関数型プログラミング
第10回　モナド

萩野　達也
hagino@sfc.keio.ac.jp

Slide URL

https://vu5.sfc.keio.ac.jp/slide/

モナドのクラス

```
class Monad m where {
  (>>=)  :: m a -> (a -> m b) -> m b ;
  return :: a -> m a
}
```

- Monad クラスのインスタンスがモナド
  - 2つの関数を実装する必要がある.
  - (>>=) は**バインド**(bind)と呼ばれる
- 2つの関数は次の規則を満たしている必要がある.
  - **モナド則**

```
1. (return x) >>= f    = f x
2. m >>= return        = m
3. (m >>= f) >>= g     = m >>= (\x -> f x >>= g)
```

Maybeモナド

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

- 「Maybe a」は失敗を扱うためによく用いられる.
  - 「Just x」は成功した場合の値を表している.
  - 「Nothing」は失敗を表している.
- f:: a -> Maybe b
  - f は「b」の型の値を返すかもしれない.
  - 「b」の型の値を返すことができない場合には「Nothing」を返す.

例

```
lookup::(Eq a) => a -> [(a, b)] -> Maybe b
```

```
instance Monad Maybe where {
   (Just x) >>= f   = f x ;
  Nothing   >>= f   = Nothing ;
   return x         = Just x
}
```

lookup

```
lookup::(Eq a) => a -> [(a,b)] -> Maybe b
```

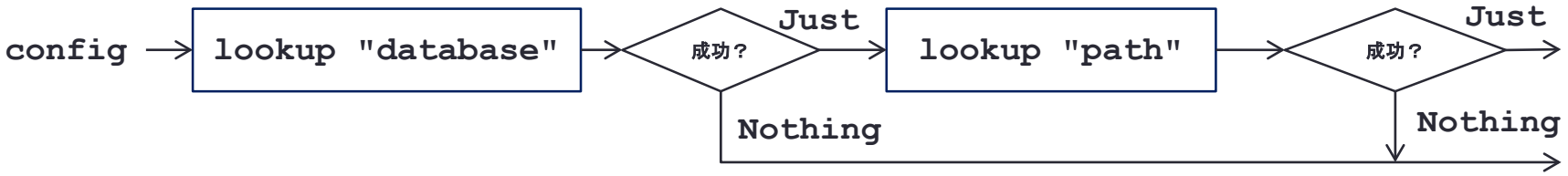
- **lookup** は2つの引数を取る:
  - インデックス
  - 連想リスト(タプルのリスト)
- **lookup** は次の値を返す:
  - 与えられたインデックスのタプルがあった場合には, 対応する値を「**Just x** 」として返す.
  - 対応するタプルがなかった場合には,「**Nothing**」を返す.

```
lookup "three" [("one", 1), ("two", 2), ("three", 3)] ⇒ Just 3
lookup "four"  [("one", 1), ("two", 2), ("three", 3)] ⇒ Nothing

lookup "path" [("type", "cgi"), ("path", "/var/app")] ⇒ Just "/var/app"
lookup "url"  [("type", "cgi"), ("path", "/var/app")] ⇒ Nothing
```

lookupを組み合わせる

```
config :: [(String, [(String, String)])]
config =
  [ ("database",  [("path", "/var/app/db"), ("encoding", "euc-jp")]),
    ("urlmapper", [("cgiurl", "/app"), ("rewrite", "True")]),
    ("template",  [("path", "/var/app/template")]) ]
```

- lookupの結果にさらにlookupを適用したい.
  - 最初のlookupが成功したかどうかを確認する必要がある.




```
case (lookup "database" config) of {
  Just entries -> lookup "encoding" entries ;
  Nothing      -> Nothing
}
```

モナド則を使う

```
instance Monad Maybe where {
   (Just x) >>= f   = f x ;
   Nothing  >>= f   = Nothing ;
   return x         = Just x
}
```

- Maybeがモナドであることから:

```
case (lookup "database" config) of {
  Just entries -> lookup "encoding" entries ;
  Nothing      -> Nothing }
```

```
lookup "database" config >>= lookup "encoding"
```

```
return config >>= lookup "database" >>= lookup "encoding"
```

