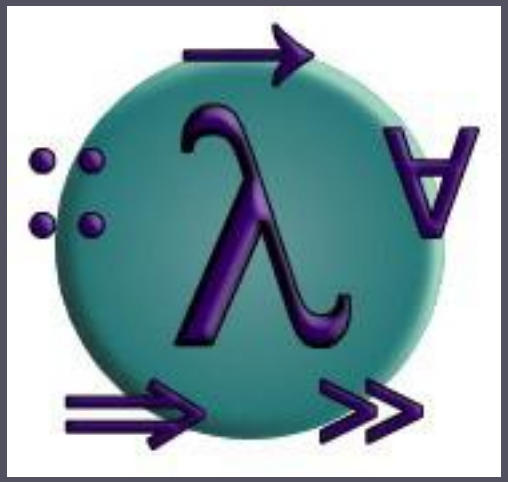
PROGRAMMING IN HASKELL
プログラミングHaskell



練習問題

愛知県立大学 情報科学部 計算機言語論(山本晋一郎・大久保弘崇、2011年)講義資料
オリジナルは http://www.cs.nott.ac.uk/~gmh/book.html を参照のこと

練習問題(3章、問題2)

- swap の型は (Int, Int) → (Int, Int) でも、あるいは (a, a) → (a, a) でも成功するが、どちらにしたほうが良いのですか?
  - swap :: (a, b) → (b, a) にすべき
    - 多くの型で動作する方が有用
    - swap (1, 'a') や swap (True, [0, 1, 2]) も動作する

練習問題(3章、問題2)

- 関数 twice の型がよく分からなかった
  - twice :: (a → a) → a → a
    twice f x = f (f x)
  - twice は 2 引数関数
    - 第 1 引数は a 型を受取り a 型を返す関数
    - 第 2 引数は a 型
    - 戻り値は a 型
  - 本体の部分式 f x より、第 1 引数で与えられた関数 f を第 2 引数 x に適用できることが分かる
    - 一般に、関数 f :: a → b は、a 型の値に適用可能で、結果は b 型の値となる
  - その値 f x にもう一度 f が適用されている

練習問題(3章、質問3、その2)

- Twice の型を自分で考えてみる
  - 仮に twice :: (a → b) → c → d としてみる
    - 第 1 引数は関数なので a → b とし、第 2 引数は c としておく
  - f x より、a = c となり (a → b) → a → d
  - f (f x) より、a = b となり (a → a) → a → d
  - f (f x) :: d は f の戻り値の型なので a = d となり (a → a) → a → a

練習問題(3章、問題4)

- 関数の等しさとは?
  - 同じ入力に対して同じ出力を返すか否かで判定
- 2つの関数 f :: Int → Int と g :: Int → Int に対して、全ての Int の値(少なくとも 30 bit なので, 10 億個の値)を比較する必要あり
  - The finite-precision integer type Int covers at least the range [-2^29, 2^29 - 1]
- あるいは帰納法を用いる(自動化困難)

練習問題(5章、問題2)

- length と同じように、replicate をリスト内包表記を用いて定義せよ
  - `rep n x = [x | _ ← [1..n]]`
  - “関数 length と同じように” の意味が分からなかった
    - `length xs = sum [1 | _ ← xs]`
    - リスト内包表記を繰り返しと捉える(生成される値は無視)
  - 個数に 0 を与えられたときどうすれば良いのか?
    -
```
rep 3 True = [True, True, True]
rep 2 True = [True, True]
rep 1 True = [True]
rep 0 True = []
```

0 < n と制限するよりも、適用範囲が広く、整合性があり美しい

練習問題(2章、問題5)

- [(x, y) | x ← [1,2,3], y ← [4,5,6]] を 1 つの生成器を持つ 2 つのリスト内包表記で表現せよ
  - concat [[(x , y) | y ← [4, 5, 6]] | x ← [1, 2, 3]]
  - 内包表記 2 つというのが分からない
    - “一方のリスト内包表記を他方の中に入れ” とは、[e | x ← [1,2,3]] か [e | y ← [4,5,6]] の形で e の部分にもう 1 つのリスト内包表記を入れて表現せよということ
    - リスト内包表記の結果はリストなので、上記式はリストのリストとなり、concat が必要となる
      - concat :: [[a]] → [a]
        concat [[1,2,3], [4,5], [6]] = [1,2,3,4,5,6]
    - e の部分に任意の式が書ける(式の組合わせに制約が少ない)ことが Haskell の強み(最初からリスト内包表記の入れ子を思いつかないだろうが)

練習問題(6章、問題3)

