パーサ言語

単純なパーサを合成した複雑なパーサを組み立てたい

- 基本パーサを構成する手段
- パーサ集合上の演算

パーサ「言語」を構成する要件

パーサ関数の型

型から考えます.

```
type Parser0 = [Token] -> Value
```

トークンの型とパース結果の値の型をパラメータとすると,

```
type Parser1 t v = [t] -> v
```

パーサ関数の型

結果には残りのトークン列が含まれて欲しい.

```
type Parser2 t v = [t] -> (v, [t])
```

複数のパース方法があるかもしれないし，入力のトークン列が文法を満していないかもしれない.

```
type Parser3 t v = [t] -> [(v, [t])]
```

パーサ関数の型

トークンは単純化して文字 (Char) とする

```haskell
type Parser a = [Char] -> [(a, [Char])]
```

String は [Char] の型シノニムなので

```haskell
type Parser a = String -> [(a, String)]
```

基本パーサ

常に成功するパーサ

```
punit :: a -> Parser a
punit c s = [(c, s)]
```

常に失敗するパーサ

```
pfail :: Parser a
pfail s = []
```

基本パーサ

条件を満すトークンを構成するパーサ

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy p (c:cs) | p c = [(c,cs)]
satisfy _ _            = []
```

指定したトークンを構成するパーサ

```
char :: Char -> Parser Char
char c = satisfy (c ==)
```

基本パーサ演算

パーサ変換，パーサ適用，パーサ選択

```
(<$>) :: (a -> b) -> (Parser a -> Parser b)
(f <$> p) s = [ (f x, t) | (x, t) <- p s ]

(<*>) :: Parser (a -> b) -> (Parser a -> Parser b)
(p <*> q) s = [ (f x, u) | (f, t) <- p s, (x, u) <- q t ]

(+++) :: Parser a -> Parser a -> Parser a
(p +++ q) s = p s ++ q s
```

基本パーサ演算

```
many :: Parser a -> Praser [a]
many p = pUnit [] +++ many1 p

many1 :: Parser a -> Praser [a]
many1 p = (:) <$> p <*> many p

pair :: Parser a -> Parser b -> Parser (a, b)
pair p q = (,) <$> p <*> q

eof :: Parser ()
eof "" = [((), "")]
eof _  = []
```

算術演算式のパーサ

具象構文（左再帰除去済み文法）

```
expression ::= additive;
additive   ::= multitive ('+' multitive | '-' multitive)*;
multitive  ::= primary ('*' primary | '/' primary)*;
primary    ::= '(' expression ')' | number;
number     ::= [0-9]+;
```

算術演算式のパーサ

抽象構文木

```
data Expr = ENum Int
          | EBinOp Expr Op Expr

data Op = Plus | Minus | Times | Divide
```

算術演算のパーサ

```haskell
pExpression :: Parser Expr
pExpression = pAdditive

pAdditive :: Parser Expr
pAdditive
 = mkBinOp <$> pMultitive <*> many (pair pAdd pMultitive)
   where pAdd = pPlus +++ pMinus

pMultitive :: Parser Expr
pMultitive
 = mkBinOp <$> pPrimary <*> many (pair pMul pPrimary)
   where pMul = pTimes +++ pDivide
```

算術演算のパーサ

```
pPrimary :: Parser Expr
pPrimary = between (char '(') (char ')') pExpression
       +++ pNumber

pNumber :: Parser Expr
pNumber = between (ENum . read) <$> many1 (satisfy isDigit)

pPlus, pMinus, pTimes, pDivide :: Parser Op
pPlus   = const Plus   <$> char '+'
pMinus  = const Minus  <$> char '-'
pTimes  = const Times  <$> char '*'
pDivide = cosnt Divide <$> char '/'
```

二項演算項の構成

```haskell
mkBinOp :: Expr -> [(Op, Expr)] -> Expr
mkBinOp = foldl (uncurry . EBinOp)
```

算術式の読込

```haskell
instance Read Expr where
  readsPrec _ = const <$> pExpression <*> eof
```

算術式の表示

```
instance Show Expr where
  show e = case e of
    ENum n           -> show n
    EBinOp e1 op e2 -> "("++show e1++show op++show e2++")"

instance Show Op where
  show Plus   = "+"
  show Minus  = "-"
  show Times  = "*"
  show Divide = "/"
```

算術式の評価

```haskell
eval :: Expr -> Int
eval (ENum n) = n
eval (EBinOp e1 op e2) = evalop op (eval e1) (eval e2)

evalop :: Op -> (Int -> Int -> Int)
evalop Plus   = (+)
evalop Minus  = (-)
evalop Times  = (*)
evalop Divide = div
```