第二回

目次

(数式を一部利用しているのでレンダリングがもしできなかったらこちらから見てください。)

Q:エラー処理はどうする？

エラー処理の話はのちにやります。出力がstringなる関数なら、エラーメッセージを返す処理などの実装は出来ます(やってる方もいたようです)。

ガード記法とパターンマッチについて

ガード記法で定義した関数において、定数に対して"=="を条件式として使用するのはあまり **Haskellらしくない** そうです。以下にHaskellらしくない例を載せます。

```
func x
    | x == 1 = --何らかの処理
    | x == 2 = --何らかの処理
```

Haskellらしい例を直すと以下のようになります。

```
func 1 = --何らかの処理
func 2 = --何らかの処理
```

P56~P57

聞き間違いなら申し訳ないが、Haskellの少数型はfloatしかないと言っていた。検証結果

使用時の丸めで誤差が出るので注意。オーバーフローにも注意。IEEEの規格で浮動小数点の規格は決まっている。

P58

Prelude モジュールで定義された、Float関連の見慣れたcalculatorが並ぶ。

課題は3.20(P59)

P60

HaskellはPythonと似てて、ブロック(構文)をインデントで判断する。この仕組みには、オフサイドという名前がついている。決まった線(インデント)より出てしまうと次のブロック(構文)になるという意味からこの名前になっている。

P62

Intなどの型名やTrue/Falseなどのコンストラクタは最初が大文字。変数は最初は小文字(命名規則なので大文字にするとエラーが出る。)。ただし命名規則を守っても、予約語があるので注意。(P63)

再定義したい場合は、前回の講義で説明したようにすればいい。(P53)

Haskellは、Unicode文字の説明規格に基づいて構築されており、ASCII規格以外のフォントからの記号を許可している。

P63

演算子は記号(`!#$%&*+./<=>?@\^|-~`)で作る。()をつければ計算順序を決めれる。しかし、()で計算順序をいちいち指定していては可視性が下がる。

```
(((4+8) *3) +2)
```

4-2-1は左結合と定義されている。(同じ優先順序なら左からやる)

```
(4-2)-1=2-1=1 --左結合
```

優先度(結合力、番号で管理されている)や結合の向きの詳細はP551。

優先度	演算子
9	`!!` , `.`
8	`^` , `^^` , `**`
7	`*` , `/` , `div` , `mod` , `rem` , `quot`
6	`+` , `-`
5	`:` , `++`
4	`==` , `/=` , `<` , `<=` , `>` , `>=` , `elem` , `notElem`
3	`&&`
2	`||`
1	`>>` , `>>=`
0	`$` , `$!` , `seq`

P64

```
2^3^2
```

これは右結合。

$$2^{3^2} = 2^{(3^2)}$$

数学と同じ。

まずは関数が実行されてから演算子が実行される。なので

```
f n + 1
```

は

```
(f n) + 1
```

と実行されてしまうので関数は疑わしければかっこをつける。

P65

演算子は(+)などとすると関数となる。

```
x = y + z
x = (+) y z
```

逆に関数を'func'とすると演算子になる。

```
x = func y  z
x = y  'func' z
```

Do-it-youself operators。これは新しい演算子を作る機能。演算子に使える記号が決まってる。結構色んな記号が使える(ユニコードのやつ？)。教科書では `&&&` なる演算子を作っている。さらに優先度や結合の向きを定義できる。

```
(&&&) :: Integer -> Integer -> Integer
x &&& y
```

```
      | x > y     = y
      | otherwise = x

a = 3 &&& 2 -- a=2になる
```

4章~~

プログラミングの書き方が一つのテーマ。haskellは関数が中心。例えば、3つの数字のなかで、真ん中の数字を返せという課題があった場合。2 2 4の場合はどうする？真ん中はない？それとも2を返す？不明確な問題設定は結構ある。これは社会に出ても同様(要件定義があいまい)。クライアントとホストで例を出し合うのも大事ってよく話すのが大事。特に、Haskellは型が特定できるのでやりやすい。

