think and error

<iPadが出たので次世代ソフトウェ...

2010-01-31

Haskell基礎文法最速マスター

11:44 | 307 users

みんなHaskellやろうぜ！！
ということで書きました。
CやJava、Rubyなど他言語を知っている人は、これを読むことでHaskellの大抵のことはマスターでき…ません。
特殊な構文が多すぎて他言語との類推があまり利かないためです。残念です。
そんなんだから宇宙語だとかShe is Haskell.*1だとか言われてしまうのです。
そこで、何とかHaskell読めるくらいになることを目標に書きました。
リファレンス的な使い方も出来…ないと思います。Haskell入門を目指しました。

ここではHaskellの文法に焦点を当てていますので、テクニカルタームや良く使われる関数は省いていたりします。重要な事もサラッと書いてあったり抜けていたりするかもしれません。
また、デファクトスタンダードの実装であるGHC(Glasgow Haskell Compiler)の6.8.2くらいを想定してします。

前提知識:

- 純粋関数型言語
- 強い型付け
  - 型推論あるから型とか書かなくてもいい
    - もちろん書いた方がいい
- 静的言語
  - 別に動的な書き方も出来る(らしい)
- 遅延評価がデフォルト
  - 別に正格評価も出来る
- Haskellは楽しい
- 俺もまだ勉強中
  - 間違ってたらご指摘ください

目次

プロフィール

ruicc PLUS
思考錯誤とか。

日記の検索

検索
詳細 一覧

人気エントリー

Vim講座1 - think and error
648users
Haskell基礎文法最速マスター - think and error
307users
Vim講座3 - think and error
61users
プログラマのためのキーマップを本気で考えてみた - think an...
24users
Vimのモードと拡張性 - think and error
20users
vimperatorからはてブ - think and error
15users

カレンダー

<<	2010/01					>>
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

最近のコメント

2010-01-31 ruicc
2010-01-31 koyama41
2010-01-31 ruicc
2010-01-31 kazu-yamamoto
2010-01-31 ruicc

なぜHaskellか？

C++ より速くて、Perl より簡潔で、Python よりきちんとしていて、Ruby より柔軟で、C# より型が充実していて、Java より頑強で、PHP とは何の共通点もないものって？

Haskell ですよ！

コーナーケースを探すのにユニットテストを書くのに疲れた？ QuickCheck を使ってコンピュータに書かせちゃいましょう。正規表現ベースのパーサはメンテナンスしにくいのに気づいた？ Parsec を使って 15分で Perl6 の完全なパーサを書く方法を勉強しましょう。デッドロックやレースコンディションはもううんざり？ STM が concurrency 問題は全部解決してくれます。XS や SWIG が頭痛の種になってる？ FFI なら C コードを簡単に、かつ安全に埋め込めますよ。

Haskell は最先端の一般向け関数型言語で、バグフリーで、簡潔で、メンテナンスしやすいコードを、めちゃくちゃ短期間で書くことができちゃうんです。このトークでは、Haskell を日常のタスクに使う方法や、他の言語と連携させる Tips、それから生産性をものすごい勢いで向上させる秘密も教えちゃいます。

http://tokyo.yapcasia.org/sessions/learning_haskell.html

LLよりずっと速くて(それはそうだ:)、かつ書きやすい！保守しやすい！らしいよ！OCaml？ごめん聞こえない。

GUIには弱いだろ、と思ってたらxmonadなどというWindow Manager発見。Haskellでたった1200行で書かれているとか。
素晴らしいので良かったらぜひ。

- タイル型ウィンドウマネージャ Xmonad を使ってみた — ありえるえりあ
- ウィンドウマネージャxmonadが最強である5つの理由 - それ、Gentooだとどうなる？

-- GUI不得意は関係ないかな。

では以下テンション抑えてですます調。

基礎

Haskellには3つの実行形式が有ります。対話式インタプリタ、スクリプト実行、コンパイル実行の3つです。

対話式インタプリタ(ghci)

対話式インタプリタはコード片実行や型の確認と何かと便利です。
"Prelude>"はghciのプロンプトを表します。

カテゴリー
言語
vim
music

最新タイトル
Haskell基礎文法最速マスター
iPadが出たので次世代ソフトウェアキーボードを考えてみた
私的memo
haskellのMaybeとEither理解した。気がする。
debian squeezeでwebcam(UVC対応)使用
texでラテン筆記体
platexと文字コード
チャイコフスキ
debian squeezeでトラックポイント動作。
matlabのGUIから解法される方法