練習問題10－1

- 次のプログラムは **x** を2で割るが，偶数でない時には失敗する．

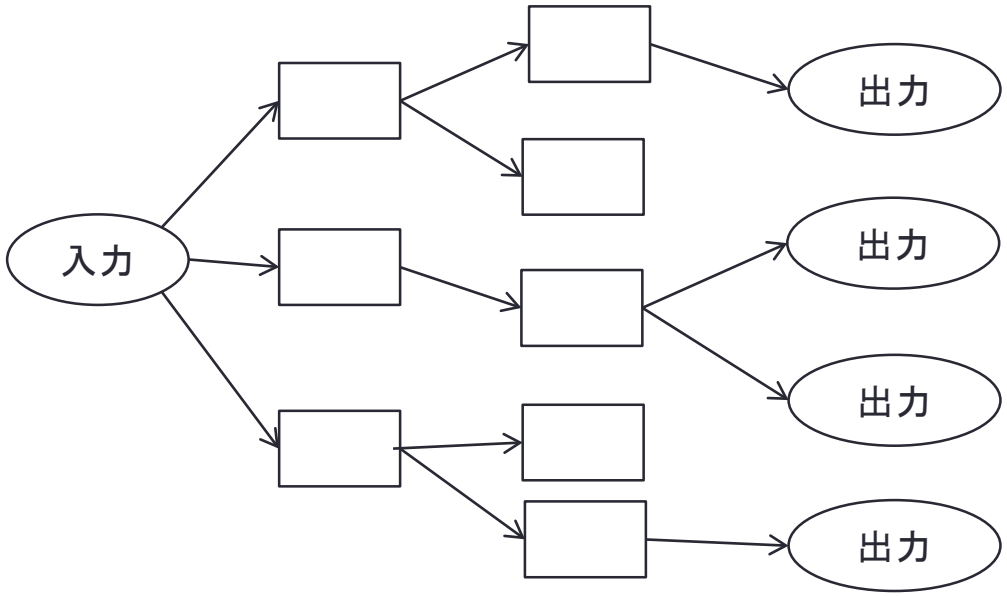
```
div2::Int -> Maybe Int
div2 x = if even x then Just (x `div` 2)
                   else Nothing
```

- 例
  - **div2 4 ⇒ Just 2**
  - **div2 3 ⇒ Nothing**
- **div2** を3回使うことによって，与えられた数字を8で割るが，8で割れない場合には失敗する関数 **div8** を定義しなさい．
  - **div8 24 ⇒ Just 3**
  - **div8 20 ⇒ Nothing**

```
div8::Int -> Maybe Int
div8 x = ...
```

Listモナド

- Maybeモナド
  - 失敗などして値が存在しない場合を扱うことができる.
- Listモナド
  - 扱う値の数が増えたり減ったりする場合を扱う.



Listモナド

```
instance Monad [] where {
  xs >>= f    = concat $ map f xs ;
  return x    = [x]
}
```

- 例
  - ファイル名の展開
    - `expandCharClass "img[012].png"`
      `⇒ ["img0.png", "img1.png", "img2.png"]`
    - `expandAltWorlds "img.{png,jpg}"`
      `⇒ ["img.png", "img.jpg"]`
  - 2つの展開関数を組み合わせる
    - `expandPattern::String -> [String]`
    - `expandPattern pattern`
      `= expandCharClass pattern >>= expandAltWords`
    - `expandPattern "img[012].{png,jpg}"`
      `⇒ ["img0.png", "img0.jpg", "img1.png", "img1.jpg",`
      `"img2.png", "img2.jpg"]`

練習問題10－2

- **replicate** は与えられた要素を与えられた回数コピーしたリストを返す関数です.
  - `replicate :: Int -> a -> [a]`
  - `replicate 3 1 ⇒ [1,1,1]`
  - `replicate 5 'a' ⇒ "aaaaa"`
- **replicate** を使って，与えられたリストのそれぞれの要素を2重にして返す関数 **double** をListモナドを使って定義しなさい.
  - `double [1,2,3] ⇒ [1,1,2,2,3,3]`
  - `double "SFC" ⇒ "SSFFCC"`