P70~P71

ゴールとスタートの間にどんな関数があれば目的を達成できるか考える。

例えば、整数 x、y、z の中央値を求めるタスクを与えられたとする。その場合、どんな関数があれば目的を達成できるだろうか。その解の一つとして、betwweenなる関数を作ることを考える。は真ん中の数を割り出すプログラム。

```
-- between関数を定義
between :: Integer -> Integer -> Integer -> Bool
between m n p = n >= m && n <= p

-- middleNumber関数を定義
middleNumber :: Integer -> Integer -> Integer -> Integer
middleNumber x y z
  | between y x z = x
  | between x y z = y
  | otherwise = z
```

課題4.1(図は省いていい。利点と欠点は書いてください。)

P72~P73

馬の画像を受け取り、画像を4つ横に並べた画像を返す関数を作りたいとする。左上が元画像、右上が白黒反転かつ左右逆転画像、左下が白黒反転画像、右下が左右反転画像である。この時、一つの方法は左上と左下がつながった画像(`変数名:left`)と右上と右下がつながった画像(`変数名:right`)を既製品として一旦、以下のように定義することである。

```
fourPics pic =
    left 'beside' right
        where
            left = ,,,
            right = ,,,
```

whereはローカル定義で `left,right` を定義している。これらはfourPics関数の定義でしか使用できないのでローカルと呼ぶ。

思考の順番としては4枚の画像を並べるには、逆算して、

1. 左と右の画像2枚を横にくっつける。(`left 'beside' right`)
2. 左の画像(左上が元画像,左下が白黒反転画像)を定義する。(`pic 'above' invertColour pic`)
3. 右の画像(、右上が白黒反転かつ左右逆転画像,右下が左右反転画像)を定義する。(`invertColour (flipV pic) 'above' flipV pic`) よって以下のようになる。

```
fourPics pic =
    left 'beside' right
        where
            left = pic 'above' invertColour pic
            right = invertColour (flipV pic) 'above' flipV pic
```

この方法は、定義が仕様の後になっちゃう。でもこれが読みやすい。

P76~P77

スコープの話。トップレベルの定義ではスクリプト内部全体で使用できる。"全体"なので、例えそれがスクリプト的には定義前でも利用できる。以下の例を示す。

```
isOdd, isEven :: Int -> Bool
isOdd n
    | n <= 0    = False
    | otherwise = isEven (n-1)
isEven n
    | n < 0     = False
    | n == 0    = True
    | otherwise = isOdd (n-1)
```

この例では`isOdd n`の`otherwise = isEven (n-1)`を見ると`isEven`が定義前にもかかわらず、使用している。このようにトップレベルの定義ではスクリプト内部全体で使用できる。

それに対し、where節で与えられるローカル定義は、スクリプト全体ではなく、単にそれらが現れる条件式内でのみ使用することが意図されている。関数定義で使用される変数のスコープも同様。以下に例を示す。

```
maxsq x y
    | sqx > sqy  = sqx
    | otherwise  = sqy
       where
        sqx = sq x
        sqy = sq y
        sq :: Int -> Int
        sq z = z*z
```

課題4.9

P78

列挙型の説明。例えば、Rock,Scissors,Paperの3通りの値をもつ型を定義できる。

つまり集合

$$Move = \{Rock, Scissors, Paper\}$$

を考えることに相当する。

P79

具体的な方の定義方法は

```
data Move = Rock | Paper | Scissors
```

これで、Rock,Scissors,Paperの3通りの値をもつ型(Move型)を定義できる。これを使って、Rockなら Scissorsに勝つなども定義できる。

```
beat :: Move->Move --beatの引数に勝つ関数。
    beat Rock = Paper
    beat P
```

これは

$$beat : Move \to Move \quad beat(x) = x\text{に勝てる手}$$

を定義したことになる。

さらに

```
deriving (Show, Eq,Ord)
```

のように、derivingで型クラスを付与することができる。型クラスには以下脳ようなものがある。

```
Show        -- print で出力可能な文字列に変換される
Read        -- 文字列から変換可能となる
Eq      -- == や /= で比較可能となる
Ord     -- < や > 等で大小比較可能となる
Enum        -- fromEnum や toEnum で数値と相互変換可能となる
```

