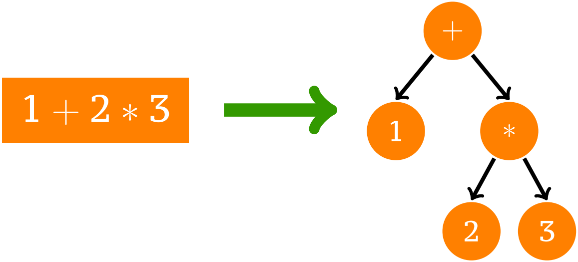
Functional parsers

A *parser* checks whether the input string is syntactically correct (according to a grammar) and constructs a corresponding abstract syntax tree.



Functional parsers

Type of parsers

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

Applying a parser

```
runParser         ::  Parser a -> String -> [(a, String)]
runParser p str  =    p str
```

Functional parsers

Primitive parsers

```
failp    ::  Parser a
failp    =   \str ->  []

succp    ::  a -> Parser a
succp x  =   \str ->  [(x, str)]

symp     ::  Char -> Parser Char
symp x   =   \str ->  case str of
                        []     -> []
                        c:cs   -> [(c,cs) | c == x]
```

Functional parsers

Sequential composition

```
infixr 6 <*>
(<*>)       ::  Parser a -> Parser b -> Parser (a,b)
p1 <*> p2   =   \str -> [((v1,v2),cs2) |  (v1,cs1) <- p1 str,
                                          (v2,cs2) <- p2 cs1]
```

Parallel composition

```
infixr 4 <|>
(<|>)       ::  Parser a -> Parser a -> Parser a
p1 <|> p2   =   \str -> p1 str ++ p2 str
```

Functional parsers

Manipulating values

```
infixr 5 <@
(<@)     ::  Parser a -> (a -> b) -> Parser b
p <@ f   =   \str -> [(f v, cs) | (v,cs) <- p str]
```

Example

```
data Tree = Nil | Bin Tree Tree

parens :: Parser Tree
parens =       symp '(' <*>  parens <*>  symp ')' <*>  parens
                                  <@ (\ (_,(x,(_,y))) -> Bin x y)
           <|> succp Nil
```

Parser monad

Type of parsers

```
newtype Parser a = P {runP :: (String -> [(a,String)])}
```

Parser monad

```
instance Monad Parser where
   return a = P $ \str -> [(a, str)]
   p >>= f  = P $ \str -> concat [runP (f a) cs |
                                  (a,cs) <- runP p str]

instance MonadPlus Parser where
   mzero     = P $ \str -> []
   mplus p q = P $ \str -> runP p str ++ runP q str
```

Parser monad

Primitive combinators

```
item :: Parser Char
item = P $ \str -> [(head str, tail str) | not (null str)]

first     ::  Parser a -> Parser a
first p   =   P $ \str ->  case runP p str of
                            []       ->  []
                            (x:xs)   ->  [x]
```

Simple parser combinators

Derived combinators

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

char        ::   Char -> Parser Char
char c      =    sat (c ==)

(<|>)       ::   Parser a -> Parser a -> Parser a
p <|> q     =    first (p `mplus` q)
```

Simple parser combinators

Derived combinators

Example

```
parens   ::   Parser Tree
parens   =    do    char '('
                    t1   <- parens
                    char ')'
                    t2   <- parens
                    return (Bin t1 t2)
              <|>   return Nil
```

```
(<|>)     ::   Parser a -> Parser a -> Parser a
p <|> q   =    first (p `mplus` q)
```

Simple parser combinators

Iteration

```
many      ::  Parser a -> Parser [a]
many p    =   many1 p <|> return []

many1     ::  Parser a -> Parser [a]
many1 p   =   do a <- p
                 as <- many p
                 return (a:as)
```

Simple parser combinators

Keywords

