

命令プログラムを関数プログラミングぽく書く

@nobsun (a.k.a. 山下伸夫)

関数プログラミングとは

関数プログラミング

- 「プログラム = 関数」と考えてプログラミングするスタイル
- 「プログラム = 関数」を表現できる言語で思考するスタイル

Haskellプログラミングは「関数プログラミング」か

- Haskellは「汎用の純粋関数型言語」（Haskell 2010 Language Report）
- Haskellは「最も美しい命令型言語」（S. Peyton Jones: Beautiful Concurrency）

関数プログラミングぽさは気分の問題

プログラミングの気分（個人的感想）

	命令ぽい	関数ぽい
計算順序	出現順	データ依存順
思考方向	ボトムアップ	トップダウン
結果伝搬	暗黙的	明示的
アクセス	ランダム	シーケンシャル
計算態度	オンプレミス	オンデマンド
資源消費	節約的几帳面	富豪的無頓着

どこまで「関数ぽく」書けるか

命令プログラミングに向いてそうなもの（個人的思い込み）

- 命令プログラムの実行シミュレーション

カーニハンのトイ・マシン・シミュレーター

Toy Machine Simulator

- アキュムレータ 1つ
- 命令とデータを格納するメモリ
- キーボード入力
- 端末表示

	命令		意味
	GET		キーボードからアキュムレータに数値を読み込む
	PRINT		アキュムレータの内容を出力欄に表示
	LOAD	<i>Val</i>	アキュムレータに値（またはメモリ番地の内容）を設定
	STORE	<i>M</i>	アキュムレータの内容の写しをメモリ番地 <i>M</i> に格納
	ADD	<i>Val</i>	アキュムレータの内容に値（またはメモリ番地の内容）を足し込む
	SUB	<i>Val</i>	アキュムレータの内容から値（またはメモリ番地の内容）を引く
	GOTO	<i>L</i>	ラベル <i>L</i> が付けられた命令に飛ぶ
	IFPOS	<i>L</i>	もしアキュムレータの内容が0または正であればラベル <i>L</i> に飛ぶ
	IFZERO	<i>L</i>	もしアキュムレータの内容がちょうど0であればラベル <i>L</i> に飛ぶ
	STOP		実行を停止する
<i>M</i>	<i>Num</i>		プログラムを起動する前に、このメモリ位置（番地 <i>M</i> ）を数値 <i>Num</i> に設定する

トイ・マシンのプログラム例：数値入力の加算

```
Top GET
    IFZERO Bot
    ADD Sum
    STORE Sum
    GOTO Top
Bot  LOAD Sum
    PRINT
    STOP
Sum 0
```


シミュレーターは入出力をともなうプログラム

```
toysim :: String -> IO ()  
toysim = interact . wrap . toy  
  
type Interaction = String -> String  
  
wrap :: Toy -> Interaction  
toy  :: String -> Toy
```

トイ・マシンの実行をシミュレートする関数

トイ・マシンのシミュレートする関数 `toy :: String -> [String] -> [String]`

- ソースコード
 - ソースプログラムの型 `String`
- `GET` 命令と `PRINT` 命令の実行をシミュレートする
 - 入力の型 `[String]`
 - 出力の型 `[String]`

```
type Toy = [String] -> [String]

toy code = filter (not . null) . map output
           . eval . initState (loadProg code)
  where
    output (_,_,_,out) = out

wrap f = unlines . f . lines
```

トイ・マシンの実行

関数として考えるなら

- トイ・マシンの命令実行は、状態遷移関数(`ToyState -> ToyState`)の適用
- トイ・マシンのプログラム（命令列）の実行は、初期状態から最終状態にいたる状態遷移列の生成

```
eval :: ToyState -> [ToyState]
eval state = state : followings
  where
    followings
      | final state = []
      | otherwise   = eval (step state)
```

シミュレーター動かしてみる

```
main :: IO ()  
main = toysim sampleCode
```

トイ用プログラムをファイルから読む

```
import System.Environment  
  
main :: IO ()  
main = toysim =<< readFile . head =<< getArgs
```

入力プロンプトを出せるか

```
iGet :: Operand -> Instruction
iGet _ ((sz,mem),acc,ins,_)
    = ((sz, tail mem), read (head ins), tail ins, "")
```

I/O の順序とデータの依存関係

step は fetch、decode、execute の1サイクル分

```
step :: ToyState -> ToyState
step state = execute (decode (fetch state)) state

fetch :: ToyState -> Code
decode :: Code -> Instruction
execute :: Instruction -> ToyState -> ToyState
```


まとめ

- I/O の順序とデータ依存関係が独立していると命令ぽい
- I/O の順序をデータ依存関係で決めてよいなら関数ぽく書ける

ToyState 実装

```
type ToyState = (Memory, Accumulator, [Input], Output)

type Memory = (Int, [(Label, Content)])

type Input = String
type Accumulator = Int
type Output = String

type Instruction = ToyState -> ToyState
```