リンク集
error and error
vim and plasticity
それフィー
ぶくまでん
かえるの開発工房
teruyastarはかく語りき
現実が好き
青春スイーツ

最近のコメント
2010-01-31 ruicc
2010-01-31 koyama41
2010-01-31 ruicc
2010-01-31 kazu-yamamoto
2010-01-31 ruicc

最近のトラックバック
2010-01-31 なんとなく日記 - 関数型言語
2010-01-31 なんとなく日記 - 基礎文法最速マスターシリーズのまとめ
2010-01-31 Life like a clown - はてな的プログラミング言語人気ランキング
2010-01-31 [Diksam]Diksam基礎文法最速マスター
2010-01-31 どうでもいい情報置き場 - Whitespace基礎文法最速マスター

```
ruicc@debian:~$ ghci
GHCi, version 6.8.2: http://www.haskell.org/ghc/  :? for hel
p
Loading package base ... linking ... done.
Prelude> putStrLn "hello"
hello
Prelude> :type putStrLn
putStrLn :: String -> IO ()
Prelude> 1 + 1
2
Prelude> (+) 1 1
2
Prelude> :type (+)
(+) :: (Num a) => a -> a -> a
```

:typeは関数等の型を調べることが出来ます。
型が解らなく鳴ったらghciで:typeしましょう。
ghci終了はCtrl-Dです。

スクリプト実行(runghc)

Haskellソースをスクリプト実行が出来ます。
ghcrunではなく、runghcです。ターミナルでghcと打った後にtab押しても、それらしき物は補完されません。

```
ruicc@debian:~$ cat hello.hs
main = putStrLn "Hello, Haskell!"
ruicc@debian:~$ runghc hello.hs
Hello, Haskell!
```

コンパイル実行(ghc)

ネイティブコードを生成する最適化コンパイラです。

```
ruicc@debian:~$ ghc -o hello hello.hs
ruicc@debian:~$ ./hello
Hello, Haskell!
```

コメント

--のあと行末までがコメントになります。

```
Prelude> putStrLn "Guten morgen!" -- コメントです
Guten morgen!
```

--の直後に記号を置くと、Haskellはそれを新しい関数と勘違いしてエラーを吐きます。なので直後は空白を入れると良いです。

```
Prelude> putStrLn "Guten morgen!" --# too stupid!!
<interactive>:1:42: parse error (possibly incorrect indentat
ion)
```

```
{- {- 複数行コメントアウトは{- -}です。-}
Haskellではいくらネストしても問題ありません。 -}

--{- 複数行コメントアウトのコメントアウトです。
main = putStrLn "Hello?"
---}
```

基本演算

変数

letで変数を値に束縛します(という言い方をします)。

```
Prelude> let e = exp 1
Prelude> e
2.718281828459045
```

一度束縛したら、再代入(再束縛？)等の破壊的操作は出来ません。

四則演算

```
Prelude> 3 + 2
5
Prelude> 3 - 2
1
Prelude> 3 * 2
6
Prelude> 3 / 2
1.5
Prelude> 3 `mod` 2  -- 余りを求める
1
Prelude> 3^2  -- Intの累乗
9
Prelude> 3**2  -- Doubleの累乗
9.0
```

他の関数型言語のように前置形式にすることも出来ます。その際、記号を()で囲みます。

```
Prelude> (+) 3 2
5
Prelude> (-) 3 2
1
Prelude> (*) 3 2
6
Prelude> (/) 3 2
1.5
Prelude> mod 3 2
1
```

modは元々2つの引数をとる関数です。
Haskellでは``(backquote)で囲むことで、2つの引数をとる関数を「何でも」中置演算子として扱うことができます。

比較演算

同じ型を2つとり、Boolを返します。

```
Prelude> "moge" == "moge"
True
Prelude> 3.3 <= 1.2
False
Prelude> "hoge" /= "moge"  -- 等しくないは!=ではない
True
```

(/=)は数学の≠に似てますね。
(==)が同一性と同値性のどちらなのか気になる人もいると思いますが、Haskellは単純です。
Haskellでの等しさと言ったら同値性判定しか存在しません。
つまり、(==)は値が等しければ常にTrueです。