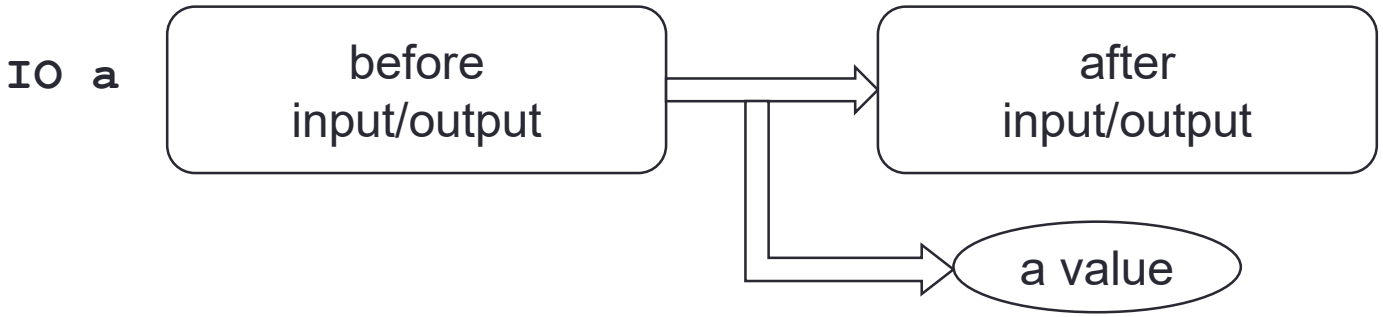
double.hs

```
double xs = ...
```

IOモナド

- 入出力には順番がある.
  - e.g. プロンプトは入力の前に出力する.
  - e.g. "Sunday"を"Monday"の前に出力する.

- 「`IO a`」の値は入出力アクションを表している.



- 「`(>>=)`」および「`return`」はシステムで実装されている.
  - `x >>= y`
  - アクション「`x`」がうまくいった場合には，その結果をアクション「`y`」の渡す.
  - そのため，アクション「`x`」はアクション「`y`」の前に行う必要がある.

IOモナドの例

cat.hs

```
main = getContents >>= putStr
```

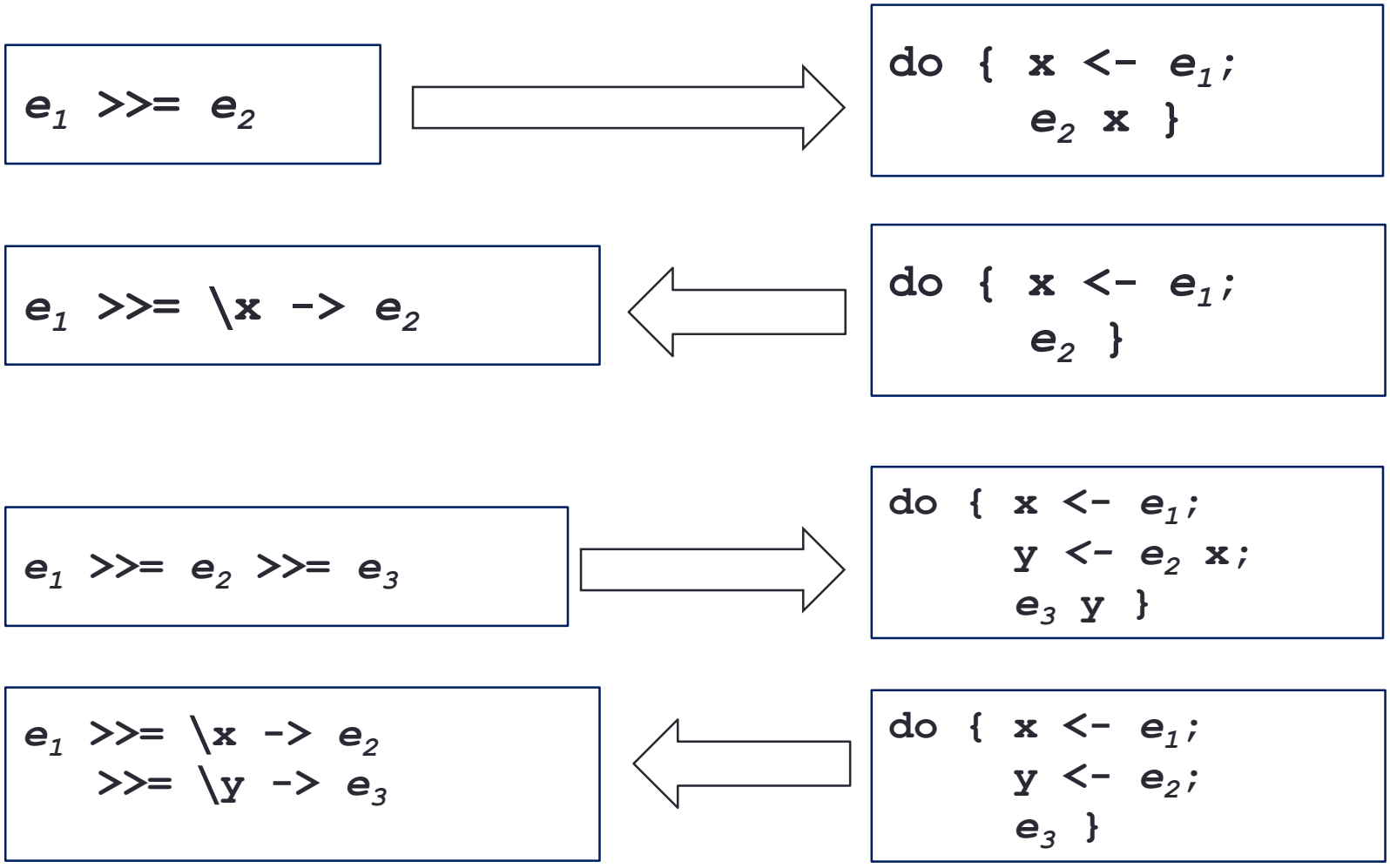
- **getContents::IO String**
  - コンソールから入力するアクション
- **putStr::String -> IO ()**
  - 文字列をコンソールに出力するアクション