プログラムローダー `loadProg :: String -> Memory` の実装

```
loadProg :: String -> Memory
loadProg src = (,) . length <*> cycle . map conv $ lines $ map toUpper src
```

状態初期化関数 `initState`

```
initState :: Memory -> [Input] -> ToyState
initState mem input = (mem, 0, input, "")
```

`fetch :: ToyState -> Code` の実装

```
fetch ((_,mem),_,_,_) = case mem of  
  (_,Code code):_ -> code
```

```
execute :: Instruction -> ToyState -> ToyState
```

```
execute = id
```

decode :: Code -> Instruction の実装

```
decode (ope, opd)
  = (case ope of
    GET    -> iGet
    PRINT  -> iPrint
    LOAD   -> iLoad
    STORE  -> iStore
    ADD    -> iAdd
    SUB    -> iSub
    GOTO   -> iGoto
    IFPOS  -> iIfpos
    IFZERO -> iIfzero
    STOP   -> iStop
  ) opd
```

iPrint :: Operand -> Instruction`の実装

```
iPrint :: Operand -> Instruction
iPrint _ ((sz,mem),acc,ins,_)
    = ((sz, tail mem), acc, ins, show acc)
```


メモリ [(Label, Content)] の実装

```
type Label = String

data Content
  = Code Code
  | Data Data

type Code = (Operator, Operand)

type Data = Int
```

Operator 実装

```
data Operator
  = GET
  | PRINT
  | STOP
  | LOAD
  | STORE
  | ADD
  | SUB
  | GOTO
  | IFPOS
  | IFZERO
  deriving (Eq, Ord, Enum, Show, Read)
```

Operand 実装

```
data Operand
  = None
  | Number Int
  | Label Label
  deriving (Eq, Show)
```

iLoad, iStore :: Operand -> Instruction の実装

```
iLoad, iStore :: Operand -> Instruction
```

```
iLoad opd ((sz,mem),_,ins,_)  
    = ((sz, tail mem), value (sz,mem) opd, ins, "")
```

```
iStore opd (m,acc,ins,_)  
    = next (update opd acc m, acc, ins, "")
```

iAdd, iSub :: Operand -> Instruction の実装

```
iAdd, iSub :: Operand -> Instruction
```

```
iAdd opd ((sz,mem),acc,ins,_)  
    = ((sz, tail mem), acc + value (sz, mem) opd, ins, "")
```

```
iSub opd ((sz,mem),acc,ins,_)  
    = ((sz, tail mem), acc - value (sz, mem) opd, ins, "")
```

iGoto, iIfpos, iIfzero :: Operand -> Instruction の実装

```
iGoto, iIfpos, iIfzero :: Operand -> Instruction
iGoto (Label l) ((sz,mem),acc,ins,_)
  = ((sz, mem'), acc, ins, "")
  where
    mem' = dropWhile ((l /=) . fst) mem

iIfpos lab@(Label _) st@(_,acc,_,_) = st'
  where
    st' = if acc >= 0 then iGoto lab st else next st

iIfzero lab@(Label l) st@(_,acc,_,_) = st'
  where
    st' = if acc == 0 then iGoto lab st else next st
```

iStop :: Operand -> Instruction の実装

```
iStop :: Operand -> Instruction
iStop _ ((_,mem),acc,ins,_)
    = ((0, mem), acc, ins, "\nstopped")
```

STOP の実行によって、メモリサイズを 0 に設定し、これを終了判定 final で判定する

```
final :: ToyState -> Bool
final st = case st of
    ((0,_),_,_,_) -> True
    _              -> False
```

補助関数 `next` の定義

```
next :: ToyState -> ToyState
next ((sz,mem),acc,ins,_) = ((sz, tail mem), acc, ins, "")
```


補助関数 `update` の定義

```
update :: Operand -> Data -> Memory -> Memory
update (Label m) val (sz, mem)
  = case break ((m ==) . fst) (take sz mem) of
    (xs, _:ys) -> (sz, cycle (xs ++ (m, Data val) : ys))
    _           -> error (m ++ ": not exist")
```

補助関数 `value` の定義

```
value :: Memory -> Operand -> Data
value (sz,mem) opd
  = case opd of
      Number num -> num
      Label lab  -> case dropWhile ((lab /=) . fst) (take sz mem) of
          (_,Data d):_ -> d
          _             -> error (lab ++ ": not exist")
```

補助関数 `conv :: String -> (Label, Content)`

```
conv :: String -> (Label, Content)
conv line = case line of
  c:cs -> if isSpace c
    then (" ", toContent (words line))
    else case words line of
      label:content -> (label, toContent content)
```

補助関数 `toContent :: [String] -> Content`

```
toContent :: [String] -> Content
toContent ss = case ss of
  [cs] | all isDigit cs -> Data (read cs)
        | otherwise      -> Code (read cs, None)
  [ope,cs]               -> Code (read ope, opd)
  where
    opd = if all isDigit cs
          then Number (read cs)
          else Label cs
```