論理演算

論理演算はTrueとFalseのみを扱います。Int型の0やList型の[](空リスト)をFalseと扱うことは有りません。
というかHaskellは型の自動変換を一切行いません。型推定はしてくれますが。

```
Prelude> False && True
False
Prelude> (&&) False True
False
Prelude> False || True
True
Prelude> (||) False True
True
Prelude> not False -- 論理否定は!ではない
True
```

データ型

基本の型

どれも型名の初めは大文字で始まります。

Char型

1文字だけ''で囲みます。Unicode文字を表します。

```
Prelude> 'a'
'a'
Prelude> 'ab' -- error!
<interactive>:1:2: lexical error in string/character literal at character 'b'
Prelude> 'た'
'¥12383'
```

Bool型

TrueとFalseです。

Int型

システム依存の長さを持つ整数です。(32bitマシンなら32bit,64bitなら64bit)

Integer型
: 長さ制限のない整数型です。

Double型
: 浮動小数点を扱う型です。大抵64bit幅。

String型
: [Char]型に等しい。Charのリスト。""(double quote)で囲んで生成します。

```
Prelude> ['a','b','c']  -- [Char]型
"abc"
Prelude> "abc"  -- String型
"abc"
```

リスト

[]で囲み、,(comma)で区切ります。リストの中は同じ型しか入れられません。

```
Prelude> let arr = [1,2,3]
Prelude> let str = ['1','2','3'] -- String型に等しい
Prelude> let boo = [True, False, False]
Prelude> let strList = ["hoge", "moge", "fuga"]
```

リスト結合は(++)です。

```
Prelude> [1,2] ++ [3,4]
[1,2,3,4]
Prelude> "abc" ++ "def"
"abcdef"
```

(:)はリストの頭に要素を一つ追加します。

```
Prelude> 0 : [1,2,3,4]
[0,1,2,3,4]
Prelude> -2 : -1 : 0 : [1,2,3,4]
[-2,-1,0,1,2,3,4]
```

ここで(:)と(-)をくっつけて書いてしまうと、やはりHaskellが(:-)を新しい関数と勘違いするので注意。

タプル

()で囲み、,(comma)で区切ります。
タプルでは異なる型を格納することが出来ます。

```
Prelude> let tpl1 = ( 1, "one" )
Prelude> let tpl2 = ( True, 'a', [(1, "eins"), (2, "twei"),
(3, "drei")] )
```

pythonではリストが可変、タプルが不可変等決まっていますが、Haskellではそのようなことは気にすることはありません。
全て変更不可能(immutable)です。

代数データ型(data)

自分で型を定義できます。
まず組み込みのBool型について見ておきます。

```
data Bool = False | True    deriving (Eq, Ord)
```

「Bool型はTrueとFalseによって構成されている」と読みます。deriving以降は取り合えず無視します。

同じく組み込みのMaybe a型について。Maybe型ではなくて、Maybe a型です。

```
data Maybe a = Nothing | Just a   deriving (Eq, Ord)
```

「Maybe a型はNothingとJust aによって構成されている」と読みます。aは型変数であり、任意の型を表現しています(後述)。HaskellにはJavaのnullに対応する値が存在しないので、そのような欠損値が欲しいときに使われます。
このNothingがNullPointerExceptionを吐くことはありません:)

Haskellでは再帰的なデータ構造を扱うことが出来ます。
例えば二分木は以下のように簡潔に表せます。

```
data Tree = Empty | Node Int Tree Tree
```

関数

関数名の後に空白を挟んで引数を並べます。最外括弧はなくても問題ありません。

```
function arg1 arg2 arg3
```

型シグネチャ

func :: a -> a などと書いてある型シグネチャについて。
情報系の方にはお馴染みでしょうか。

```
Prelude> :type (&&)
(&&) :: Bool -> Bool -> Bool
```

2行目は、「(&&)関数は、引数にBool型とBool型をとり、Bool型を返す」と読みます。

```
Prelude> :type tail
tail :: [a] -> [a]
```

こちらは「tail関数は、引数に"何らかの型aのリスト"をとり、"その型aのリスト"を返す」と読みます。

```
Prelude> :type map
map :: (a -> b) -> [a] -> [b]
```

