FUNCTIONAL PROGRAMMING
第10回　モジュール

萩野　達也
hagino@sfc.keio.ac.jp

モジュール

- モジュールは以下のエンティティを含みます.
  - 変数
  - 型コンストラクタ
  - データコンストラクタ
  - フィールドラベル
  - 型クラス
  - クラスメソッド
- Javaのパッケージに似ている
- 名前空間はモジュールごとに分かれている
  - 名前(識別子)はモジュールで一意的でなくてはいけない
  - モジュールが異なれば同じ名前を使ってもかまわない

Haskellの標準モジュール

module	description
`Prelude`	基本的な関数と型とクラス
`Data.Ratio`	有理数
`Data.Complex`	複素数
`Numeric`	数値
`Data.Ix`	値と整数の対応（配列の添え字で利用）
`Data.Array`	配列
`Data.List`	リスト関係の関数など
`Data.Maybe`	Maybeモナド
`Data.Char`	文字関係の関数など
`Control.Monad`	モナド関係の関数など
`System.IO`	入出力
`System.Directory`	ディレクトリ操作
`System.Environment`	コマンド引数や環境変数など
`Data.Time`	日時

See https://downloads.haskell.org/~ghc/latest/docs/html/libraries/

module 宣言

```
module name where
...
```

- ***name*** のモジュールを宣言する.
  - モジュール名は大文字で始めること
  - 複数の名前を「.」でつなぐこともできる.
  - モジュールで定義されたエンティティ(変数, 関数, 型, クラスなど)が外部にエキスポートされる.

例

```
module FileUtils where
data SomeType = ConsA String | ConsB Int
makePath = ....
forceRemove = ...
```

一部のみエキスポートする

- 通常はすべてがエキスポートされる

```
module FileUtils (makePath, forceRemove) where
```

- エキスポートしたいものをリストする
- 上記の宣言では **makePath** と **forceRemove** のみがエキスポートされ，他は隠される．

```
module FileUtils (joinPath, (+), concatPath) where
```

- データコンストラクタ以外のものはエキスポートするリストに書くことができる．
- 演算子は括弧で囲むこと．

データコンストラクタのエキスポート

- データコンストラクタだけをエキスポートすることはできない.
- データ型と一緒にエキスポートする必要がある.

```
module AnyModule (SomeType(ConsA), a, b, c) where
data someType = ConsA String | ConsB Int
```

- データコンストラクタ **ConsA** をデータ型 **SomeType** と一緒にエキスポートする.
- **consB** はエキスポートされない.

```
module AnyModule (SomeType(ConsA, ConsB), a, b, c) where
```

- データコンストラクタ **ConsA** および **ConsB** の両方がエキスポートされる.

```
module AnyModule (SomeType(..), a, b, c) where
```

- **SomeType** のすべてのデータコンストラクタおよびフィールドラベルがエキスポートされる.

モジュールをエキスポートする

- インポートしたモジュールをそのままエキスポートすることができる.

```
module LineParser
    (module Text.ParserCombinators.Parsec.Prim,
     LineParser, indented, blank, firstChar, anyLine) where

import Text.ParserCombinators.Parsec.Prim
```

- **Text.ParserCombinators.Parsec.Prim** で定義されたすべてのエンティティがエキスポートされる.

Mainモジュール

- モジュール宣言で始まらないファイルは，次の **Main** モジュールの宣言が最初になされたものとみなされる．

```
module Main(main) where
```

- **Main** モジュール
- **main** のみがエキスポートされる．
- **main** の型は **(IO a)** でなくてはいけない．
- **main** はプログラムのエントリーポイントとなる．

import 宣言

- モジュールで宣言されたエンティティを利用するためにはインポートする必要がある.

```
import Text.Regex
```

- 何も指定しない場合は，モジュールで定義されたすべてのエンティティがインポートされる.
- インポートするエンティティをリストすることで制限することができる.
  - データコンストラクタはデータ型と一緒に指定すること.

```
import Text.Regex(mkRegex, matchRegex)
```

- インポートしないエンティティを指定することもできる.

```
import Monad hiding (join)
```

- **Monad** で定義された **join** 以外のエンティティをインポートする.

