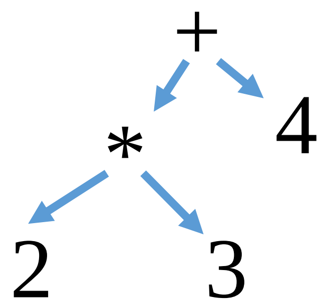
Lesson 12

monadic parsing

a parser is a program that takes a string of characters as input and produces some form of a tree

«2*3+4»



* has higher priority than + and both have arity 2

many parsers:

-real life programs parse their inputs

-GHC parses the Haskell programs

parsers as functions

we assume to have some general type Tree

type Parser = String -> Tree

more convenient

type Parser = String -> (Tree, String)

but parsers may fail

type Parser = String -> [(Tree, String)] and [] = failure different parsers return different type of trees

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

observe the similarity with ST, the difference is [] = failure

```
import Control. Applicative
import Data.Char
```

```
newtype Parser a = P (String -> [(a, String)])
parse :: Parser a -> String -> [(a,String)]
parse (P p) inp  = p inp
```

first basic parser

```
item :: Parser Char
item = P (\inp -> case inp of
                    []        ->   []
                    (x:xs)   -> [(x,xs)]
```

consumes one character

all other parsers follow from it

parse item «» ➔ []

parse item «abc» ➔ [(‘a’, «bc»)]

we make Parser a Functor, an Applicative and a Monad

```
instance Functor Parser where
--fmap :: (a -> b) -> Parser a -> Parser b
fmap  g  p = P (\inp -> case parse p inp of
                            []           ->   []
                            [(v,out)]  -> [(g v, out)]
```

parse (fmap toUpper item) «abc» ➔ [(‘A’, «bc»)]

parse (fmap toUpper item) «»   ➔ []

toUpper comes from Data.Char

```
instance Applicative Parser where
--pure :: a -> Parser a
pure v = P (\inp -> [(v,inp)]

--(<*>) :: Parser (a -> b) -> Parser a -> Parser b
pg <*> px = P (\inp -> case parse pg inp of
                          []            ->  []
                          [(g, out)]  -> parse (fmap g px) out)

parse (pure 1)  «abc»  ➔ [(1, «abc»)]
```

```
three :: Parser (Char, Char)

three = pure g <*> item <*> item <*> item
        where g x y z  = (x,z)
```

parse three «abcdef» ➔ [((‘a’,’c’), «def»)]

parse three «ab» ➔ []

```
instance Monad Parser where
--(>>=) :: Parser a -> (a -> Parser b) -> Parser b
p >>= f = P (\inp -> case parse p inp of
                        []                -> []
                        [(v,out)]         -> parse (f v) out )

three :: Parser (Char, Char)
three = do x <- item
           item
           z <- item
           return (x,z)
```

making choices

```
class Applicative f => Alternative f where
  empty :: f a
 (<|>) :: f a -> f a -> f a
```

they must satisfy the laws:

```
empty <|> x  =  x
x <|> empty = x
x <|> (y <|> z)   =  (x <|> y ) <|> z
```

```
instance Alternative Maybe where
--empty :: Maybe a
empty = Nothing

--(<|>) :: Maybe a -> Maybe a -> Maybe a
Nothing <|> my  = my
(Just x)  <|> _      = Just x
```

```
instance Alternative Parser where
--empty :: Parser a
empty = P (\inp -> [])

--(<|>) :: Parser a -> Parser a -> Parser a
p <|> q = P (\inp -> case parse p inp of
                         []              ->   parse  q  inp
                         [(v,out)]       -> [(v,out)]
```

parse empty «abc» ➔ []

parse (item <|> return ‘d’) «abc» ➔ [(‘a’, «bc»)] parse (empty <|> return ‘d’) «abc» ➔ [(‘d’, «abc»)]

Derived parsers

we have 3 parsers: item, return v and empty

we make new ones:

```
sat :: (Char -> Bool) -> Parser Char
sat p = do x <- item
           if p x then return x else empty
```

```
digit :: Parser Char
digit = sat isDigit

lower :: Parser Char
lower = sat isLower

upper :: Parser Char
upper = sat isUpper

letter :: Parser Char
letter = sat is Alpha
```

```
alphanum :: Parser Char
alphanim = sat isAlphaNum

char :: Char -> Parser Char
char x = sat (==x)
```

examples:

parse (char ‘a’) «abc» ➔ [(‘a’, «bc»)]

parse (char ‘b’) «abc» ➔ []

```
string :: String -> Parser String
string []       =  return []
string (x:xs) =  do char x
                    string xs
                    return (x:xs)
```

parse (string «abc») «abcdef» ➔ [(«abc», «def»)]

parse (string «abc») «ab1234» ➔ []

many and some

parse (many digit) «123abc» ➔ [(«123», «abc»)]

parse (many digit) «abc» ➔ [(«», «abc»)]

parse (some digit) «abc» ➔ []

are operators of Alternative

```
class Applicative f => Alternative f where

empty :: f a
(<|>) :: f a -> f a -> f a
many :: f a -> f [a]
some :: f a -> f [a]

many x = some x <|> pure []
some x = pure (:) <*> x <*> many x
```

more parsers:

```
ident :: Parser String
ident = do x <- lower
           xs <- many alphanum
           return (x:xs)

nat :: Parser Int
nat = do xs <- some digit
         return (read xs)
```

```
space :: Parser ()
space = do many (sat isSpace)
           return ()
```

parse ident «abc def» ➔ [(«abc», « def»)]

parse nat «123 abc» ➔ [(123, « abc»)]

parse space «   abc» ➔ [((), «abc»)]

a parser for integer numbers:

```
int :: Parser Int

int = do char '-'
         n <- nat
         return (-n)
      <|> nat
```

parse int «-123 abc» ➔ [(-123, « abc»)]

handling spacing

```
token :: Parser a -> Parser a
token p = do space
             v <- p
             space
             return v

identifier :: Parser String
identifier = token ident

natural :: Parser Int
natural = token nat
```

```
integer :: Parser Int
integer = token int

symbol :: String -> Parser String
symbol xs = token (string xs)
```

```
nats :: Parser [Int]
nats = do symbol «[»
          n <- natural
          ns <- many ( do symbol «,»
                          natural)
          symbol «]»
          return (n:ns)
```

parser for arithmetic expressions

expr ::= term (+ expr | epsi)

term ::= factor (* Term | epsi)

factor ::= (expr) | nat

nat ::= 0 | 1 |...

```
expr :: Parser Int
expr = do t <- term
          do symbol «+»
             e <- expr
             return (t + e)
          <|> return t

term :: Parser Int
term = do f <- factor
          do symbol «*»
             t <- term
             return (f * t)
          <|> return f
```

```
factor :: Parser Int
factor = do symbol «(«
            e <- expr
            symbol «)»
            return e
         <|> natural

eval :: String -> Int
eval xs = case (parse expr xs) of
        [(n,[])]   -> n
        [(_,out)] -> error («Unused input»++out)
        []           ->error «Invalid input»
```