「map関数は、引数に"型aをとって型bを返す関数"と"型aのリスト"をとり、"型bのリスト"を返す」

ここで出てくるaやbは型変数と呼び、任意の型(Int型でも[Int]型でも(Int,[(Int, String)],[[Bool]])型でも良い)を表します。そして型変数が2回め現れたときは、その型は1回目と同じでなければならない、ということを意味しています。
a,b,c...とアルファベット順に現れるのは単に慣習です。先頭が小文字であれば何でも良いです。

```
Prelude> :type (+)
(+) :: (Num a) => a -> a -> a
```

「(+)関数は、引数に型aと型aをとり、型aを返す。ただし型aは数字に限定する」と読みます。だいたい。
"(Num a) =>"と言う部分は、型aを数字のみに限定していることを表しています。数字以外を持ってこられても(+)は何も出来ませんので。

関数定義

加算をする関数addを定義してみましょう。引数と返り値はIntに限定しておきます。

```
add :: Int -> Int -> Int
add a b     = a + b
```

1行目は型シグネチャです。なくてもHaskellは型推論でだいたい何とかしてくれます。
型シグネチャがない場合、add内で使用されている(+)の引数は数字に限定されるので、Haskellはaddの引数a,bは共に数字だと理解します。しかし型推定も万能ではないのでなるべく書きましょう。
2行目は「add関数は変数aに引数1を代入、変数bに引数2を代入して、(a + b)を返す」と読みます。"代入"と言ってますがこれは簡単のためで、正しくは"束縛"です。
(=)の右側はそのまま返り値です。別にreturnとか要りません。
-- Haskellでreturnは他言語とは別の意味で使われます。

パターンマッチ

Haskell特有なのでしょうか。
-- 調べたら他にも色々あった。Prolog,Erlang,ML,Ocaml,...
階乗を求める関数を考えます。

```
factorial :: Int -> Int
factorial 0     = 1
factorial n     = n * factorial (n - 1)
```

1行目は型シグネチャです。
2,3行目は上から下へ順に考えていきます。
「まず引数が0ならば、factorialは1を返す。(2行目)
そうでなければ変数nを引数に束縛し、(n * factorial (n-1))を返す(3行目)」

-- 因みに3行目の(n-1)の括弧はないとstack overflowエラーになります。
-- Haskellでは関数が真っ先に評価されるため、
-- (n * factrial n - 1) では ((n * (factrial n)) - 1) と解釈されてしまうためです。

map関数のhaskellでの実装です。

```
map :: ( a -> b ) -> [a] -> [b]
map _ []          = []
map f (x:xs)      = f x : map f xs
```

2行目の"_"はワイルドカードです。「引数に何が来てもマッチするが、(使わないから)変数で束縛しない」という意味です。
3行目は第1引数をfで、第2引数をxとxsで束縛しています。
(:)とはリストの頭に要素を加える関数ですが、関数定義の"="の左側では変数の束縛にも使われます。
「第2引数(リスト)の先頭をxで、先頭を除いたリストをxsで束縛する」という意味です。

パターンマッチはどんなに複雑でも構いません。

```
func :: (Int, String, [Bool]) -> String
func (n, s, [])          = show n ++ s ++ "Empty list"
func (n, s, (b:bs))      = show n ++ s ++ show b
```

この関数はただのパターンマッチの例です。意味はありません。
showは基本的な型を文字列に変換してくれる関数です。

ガード条件

型だけでは分岐しきれない場合、ガード条件を用いてさらに詳細に分岐が出来ます。

```
absolute :: (Num a, Ord a) => a -> a
absolute a | a < 0        = -a
           | otherwise    = a
```

ここで'='の左側、'|'の右側がガード条件です。
ガード条件も上から順に評価されます。
「abs関数は引数をaで束縛したあと、(a < 0)がTrueならば -aを返す。
そうでないならば、aを返す。」
-- otherwiseてただのTrueらしい。

再帰

まあ知ってますよね。
Hakellにはforやwhileなんてありません。
しかしRubyでfor文をほとんど(明示的に)使わないように、Haskellでも明示的な再帰は余り使われないようです。
mapやfilter,foldなどの高階関数が使えるなら使いましょう。とのこと。
文法に特殊性はないので省きます。

代数データ型のパターンマッチ

Maybe a型の(Just x)からxを取り出したいときはパターンマッチを用います。

```
func :: (Show a) => Maybe a -> String
func (Just x)    = "This is just a " ++ show x
func Nothing     = "Nothing!"
```

