリでは、二つのボールの重さを比べることができます。 ボールを軽い順にソートしてください。
>
> **制約**
>
> $$(N, Q) = (26, 1000), (26, 100), (5, 7) \quad \text{のいずれかである}$$

(つづく)

入出力

最初に、標準入力からNとQが以下の形式で与えられる:

```
N Q
```

次に、あなたはQ回以下クエリを質問する。各クエリは、標準出力に以下の形式で出力されなければならない:

```
? c1 c2
```

ここでc_1とc_2は異なる最初のN個の大文字のどれかでなければならない。

(つづく)

次に、クエリの答えが標準入力から以下の形式で与えられる:

ans

ここで*ans*は`<`または`>`である。`<`のときはc_2のほうが重く、`>`のときはc_1のほうが重い。最後に、答えを以下の形式で出力しなければならない：

! *ans*

ここで*ans*はN文字の文字列であり、最初のN個の大文字を軽い順に並べたものでなければならない。

状態遷移系？

お題（２）のライブラリは使えそうか？

`MachineState` の内部状態および `step` をどのように具体化するか？

N.B.: 実は、$(5, 7)$ の場合は、以下のように直截に書ける方法が有効。

- 最適質問木を予め構成（ `unfoldTree :: (a -> (b, [a])) -> a -> Tree b` を使う）
- 内部状態 `innerState` を最適質問木に設定
- `step` ごとに `innerState` のルートラベルを出力
- 同時に、 `inChan` の先頭にしたがって `innerState` を左または右の子ノードで更新

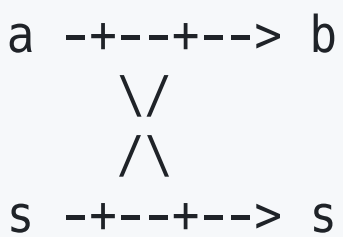
`sortBy'` 関数

```haskell
sortBy' :: (a -> a -> Ordering) -> [a] -> [a]
sortBy' cmp xs = case ys of
    [] -> zs
    _  -> merge $ (sortBy' cmp ys, sortBy' cmp zs)
        where
            merge (aas@(a:as), bbs@(b:bs)) = mrg $ cmp a b
                where
                    mrg = \ case
                        GT -> b : merge (aas, bs)
                        _  -> a : merge (as, bbs)
            merge ([], bs) = bs
            merge (as, []) = as
    where
        (ys,zs) = splitAt (length xs `div` 2) xs
```

状態遷移を伴う関数

状態遷移 `s -> s` と、関数 `a -> b` をあわせた、状態遷移を伴う関数

```
f :: (a,s) -> (b,s)
```



カリー化すると

```
                                        (St s) b
                                        +------+
f' :: a -> (s -> (b,s))          a -->+--+--+--> b
           ------------                 |  |  |
               余域                     +  +--+--> s
                                        |  |  |
                                        +--↑--+
                                           s
```

余域を抽象化して

```
f' :: a -> (St s) b
```

関数をモナド上に拡張する

前述の抽象化において、

- `St s` は「モナド」という計算機構の具体例になる
- `St` は「状態遷移系」という計算機構の具体例になる

そこで、`msortBy` をモナド上で使えるように拡張してから、具体的な状態遷移を組込むことを考える。

```
msortBy :: (a -> a -> Ordering) -> [a] -> [a]
```

を

```
msortByM :: Monad m => (a -> a -> m Ordering) -> [a] -> m [a]
```

に拡張する。計算機構の拡張は、関数適用の拡張を通じておこなう。

関数適用の拡張

```
($)   :: (a -> b) -> (a -> b)                    -- 関数適用

(<$>) :: Functor f                               -- (<$>) = fmap
      => (a -> b) -> (f a -> f b)
(<*>) :: Applicative af                          -- Applicative クラスの method
      => af (a -> b) -> (af a -> af b)
(=<<) :: Monad m                                 -- (=<<) = flip (>>=)
      => (a -> m b) -> (m a -> m b)
```

素朴版再掲

関数適用を強調するために、関数適用演算子を追加

```haskell
sortBy' :: (a -> a -> Ordering) -> [a] -> [a]
sortBy' cmp xs = case ys of
    [] -> zs
    _  -> merge $ ((,) $ (sortBy' cmp ys)) $ sortBy' cmp zs
        where
            merge (aas@(a:as), bbs@(b:bs)) = mrg $ cmp a b
                where
                    mrg = \ case
                        GT -> (b :) $ merge (aas, bs)
                        _  -> (a :) $ merge (as, bbs)
            merge ([], bs) = bs
            merge (as, []) = as
    where
        (ys,zs) = splitAt (length xs `div` 2) xs
```