- **do** 式を使うと次のように書くこともできる.

```
main = do { cs <- getContents ;
            putStr cs }
```

モナド構文

- do 式


e1 >>= e2
do { x <- e1;
e2 x }
e1 >>= \x -> e2
do { x <- e1;
e2 }
e1 >>= e2 >>= e3
do { x <- e1;
y <- e2 x;
e3 y }
e1 >>= \x -> e2
>>= \y -> e3
do { x <- e1;
y <- e2;
e3 }

IOモナドと (>>)

- 次のdo式は (>>=) を使って書き直すことができる.

```
do { putStrLn "Hello, World!";
     putStrLn "Hello, again!!!" }
```

```
putStrLn "Hello, World!" >>= \x -> putStrLn "Hello, again!!!"
```

- 2つ目の **putStrLn** は変数 **x** を使わないので, **Monad** のクラスメソッド **(>>)** を使うことができる.

```
putStrLn "Hello, World!" >> putStrLn "Hello, again!!!"
```

```
class Monad m where {
  (>>)::m a -> m b -> m b ;
  f >> g = f >>= (\x -> g)
}
```

例(1)

```
nameNoDo::IO ()
nameNoDo = putStr "What is your first name? " >>
           getLine >>= \first ->
           putStr "And your last name? " >>
           getLine >>= \last ->
           let full = first ++ " " ++ last
           in putStrLn ("Please to meet you, " ++ full ++ "!")
```

```
nameDo::IO ()
nameDo = do { putStr "What is your first name? ";
              first <- getLine;
              putStr "And your last name? ";
              last <- getLine;
              let full = first ++ " " ++ last;
              putStrLn ("Please to meet you, " ++ full ++ "!") }
```

例(2)

- **lookup** を二重に行う場合を，do式で書いてみる．

```
case (lookup "database" config) of {
  Just entries -> lookup "encoding" entries ;
  Nothing      -> Nothing }
```

```
lookup "database" config >>= lookup "encoding"
```

```
do { entries <- lookup "database" config ;
     lookup "encoding" entries }
```

練習問題10－3

- 練習問題10－1の **div8** を **do** 式を用いて書きなさい.