2行目で変数xに実際の値を束縛しています。
ここでも(Just　x)の括弧は欠かせません。そうでないとHaskellは、2行目でfuncの引数が2つに見えてしまうからです。

制御構文?
if式はもはや他言語の3項演算子に近いような。else以降は省略不可です。

```
if str=="hoge" then True else False
```

見にくい?

```
if str=="hoge"
   then True
   else False
```

case式。ここでもパターンマッチとガード条件を使うことが出来ます。

```
case value of
    (Just x) | x > 0    -> "Positive!"
             | x = 0    -> "Zero!"
             | othewise -> "Negative!"
    Nothing             -> "Nothing!"
```

ここではvalueの値によって識別されます。

IO

入出力なんて不純なものはpureなHaskellに必要ありませんね。嘘です。
入出力を扱う関数は、型を調べると"IO"という2文字がくっついています。型を調べるだけで、純粋関数か否かがわかるということです。
ファイルの入出力からです。以下"Real World Haskell"7章からのサンプル。

```
import System.IO
import Data.Char(toUpper)

main = do
       inh <- openFile "input.txt" ReadMode  -- 読み込みでinput.txtを開く
       outh <- openFile "output.txt" WriteMode -- 書き込みでoutput.txtを開く
       inpStr <- hGetContents inh          -- ハンドラから取り出す
       hPutStr outh (map toUpper inpStr)   -- ハンドラに渡す
       hClose inh  -- ファイルを閉じる
       hClose outh  -- ファイルを閉じる
```

このプログラムはinput.txtを読み込み、全て大文字に変換してoutput.txtに書き

込むというプログラムです。
初めの2行はモジュールのインポートです。
その下にmain関数が出てきました。Haskellのプログラムはここから始まります。
doブロックは以降を順次実行します。doが現れたら以降はインデントを揃えます。
"<-"はモナドから値を一つ取り出し、変数をそれに束縛させます。型シグネチャの"->"に似ていますが関係はなく、数学の　から来ているそうです。

hから始まる関数はファイルハンドラを扱う関数です。hから始まる関数はhなしのものも大抵存在しており、それはハンドラの代わりに標準入出力を扱います。

ハンドラ	標準入出力	説明
openFile	なし	ファイルを開く
hGetContents	getContents	中身を取り出す
hGetLine	getLine	一行取り出す
hPutStr	putStr	Stringを書き込む
hPutStrLn	putStrLn	Stringを書き込んで改行を追加する
hPrint	print	(h)putStrLn . show に同じ
hClose	なし	ファイルを閉じる

他言語ではハンドラを用いて「fileから一行ずつメモリに取り込んで処理」と言う方法が定番ですが、Haskellは遅延評価というものがあります。必要になって初めて評価するのがHaskellです。
つまりどんな大きいファイルでも「一度に取り込み後はそれを処理する」という書き方をすれば、後はHaskellが必要な所だけ取り出して処理してくれるのです。
ファイルを閉じるhCloseは、書かなくてもGCが適宜回収してくれますが、Cとの混成プログラミング等でC側にバグがある時など危険なこともあるので出来れば書いておきましょう。

しかし上記プログラムは長すぎます。readFileとwriteFileを用いると短く書くことが出来ます。

```
import System.IO
import Data.Char(toUpper)

main = do
       inStr <- readFile "input.txt" -- openFileしたあとhGetContentsし、Stringを返す
       writeFile "output.txt" (map toUpper inpStr) -- openFileしたあとhPutStrで書き込み、hCloseする
```

これでハンドラが見えなくなりました。
こんな大雑把な書き方でいいのは遅延処理のおかげでしょう。

さらに、標準入力から取り込んで標準出力へ返す、という操作もまた頻繁に用いられるので、interactという関数が用意されています。

```
import Data.Char(toUpper)

main = interact (map toUpper)
```

interactに(String -> String)型の関数を渡してあげれば、標準入力にその関数を適用して標準出力に返してしてくれます。
このファイルをtoupper.hsとしてターミナル上で

```
$ ls | runghc toupper.hs
```

とうつと、全ての文字が大文字になって出力されます。

知っておいた方が良い文法

俺もまだ勉強中なので良く知らないのです。
何かあったら追加します。

括弧を少なく($)