良い質問! 自分で導けるようにしましょう
なぜ product [] = 1 なのか問題と同じ

- and は [] に True と False のどちらを返せば良いのか?
  - ```
    and [x, y, z] = x && y && z
    and [x, y]    = x && y
    and [x]       = x
    and []        = ???
    ```
  - 任意の論理式 x に対して、True && x = x が成り立つので
    以下のように考える
    ```
    and [x, y, z] = True && x && y && z
    and [x, y]    = True && x && y
    and [x]       = True && x
    and []        = True
    ```
  - 結局、and [x] = and [True, x] より、以下が成立
    and [] = and [True] = and [True, True] = … = True
    ```

練習問題(6章、追加問題1)

- product に関する再帰の秘訣(6.6節、p.66)において、product [] = 1 と定義する理由を、"1 は乗法の単位元だから" としている。これを解説せよ。
  - 左単位元と見る
    - prod [a, b, c] = 1 * a * b * c
      prod [a, b] = 1 * a * b
      prod [a] = 1 * a
      prod [] = 1
  - 右単位元と見る
    - prod [c, b, a] = c * b * a * 1
      prod [b, a] = b * a * 1
      prod [a] = a * 1
      prod [] = 1
  - 結局、prod [x] = prod [1, x] より、以下が成立
    prod [] = prod [1] = prod [1, 1] = … = 1

練習問題(7章、問題4)

- 関数 foldl を用いて 10 進表記を表すリストを整数値に変換する dec2int を定義せよ
- MSD (Most Significant Digit、最上位桁)は先頭
- 例: dec2int [1,2,3] = 123
- ```
  dec2int =
    foldl (λac y → ac * 10 + y) 0
  ```
  ```

dec2int の λac x → ac * 10 + x はどうやって求める?

- ```
  dec2int [1,2,3]
  = foldl f v (1 : (2 : (3 : [])))
  = (((v `f` 1) `f` 2) `f` 3
    where v = ???
          f x y = ???
  ```
- [1,2,3] から 123 を手で計算して比べてみる

  ```
  ((v * 10 + 1) * 10 + 2) * 10 + 3
  = v * 1000 + 1 * 100 + 2 * 10 + 3
  = v * 1000 + 123
  ```

  よって v = 0
- f ac x は、前方のリストに対する値を ac に、要素を x に取って、ac を 10 倍して x を加えて返せば良い

  ```
  f ac x = ac * 10 + x
  ```
  ```

練習問題(7章、問題4)

- 再帰を用いた定義も書いてくれた人がいた

```
dec2int [] = 0
dec2int (x:xs) =
  10^((length (x:xs))-1) * x + dec2int xs
```

  - length が何度も呼ばれるので効率が悪い
  - length がどのような引数で何回呼ばれるか考えてみると良い

foldl 版よりも先に再帰版を求めることは重要

- 再帰版の正解

```
dec2int xs = f' 0 xs
f' ac []     = ac
f' ac (x:xs) = f' (ac * 10 + x) xs
```

練習問題(7章、問題4)

dec2int [1,2,3] は dec2int [0,...,0,1,2,3] と等価

- 再帰版の動作

```
   dec2int [1,2,3]
= f' 0 [1,2,3]
= f' (0 * 10 + 1) [2,3]
= f' 1 [2,3]
= f' (1 * 10 + 2) [3]
= f' 12 [3]
= f' (12 * 10 + 3) []
= f' 123 []
= 123
```

練習問題(7章、問題4)

- foldl 版の動作

```
   dec2int [1,2,3]
= foldl (λx y → x * 10 + y) 0 [1,2,3]
= foldl (…) (0 * 10 + 1) [2,3]
= foldl (…) 1 [2,3]
= foldl (…) (1 * 10 + 2) [3]
= foldl (…) 12 [3]
= foldl (…) (12 * 10 + 3) []
= foldl (…) 123 []
= 123
```

練習問題(7章、問題4)

- 再帰版と foldl 版との関係
- ```
  dec2int xs = f' 0 xs
  f' ac []     = ac
  f' ac (x:xs) = f' (ac * 10 + x) xs
  ```

```
f xs = f' v xs
f' ac []     = ac
f' ac (x:xs) = f' (ac ⊕ x) xs

f xs     = foldl (⊕) v xs
```

- ```
  dec2int = foldl (λac x → ac * 10 + x) 0
  ```

練習問題(7章、問題4、再帰版と foldl 版との関係)

- 再帰版

```
dec2int xs = f' 0 xs
f' ac []     = ac
f' ac (x:xs) = f' (ac * 10 + x) xs
```

- foldl 版

```
f xs = f' v xs
f' ac []     = ac
f' ac (x:xs) = f' (ac ⊕ x) xs
```

このような f に対して f xs = foldl (⊕) v xs が成り立つ

- 両者を見比べて

```
dec2int = foldl (λac x → ac * 10 + x) 0
```

練習問題(8章、その他)

- 教科書で説明されたパーサーの部品を使うためには、スクリプトの先頭に `import Parsing` が必要
  - Parsing.lhs のダウンロード方法は、動作確認のスライドを参照のこと
- 問題 3, 4 の“構文規則の何番目”という表現が理解できない
  - 教科書には 3 種類の構文規則が表れている
    - (素朴な)最初の構文規則 (p.99)
    - 優先度を考慮した 2 番目の構文規則(p.100 上)
    - 優先度と右結合性を考慮した 3 番目の構文規則 (p.100 下、完成版)
    - スライドでは個別に説明している