モナド版

```haskell
msortBy :: Monad m => (a -> a -> m Ordering) -> [a] -> m [a]
msortBy cmp xs = case ys of
    [] -> pure zs
    _  -> merge =<< (,) <$> msortBy cmp ys <*> msortBy cmp zs
     where
         merge (aas@(a:as), bbs@(b:bs)) = mrg =<< cmp a b
             where
                 mrg = \ case
                     GT -> (b :) <$> merge (aas, bs)
                     _  -> (a :) <$> merge (as, bbs)
         merge ([], bs) = return bs
         merge (as, []) = return as
    where
        (ys,zs) = splitAt (length xs `div` 2) xs
```

状態遷移系

状態遷移系のモナド計算を実現するためには、

- モナド計算： `St s` を `Monad` クラスのインスタンスとして宣言
- 状態遷移系： `St` を `MonadState` クラスのインスタンスとして宣言

でよいのであるが、出力トークンの蓄積を別立てで用意し、遷移する状態は入力トークン列のみにしておくのが簡便である。そこで、Control.Monad.RWS のRWSモナドの仕組みを流用する。

```
type St w s = RWS () w s
```

これで、モナド計算系機構の以下の演算が追加で手にはいったことになる

- 現在の状態の取得と設定

```
get :: St w s a -> St w s s
put :: s -> St w s ()
```

- 現在の出力

```
tell :: w -> St w s ()
```

- 初期状態を与えて状態遷移の実現

```
runSt  :: St w s a -> s -> (a, s, w)
runSt st s = runRWS st () s
evalSt :: St w s a -> s -> (a, w)
evalSt st s = evalRWS st () s
execSt :: St w s a -> s -> (s, w)
execSt st s = execRWS st () s
```

全体構造

以下のような構造にできる

```
main :: IO ()
main = interact solve

solve :: String -> String
solve = unlines . snd . evalSt action . lines

type Dio = St [String] [String]
```

テストセット２対応

```
action :: Dio ()
action = answer =<< msortBy cmp . flip take ['A' .. 'Z'] =<< getBalls

getBalls :: Dio Int
getBalls = read @Int . takeWhile isDigit <$> getLn

answer :: String -> Dio ()
answer ans = tell (unwords ["!", ans])

getLn :: Dio String
getLn = uncurry (>>) . (put . drop 1 &&& return . (!! 0)) =<< get
```

`cmp :: Char -> Char -> Dio Ordering`

```
cmp :: Char -> Char -> Dio Ordering
cmp x y =  tell [unwords ["?", [x], [y]]]
        >> (toOrdering <$> getLn)

toOrdering :: String -> Ordering
toOrdering = \ case
    '<':_ -> LT
    _     -> GT
```

あれれっ？

気づいていた方も（多く）いたと思いますが。。。

```
msortBy :: Monad m => (a -> a -> m Ordering) -> [a] -> m [a]
```

が書けてるのなら、 ↓ でよいのでは。。。

どんでん返し！？

```haskell
main :: IO ()
main = solve

solve :: IO ()
solve = answer =<< msortBy cmp . flip take ['A' .. 'Z'] =<< getBalls

answer :: String -> IO ()
answer ans = putStrLn (unwords ["!", ans])

getBall :: IO Int
getBall = read @Int . (!! 0) . unwords <$> getLine

cmp :: Char -> Char -> IO Ordering
cmp x y =  putStrLn (unwords ["?",[x],[y]])
        >> (toOrdering <$> getLine)
```

まとめ

「命令的記述を1箇所に小さくまとめて、関数的記述に集中したい」

- `interact` を使えば、標準入出力しかない I/O は完全に分離可能である
- I/O は排除できるが、「明示的データ依存関係のない順序依存」は排除できない
- 「明示的データ依存関係のない順序依存」がコードに現れると「命令的記述」に見える

お題	Ｉ／Ｏの特徴	プログラミングの要点
1	まとめて入力、まとめて出力	処理の共通化とパターン照合の活用
2	処理とI/Oが交互に出現	I/Oを分離をしたプログラミング
3	DAG形処理と1次元形I/Oの混合	モナド計算に1次元のI/Oをつなぐ