次の2つは同じことを表現しています。

```
func1 (func2 (func3 (func4 (func5 arg1 arg2))))
```

```
func1 $ func2 $ func3 $ func4 $ func5 arg1 arg2
```

$の位置から文末までを括弧で囲む、という意味です。確か。
括弧が少なくなります。

合成関数

数学の と一緒です。引数が1つの関数をつなぎ合成関数を作ります。

```
( func1 . func2 . func3 . func4. func5 ) arg1 arg2
```

こちらは(func1 . func2 . func3 . func4 . func5)という合成関数に引数arg1,arg2を渡す、という意味です。
前述の$とは意味が異なります。
JavascriptやRubyの"."とは関数適用順番が違うので注意。

where節

局所的な束縛を導入します。簡単に言えば、ローカル変数、ローカル関数が定義出来るイメージです。
以下は良く見かけるHaskellの例、quicksortです。

```
qsort []     = []
qsort (x:xs) = qsort [y | y <- xs, y < x] ++ [x] ++ qsort
[y | y <- xs, y >= x]
```

このままでもいいのですが、whereを使うと例えば以下のように書けます。

```
qsort []     = []
qsort (x:xs) = qsort lt ++ [x] ++ qsort geq
               where
                 lt   = [y | y <- xs, y < x]
                 geq  = [y | y <- xs, y >= x]
```

ここで束縛したltとgeqはqsortの中でしか使えません。
また、whrer節の中ではqsortでの変数x,xsが使えています。
まあ英語のwhereそのままです。
-- where以降のインデントは揃えます。

let式

局所的な束縛を導入します。また、式なのでlet式自体も値を持ちます。
何かもうサンプルが浮かばないので"普通のHaskellプログラミング"P185より。

```
f n = let x = n + 1
          y = n + 2
          z = n + 3
      in x * y * z
```

in以降がlet式の値です。
-- こちらもインデント注意。

部分関数適用

2つ引数を取る関数に引数を一つだけ渡すと、「1つの引数をとる関数」になります。

```
Prelude> :type (+)  -- 引数を2つ取る関数
(+) :: (Num a) => a -> a -> a
Prelude> :type (3 +)  -- 引数を1つ取る関数
(3 +) :: (Num t) => t -> t
Prelude> (3 +) 8   -- 関数(3 +)に引数8を与える
11
```

よって、Listの全ての要素をそれぞれ3ずつ足す、という関数は以下のようにかけます。

```
Prelude> map (3 +) [1..5]
[4,5,6,7,8]
```

リスト生成(..)

単純な数字のリストは簡単に生成出来ます。

```
Prelude> [1..3]  -- 1から3まで
[1,2,3]
Prelude> [1,3..10]  -- 1,3,5,7...を10まで
[1,3,5,7,9]
Prelude> [1..]  -- 無限リスト
[1,2,3,4, (...ずっと続くので省略)
Prelude> [0.1,0.2..0.5]  -- doubleでも出来ます
[0.1,0.2,0.30000000000000004,0.4,0.5]
```

doubleではdoubleぽいリストになってしまいますね。

あ、Char型でも出来ますね。

```
Prelude> ['a'..'d']
"abcd"
Prelude> ['A'..'E']
"ABCDE"
Prelude> ['1'..'4']
"1234"
Prelude> ['あ'..'か']
"¥12354¥12355¥12356¥12357¥12358¥12359¥12360¥12361¥12362¥1236
3"
```

リスト内包表記

数学での内包表記　　と同じようなことをリストで実現出来ます。

```
Prelude> [abs x| x<-[-5..5] ]
[5,4,3,2,1,0,1,2,3,4,5]
Prelude> [x| x<-[1..10], odd x ]
[1,3,5,7,9]
Prelude> [(x,y)| x<-[1..3], y<-['a'..'c']]
[(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b'),(2,'c'),(3,'a'),
(3,'b'),(3,'c')]
Prelude> [x*x| x<-[1..10], odd x ]
[1,9,25,49,81]
```

以下が一番特徴を掴みやすいでしょうか。

```
Prelude> [ x*x | x<-[1..10], odd x ]
[1,9,25,49,81]
```

「[1..10]のリストの要素について(x<-[1..10])、(odd　x)がTrueである要素を集めリストを作り、そのリスト要素をそれぞれ二乗する(x*x)」と読みます。
listにfilter関数とmap関数をくっつけたようなものでしょうか。