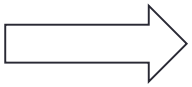
div8.hs

```
import System.Environment

div2::Int -> Maybe Int
div2 x = if even x then Just (x `div` 2)
                   else Nothing

div8::Int -> Maybe Int
div8 x = do { y <- div2 x;
              ...;
              ... }

main = do { args <- getArgs;
            print $ div8 $ read $ head args }
```

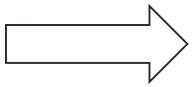
リスト内包表記とListモナド

- Haskellには便利なリスト内包表記があります.
  - `[ x * 2 | x <- [1..5] ]` ⇒ `[2,4,6,8,10]`
  - `[ (x,y) | x <- [1,2], y <- [3,4] ]`
    ⇒ `[(1,3),(1,4),(2,3),(2,4)]`
- これはListモナドの糖衣構文(syntax sugar)です.

```
[ x * 2 | x <- [1..5] ]
```



```
[1..5] >>= \x -> return(x * 2)
```



```
do { x <- [1..5];
     return (x * 2) }
```

```
[ (x,y) | x <- [1,2],
          y <- [3,4] ]
```

```
[1,2] >>= \x ->
[3,4] >>= \y -> return (x,y)
```

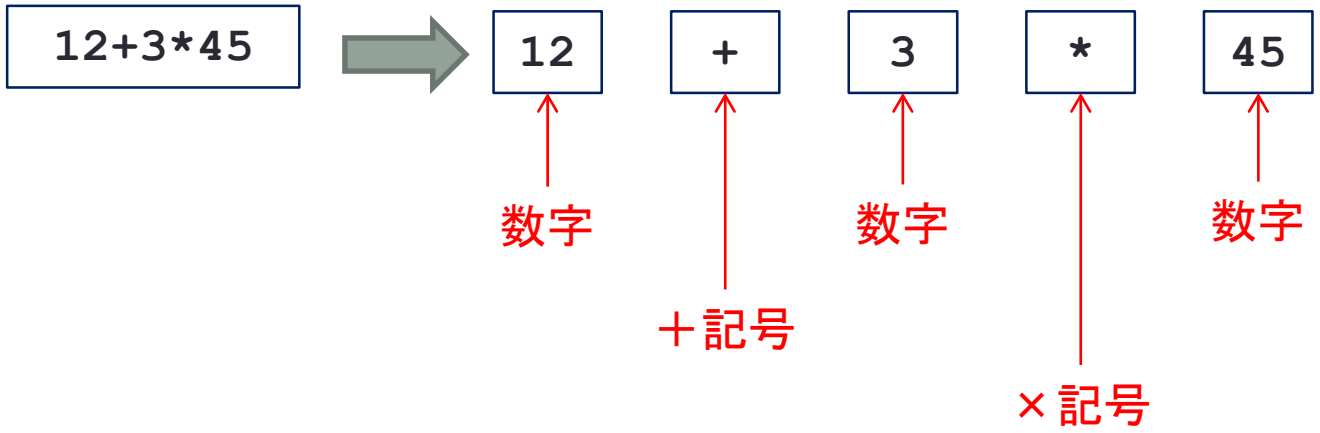
```
do { x <- [1,2];
     y <- [3,4];
     return (x,y)}
```

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.

```
1+2+3+4 ⇒ 10
12+3*45 ⇒ 147
```

- 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10－4

- Tokenが正しく動くかテストしなさい.

token.hs

```
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)

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

- 実行例

```
% stack runghc token.hs
1+2*3
Num 1 Add Num 2 Mul Num 3
```

練習問題10－5

- 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

```
{- token.hs -}
data Token = ...

tokens ...