修飾された名前（Qualified Name）

- エンティティはモジュール名を付けた形の完全に修飾された名前の形で利用することができる.

```
moduleName.entityName
```

- インポートしたエンティティは修飾された名前あるいはモジュール名を省略した名前の両方で利用することができる.

```
import Text.Regex(mkRegex)
... mkRegex ...
... Text.Regex.mkRegex ...
```

- 修飾された名前のみをインポートしたい場合には，**qualified** を付けてインポートすればよい.
  - 名前の衝突を回避できる.

```
import qualified Text.Regex
```

- インポートするときに **as** を使ってモジュール名に別名を付けることもできる.

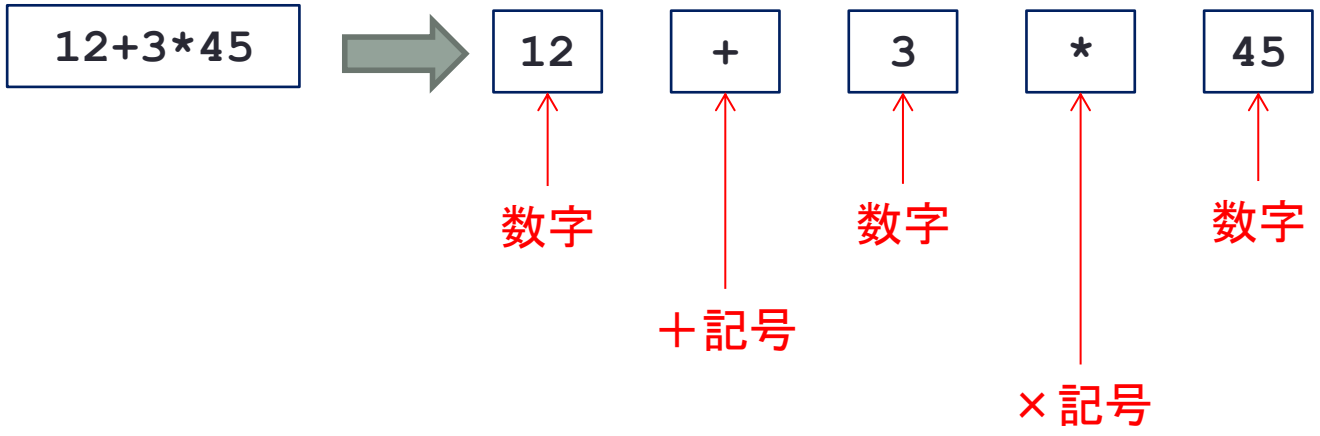
```
import qualified Distribution.Simple.GHCPackageConfig as Conf
```

電卓を作ってみよう

- 次のような簡単な計算のできる電卓を作成してみよう.

```
12+3*45 ⇒ 147
(1+2)*(3+4) ⇒ 21
```

- 最初に，入力された文字列を字句（token）のリストに変換する.



字句をデータ型として定義

```
data Token = Num Int | Add | Sub | Mul | Div
```

- 字句は数字か記号(4種類)のどちらか.

```
tokens::String -> [Token]
tokens [] = []
tokens ('+':cs) = Add:(tokens cs)
tokens ('-':cs) = Sub:(tokens cs)
tokens ('*':cs) = Mul:(tokens cs)
tokens ('/':cs) = Div:(tokens cs)
tokens (c:cs) | isDigit c = let (ds,rs) = span isDigit (c:cs)
                            in Num(read ds):(tokens rs)
```