俺はこのリスト内包表記知って感動しました。
プログラミングで集合を簡単に(まさに数学の内包定義のように！)定義出来たらなーと以前から考えていたので。既にあったとは。
当然mapとfilter使えば似たようなことは出来ますが、読みやすいのがまたいい。
数学の定義そのままだし。Haskell最高。

また、この表記では"<-"が　という感じが出ててまたいいですね。

こんなことも出来ます。

```
Prelude> let set = [(x,y)| x<-[1..5], y<-['a'..'c']]  -- リ
ストset を定義しておく
( ...省略... )
Prelude> [ e | e@(n,c)<-set, odd n, c=='d']  -- setの中か
ら、タプル1項目が奇数、2項目が'd'のものを集めたリスト
[(1,'d'),(3,'d'),(5,'d')]
```

パターンマッチが使えるということです。リストの要素を精査して、選り抜き、好きなように変換することが出来るのです。

これによりHaskellが最強になりました。たぶん。

Haskell資料

- 簡潔かつ段階的に読者の理解を深めながらも、遅延処理の仕組みにきちんと答えてくれた筆者に感謝を。

ふつうのHaskellプログラミング ふつうのプログラマのための関数型言語入門

作者: 青木峰郎,山下伸夫
出版社/メーカー: ソフトバンククリエイティブ
発売日: 2006/06/01
メディア: 単行本
購入: 11人 クリック: 87回
この商品を含むブログ (311件) を見る

- 俺の知識欲に十分に答えてくれるO'Reillyと筆者、訳者に敬意を表して。

Real World Haskell―実戦で学ぶ関数型言語プログラミング

作者: Bryan O'Sullivan,John Goerzen,Don Stewart,山下伸夫,伊東勝利,株式会社タイムインターメディア
出版社/メーカー: オライリージャパン
発売日: 2009/10/26
メディア: 大型本
購入: 5人 クリック: 73回
この商品を含むブログ (33件) を見る

- こちらは未読。

プログラミングHaskell

作者: Graham Hutton,山本和彦
出版社/メーカー: オーム社
発売日: 2009/11/11
メディア: 単行本(ソフトカバー)
購入: 5人 クリック: 262回
この商品を含むブログ (40件) を見る

- Haskell / Wikipedia
- 本物のプログラマはHaskellを使う - 本物のプログラマはHaskellを使う:ITpro
- Hoogle, haskell API検索
- Real World Haskell online版
- Haskell階層ライブラリー覧

その他の基礎文法最速マスター

増えてきましたね。OCalmとScala,clojureあたりを誰か書いてくれないかなー。

- Perl基礎文法最速マスター　-　Perl入門〜サンプルコードによるPerl入門〜
- Route 477 - Ruby基礎文法最速マスター - , 1. 基礎 , 2. 数値 , 3. 文字列 , 4. 配列 , 5. ハッシュ , 6. 制御文 , 7. サブルーチン , 8. ファイル入出力 , 知っておいた方がよい文法 , 余談 , (おまけ)Ruby書籍紹介
- PHP基礎文法最速マスター | Shin x blog
- Python基礎文法最速マスター - D++のはまり日誌
- Java基礎文法最速マスター - 何かしらの言語による記述を解析する日記
- VBA基礎文法最速マスター - 何かしらの言語による記述を解析する日記
- Bash基礎文法最速マスター - 何かしらの言語による記述を解析する日記
- Brainf*ck基礎文法最速マスター - 医者を志す妻を応援する夫の日記
- VBScript 基礎文法最速マスター - CX's VBScript Diary - VBScript グループ
- JavaScript基礎文法最速マスター - なんとなく日記
- Vimスクリプト基礎文法最速マスター - 永遠に未完成
- D言語基礎文法最速マスター - はてなかよっ！
- C++0x基礎文法最速マスター - Faith and Brave - C++で遊ぼう
- 読書メモ＋tips＋日記:[Flash] ActionScript 3.0 基礎文法最速マスター
- Emacs Lisp基礎文法最速マスター - (rubikitch loves (Emacs Ruby CUI))