課題4.11と4.12

P81~P83

再帰は大事。例えば階乗なる演算子!を関数で書くと、!は一変数関数なので

$$! : N \to N \quad !(n) =!(n-1) \times n, \quad !(0) = 0$$

これをhaskellで書くと

```
fac :: Integer -> Integer
fac n = fac (n-1)*n -- !(n)=!(n-1)×n
fac 0 = 1 -- !(0)=0
```

↑ガード記法でもかける(P82)。

例えばここで、数学なら、!(-1)は **定義されていないという立場**($-1 \notin N$) をとれるがプログラミングの場合は(`Integer -> Integer` なので)そうもいかない。そこでotherwiseを使う。

課題P84(4.17と4.18)

P84~P85

P85の上にpower関数がある。2^n乗を求める再帰関数。

$$2^n = 2 \times 2^{n-1}$$

より

```
power2 n
    | n == 0    = 1
    | n > 0     = 2 * power2(n-1) --2^n=2 × 2^{n-1}
```

次に以下のような関数の定義を考える。

```
sumFacs n = fac 0 + fac 1 + … + fac (n-1) + fac n
```

$a_n = sumFacs(n)$と置けば、facは階乗のことなので$a_0 = 0! = 1$に注意して

$$a_n = a_{n-1} + fac(n), \quad a_0 = 1$$

よって

```
sumFacs n
    | n == 0 = 1
    | n > 0 = sumFacs (n-1) + fac n
```

課題 4.19,4.21

4.6章はとばす

4.7(P89)

より一般的な再帰。例えばフィボナッチ数列は

$$a_n = a_{n-1} + a_{n-2}, \quad a_1 = 1, \quad a_0 = 0$$

みたいな感じで、複数個を使う。

```
fib :: Integer -> Integer
fib n
    | n == 0    = 0
    | n == 1    = 1
    | n > 1     = fib (n-2) + fib (n-1)
```

P90

割り算の例だが授業ではほとんど流していたと思うので割愛。

P97

データ型には、多言語同様にタプル()とリスト[]も存在する。

- タプルは違う型(勿論同じでもいい)を数の制限を設けて定義 `例:(String,Int)`。
- リストは同じ型を数の制限無しで定義 `例:[Int]`。

```
("Salt: 1kg",139)
```

などといった感じで違うデータ型を一つのタプル内に混在できる。上記のタプルのデータ型は

```
(String,Int)
```

となる。さらに以下のようにエイリアス(別名)も定義できる。

```
type ShopItem = (String,Int)
```

リストの場合は同一型縛りがるため、型宣言は要素が何個あろうと

```
[(String,Int)]
[ShopItem]
```

などで済む。例えば

```
a = [1,5,8,10,25]
b = [1,2]
```

も同じ、`[Int]`型。

P100

より一般なタプルの形式は以下のようになる。

$$(t_1, t_2, \cdots, t_n)$$

$$(v_1, v_2, \cdots, v_n)$$

ただしtはデータ型、vは値。タプルを使うことで関数が複数の値を返せるようになる。例えば、

```
minAndMax :: Integer -> Integer -> (Integer , Integer)
minAndMax x y
    | x>=y      = (y,x)
    | otherwise = (x,y)
```

などといった形になる。

P101

タプルのパターンマッチング

```
addPair :: (Ineger,Integer) -> Integer
addPair (x,y) = x + y
```

といった形で関数側から使える。課題P103(5.1)、やりたいんだったら5.2もやっていいんですよ。

その他

"Haskellの少数型はfloatしかない"についての、聞き間違えか心配だったので検証結果。

```
ghci> a = 0.31426
ghci> :t a
a :: Fractional a => a
```

```
ghci> c = 0.31426 :: Float
ghci> d = 0.31426 :: Double
ghci> :t c
c :: Float
ghci> :t d
d :: Double
```

結果としては、小数をオーバーライドした型(`Fractional`)がデフォルトではつくようです。よって `Float` や `Double` に変換可能。自分の聞き間違え？