- **span** はリストの先頭から条件を満たす部分を切り出す関数
  - `span :: (a -> Bool) -> [a] -> ([a], [a])`
  - `span (< 3) [1,2,3,4,1,2,3,4] = ([1,2],[3,4,1,2,3,4])`
  - `span (< 9) [1,2,3] = ([1,2,3],[])`
  - `span (< 0) [1,2,3] = ([],[1,2,3])`

練習問題10－1

- Tokenモジュールを定義し，正しく動くかテストしなさい．

Token.hs

ghcではファイル名をモジュール名に合わせる必要がある．

```
module Token(Token(..),tokens) where

import Data.Char

data Token = Num Int | Add | Sub | Mul | Div
             deriving Show

tokens::String -> [Token]
tokens [] = []
tokens ('+':cs) = Add:(tokens cs)
tokens ('-':cs) = Sub:(tokens cs)
tokens ('*':cs) = Mul:(tokens cs)
tokens ('/':cs) = Div:(tokens cs)
tokens (c:cs) | isDigit c = let (ds,rs) = span isDigit (c:cs)
                            in Num(read ds):(tokens rs)
```

tokenTest.hs

```
import Token

main = do cs <- getContents
          putStr $ unlines $ map (unwords . (map show) . tokens) $ lines cs
```

練習問題10－2

- 字句リストを評価して，計算を行いましょう．
  - 下のプログラムは足し算を行う部分だけです．他の演算子も追加してください．

calc.hs

```
import Token

calc::[Token] -> Int
calc [Num x] = x
calc (Num x:Add:Num y:ts) = calc (Num (x+y):ts)
....

main = do cs <- getContents
          putStr $ unlines $ map (show . calc .tokens) $ lines cs
```

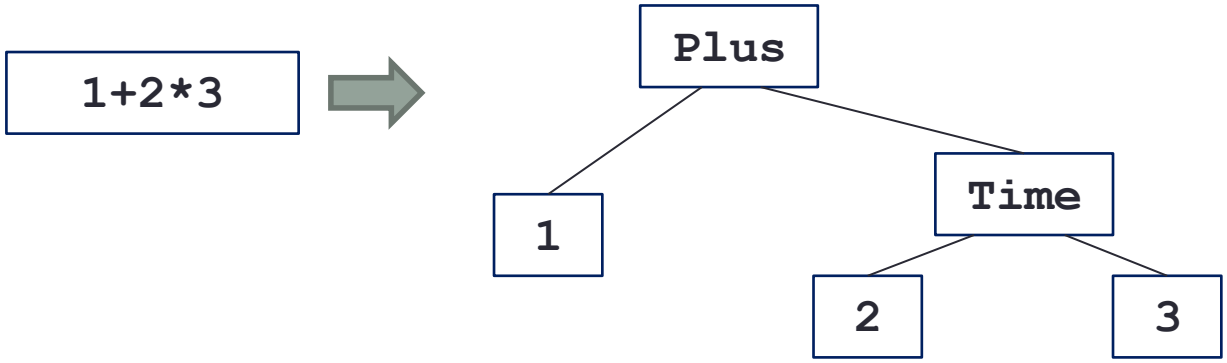
- **実行例**

```
% ghc calc.hs
...
% ./calc
1+2
3
1+2+3+4+5+6+7+8+9
45
1+2*3-4/5
7
```

構文木の作成

- 1+2*3 を 1+(2*3) と解釈するためには，字句のリストを先頭から順に計算するのではなく，一度構文木を作成した方が簡単にできます．
- 構文木をデータ型として定義します．

```
data ParseTree = Number Int |
                 Plus ParseTree ParseTree |
                 Minus ParseTree ParseTree |
                 Time ParseTree ParseTree |
                 Divide ParseTree ParseTree
```



パーサ

- 字句のリストから構文木を作るのがパーサです.
- パーサは次の型を持ちます.

```
[Token] -> (ParseTree, [Token])
```

  - 字句の列が与えられ，解析して出来上がった構文木と残りの字句の列を返します.