*1:http://d.hatena.ne.jp/shunsuk/20100127/1264587276

コメントを書く

74 2010/01/31 15:14
curry化ならcurryとuncurryですね

ruicc 2010/01/31 21:44
>74さん
ありがとうございます。
curry化についてはちゃんと調べてから書くことにしました。

8329 2010/01/31 22:51
関数適用は中置演算子適用よりも優先順位が高いので、
func1 . func2 . func3 . func5 arg1 arg2
は
func1 . func2 . func3 . (func5 arg1 arg2)
と同じ意味になります。

kazu-yamamoto 2010/02/01 10:52
8329 さんも書いていますが、正しくは以下のようにします。
(func1 . func2 . func3 . func5) arg1 arg2
Maybe の本当の意味は、
http://d.hatena.ne.jp/kmaebashi/20091223/p1#20091223f2
を読むといいでしょう。

ruicc 2010/02/01 11:42
>8329さん、kazu-yamamotoさん
ありがとうございます。
"."に関して勘違いしていたようです。変更しました。

ruicc 2010/02/01 13:18
>kazu-yamamotoさん
>Maybe の本当の意味は、
>http://d.hatena.ne.jp/kmaebashi/20091223/p1#20091223f2
>を読むといいでしょう。

読みました。言語設計としてはnullはない方が良いのですね。
再帰出来るデータ型についても後で追加したいと思います。

kazu-yamamoto 2010/02/01 18:46
ちなみに、関数の入り口での分割も、if も、case の糖衣構文だと知れば、もう少し簡単に文法を理解できるようになるでしょう。

ruicc 2010/02/02 01:05
>kazu-yamamotoさん
ifも内部はcaseなのですか。
分岐は全てcaseが担っているのですかね。

koyama41 2010/02/02 14:36
Boolは
data Bool = False | True deriving (Eq, Ord)
ですね（ほんとうはもっとderivingしてるけど）
Falseの方が左でないと、derivig Ord との辻褄が合いませんです。(False < True を評価したら True になる）

ruicc 2010/02/02 17:21
>koyama41さん
ありがとうございます。直しました。
順番に意味があったのですか。自分で書いててderiving Ordの意味が良く解ってませんでした…Boolにも序列があったのですね。

トラックバック - http://d.hatena.ne.jp/ruicc/20100131/1264905896
医者を志す妻を応援する夫の日記 - Brainf*ck基礎文法最速マスター
なんとなく日記 - JavaScript基礎文法最速マスター
燈明日記 - 基礎文法最速マスターぞくぞくキターーー！
何かしらの言語による記述を解析する日記 - Java基礎文法最速マスタ...
何かしらの言語による記述を解析する日記 - Bash基礎文法最速マスタ...
何かしらの言語による記述を解析する日記 - VBA基礎文法最速マスター
CX's VBScript Diary - VBScript 基礎文法最速マスター
surume000の日記 - プログラミング言語基礎文法最速マスターまとめ
きまぐれメモ - 各種言語による基礎文法最速マスターまとめ
shikaku's memo blog - ○○基礎文法最速マスター
永遠に未完成 - Vimスクリプト基礎文法最速マスター
きまぐれメモ - 各種言語による基礎文法最速マスターまとめ
[Flash] ActionScript 3.0 基礎文法最速マスター
(rubikitch loves (Emacs Ruby CUI)) - Emacs Lisp基礎文法最速マス...
どうでもいい情報置き場 - Whitespace基礎文法最速マスター
[Diksam]Diksam基礎文法最速マスター
Life like a clown - はてな的プログラミング言語人気ランキング
なんとなく日記 - 基礎文法最速マスターシリーズのまとめ
なんとなく日記 - 関数型言語

リンク元
270 http://b.hatena.ne.jp/hotentry
212 http://reader.livedoor.com/reader/
145 http://b.hatena.ne.jp/hotentry/it
117 http://pipes.yahoo.com/pipes/pipe.info?_id=faa858a20082ef6d25ad27557e37e011
107 http://d.hatena.ne.jp/gifnksm/20100131/1264934942
106 http://www.google.co.jp/reader/view/
86 http://www.google.com/reader/view/
77 http://twitter.com/
54 http://d.hatena.ne.jp/
49 http://www.google.co.jp/reader/view/?hl=ja&tab=wy

おとなり日記
2010-02-02 use GFx::WebLog; 5/91 5%

<iPadが出たので次世代ソフトウェ...