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 }
```

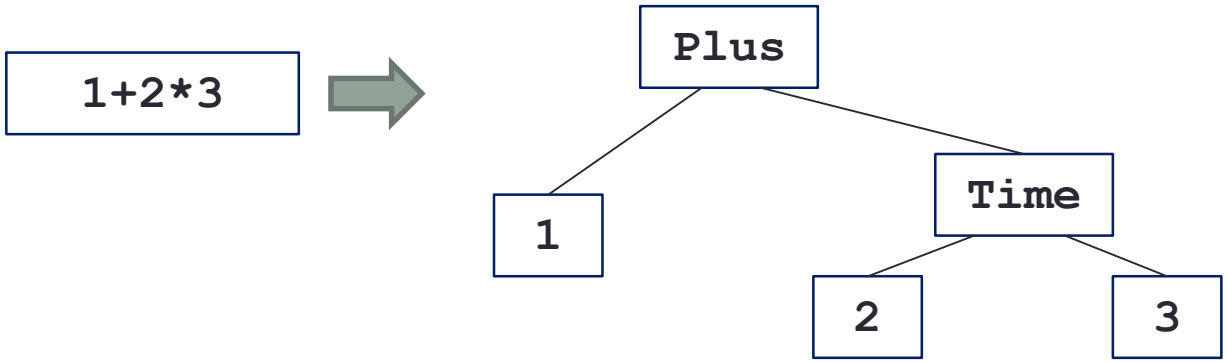
- **実行例**

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

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 ::= number (("*" | "/") number)*
```

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10−6

- 足し算と引き算の式をパースする.

```
data Token = ...

tokens ...

data ParseTree = ... deriving Show

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

parseNumber::Parser
parseNumber(Num x:ts) = (Number x, ts)

parseTerm::Parser
parseTerm ts = nextNumber $ parseNumber ts
  where { nextNumber x = x }

parseExpr::Parser
parseExpr ts = nextTerm $ parseTerm ts
  where { nextTerm(p1, Add:ts1) = let (p2, ts2) = parseTerm ts1
                                  in nextTerm(Plus p1 p2, ts2);
          nextTerm(p1, Sub:ts1) = let (p2, ts2) = parseTerm ts1
                                  in nextTerm(Minus p1 p2, ts2);
          nextTerm x = x }

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

parseExpr の動作

```
parseExpr ts =
  nextTerm $ parseTerm ts
```

```
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,ts2) = parseTerm [Num 2,Add,Num 3]
   in nextTerm(Plus(Number 1) p2, ts2)

⇒ let (p2,ts2) = (Number 2,[Add,Num 3])
   in nextTerm(Plus(Number 1) p2, ts2)

⇒ nextTerm(Plus(Number 1)(Number 2),[Add,Num 3])

⇒ let (p2,ts2) = parseTerm [Num 3]
   in nextTerm(Plus(Plus(Number 1)(Number 2)) p2,ts2)

⇒ let (p2,ts2) = (Number 3,[])
   in nextTerm(Plus(Plus(Number 1)(Number 2)) p2,ts2)

⇒ nextTerm(Plus(Plus(Number 1)(Number 2))(Number 3),[])

⇒ (Plus(Plus(Number 1)(Number 2))(Number 3),[])
```

```
nextTerm(p1, Add:ts1) =
  let (p2, ts2) = parseTerm ts1
  in nextTerm(Plus p1 p2, ts2)
```

```
nextTerm x = x
```

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(Times p1 p2) = ...
eval(Divide p1 p2) = ...
```

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

練習問題10－7

- 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．

calc.hs

```
import Data.Char

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

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

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

parseNumber = ...
parseTerm = ...
parseExpr = ...

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

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

実行例

```
% stack runghc calc
2+3*4
14
4/3*5
5
5-3-4
-2
```

練習問題10－8

- この電卓では，0での割り算を行う式を入力するとおかしなことになります．

```
eval::ParseTree -> Int
eval(Divide p1 p2) = ....
```

- **eval** を **Int** を返す関数としてではなく **Maybe Int** を返す関数として定義し，0での割り算があった時にはエラーメッセージを出力して，次の入力を受け付けるようにしなさい．
  - `eval::ParseTree -> Maybe Int`

calc.hs

```
import Data.Char
data Token = Num Int | Add | Sub | Mul | Div
data ParseTree = ...
type Parser = [Token] -> (ParseTree, [Token])
...

eval::ParseTree -> Maybe Int
eval(Number x) = Just x
...
eval(Divide p1 p2) = ...

showResult::Maybe Int -> String
showResult Nothing = "error: division by 0"
showResult Just x = show x

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

実行例

```
% stack runghc calc.hs
5+4*3/2
11
5+2/0-3
error: division by 0
1+2-3*4/5
1
```