- 式の構文(BNF)

```
expr ::= term (("+" | "-") term)*
term ::= factor (("*" | "/") factor)*
factor ::= number | "(" expr ")"
```

BNFでは`(...)*`は`{...}`で書かれることも多い. 0回以上の繰り返しを意味する.

```
[Num 1, Mul, Num 2, Add, Num 3]
```

term パーサ

```
((Time (Number 1)(Number 2)), [Add, Num 3])
```

練習問題10−3

- 足し算をパースする.

```
import Token

data ParseTree = ... deriving Show

type Parser = [Token] -> (ParseTree, [Token])

parseFactor::Parser
parseFactor(Num x:l) = (Number x, l)

parseTerm::Parser
parseTerm l = parseFactor l

parseExpr::Parser
parseExpr l = nextTerm $ parseTerm l
  where nextTerm(p1, Add:l1) = let (p2, l2) = parseTerm l1
                               in nextTerm(Plus p1 p2, l2)
        nextTerm x = x

main = do cs <- getContents
          putStr $ unlines $ map (show . fst . parseExpr .tokens) $ lines cs1
```

parseExpr の動作

```
parseExpr [Num 1,Add,Num 2,Add,Num 3]
⇒ nextTerm $ parseTerm [Num 1,Add,Num 2,Add,Num 3]
⇒ nextTerm (Number 1,[Add,Num 2,Add,Num 3])
⇒ let (p2,l2) = parseTerm [Num 2,Add,Num 3]
   in nextTerm(Plus(Number 1) p2, l2)
⇒ let (p2,l2) = (Number 2,[Add,Num 3])
   in nextTerm(Plus(Number 1) p2, l2)
⇒ nextTerm(Plus(Number 1)(Number 2),[Add,Num 3])
⇒ let (p2,l2) = parseTerm [Num 3] in
   nextTerm(Plus(Plus(Number 1)(Number 2)) p2,l2)
⇒ let (p2,l2) = (Number 3,[])
   in nextTerm(Plus(Plus(Number 1)(Number 2)) p2,l2)
⇒ nextTerm(Plus(Plus(Number 1)(Number 2))(Number 3),[])
⇒ (Plus(Plus(Number 1)(Number 2))(Number 3),[])
```

式の評価

- パースしてできた構文木を評価して値を求める.

```
eval::ParseTree -> Int
eval(Number x) = x
eval(Plus p1 p2) = eval p1 + eval p2
eval(Minus p1 p2) = ...
eval(Time p1 p2) = ...
eval(Divide p1 p2) = ...
```

```
main = do cs <- getContents
          putStr $
            unlines $
            map (show . eval . fst . parseExpr . tokens) $
            lines cs
```

練習問題10－4

- 足し算だけでなく，他の四則演算も扱えるようにしなさい．
  - 空白は無視するようにしましょう．
  - 括弧についても正しく計算できるようにしなさい．
  - 「+」と「-」は2項演算だけでなく単項演算子でもあることを，正しく計算できるようにしなさい．
  - 課題の提出の都合上Token.hsを分離せずにcalc.hsの中にすべて入れてください．

calc.hs

```
import Data.Char

data Token = Num Int | Add | Sub | Mul | Div | LPar | RPar

tokens::String -> [Token]
tokens = ...

data ParseTree = ...
type Parser = [Token] -> (ParseTree, [Token])

parseFactor = ...
parseTerm = ...
parseExpr = ...

eval::parseTree -> Int
eval = ...

main = do cs <- getContents
          putStr $ unlines $ map (show . eval . fst . parseExpr . tokens) $ lines cs
```

実行例

```
% ./calc
2 + 3   *  4
14
3 / 4 * 5
0
10 - 6 - 2 * 2
0
(1 + 2)*(5  - 3) / 3
2
- 3 * + 2
-6
```