```
string         :: String -> Parser String
string ""      =  return ""
string (c:cs)  =  do char c
                     string cs
                     return (c:cs)
```

Identifiers

```
identifier :: Parser String
identifier =  do c   <-  lower
                 cs  <-  many alphanum
                 return (c:cs)
```

Simple parser combinators

Natural numbers

```
digit   :: Parser Int
digit   =  do c <- sat isDigit
              return (ord c - ord '0')

natural :: Parser Int
natural =  do ds <- many1 digit
              return (foldl1 (\a b -> 10*a + b) ds)
```

Integers

```
integer :: Parser Int
integer =      do {char '-'; n <- natural; return (-n)}
           <|> natural
```

Simple parser combinators

Floating point numbers

```
fraction :: Parser Double
fraction =  do char '.'
               ds <- many1 digit
               return (foldr op 0 ds)
   where d `op` x = (x + fromIntegral d)/10

floating :: Parser Double
floating =  do i  <-  integer
               f  <-  fraction <|> return 0
               return (fromIntegral i + f)
```

Simple parser combinators

Spaces

```
space   :: Parser String
space   = many (sat isSpace)

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

keyw cs = token (string cs)
ident   = token identifier
nat     = token natural
int     = token integer
float   = token floating
```

Simple parser combinators

Bracketing constructions

```
pack :: Parser a -> Parser b -> Parser c -> Parser b
pack s1 p s2 = do s1
                  x <- p
                  s2
                  return x

paren p      = pack (keyw "(") p (keyw ")")
brack p      = pack (keyw "[") p (keyw "]")
block p      = pack (keyw "begin") p (keyw "end")
```

Simple parser combinators

Sequences

```
sepby           ::  Parser a -> Parser b -> Parser [a]
p `sepby` sep   =   (p `sepby1` sep) <|> return []

sepby1          ::  Parser a -> Parser b -> Parser [a]
p `sepby1` sep  =   do  a    <-  p
                        as   <-  many (sep >> p)
                        return (a:as)

commaList p     =   sepby p (keyw ",")
semicList p     =   sepby p (keyw ";")
```

Simple parser combinators

Sequences

```
chainl  :: Parser a -> Parser (a->a->a) -> Parser a
chainl p s = do
        x   <-  p
        ys  <-  many (do {op <- s; y <- p; return (op,y)})
        return (foldl (\a (op,y) -> a `op` y) x ys)

chainr  :: Parser a -> Parser (a->a->a) -> Parser a
chainr p s = do
        ys  <-  many (do {y <- p; op <- s; return (y,op)})
        x   <-  p
        return (foldr (\(y,op) b -> y `op` b) x ys)
```

Simple parser combinators

Parsing the whole input

```
parse :: Parser a -> String -> a
parse p cs   =  case  runP (first (space >> p)) cs of
                      [(x,"")]   ->   x
                      _          ->   error "Parse error"
```

Example: parsing arithmetic expressions

Arithmetic expressions

Grammar

```
expr  =   int              |   expr  +  expr   |   expr  -  expr
      |   expr * expr      |   expr  /  expr   |   expr  ^  expr
      |   (expr)
```

Abstract syntax tree

```
data Expr  =  Num Int
           |  Expr  :+:  Expr
           |  Expr  :-:  Expr
           |  Expr  :*:  Expr
           |  Expr  :/:  Expr
           |  Expr  :^:  Expr
```

Arithmetic expressions

Parser ver. 0

```
expr0 =       do  {  e0  <-  expr0; keyw "+";
                     e1  <-  expr0; return(e0 :+: e1)}
          <|> do  {  e0  <-  expr0; keyw "-";
                     e1  <-  expr0; return(e0 :-: e1)}
          <|>                ...
          <|> do  {  e0  <-  expr0; keyw "^";
                     e1  <-  expr0; return(e0 :^: e1)}
          <|> do  {i <- int; return (Num i)}
          <|> paren expr0
```

Arithmetic expressions

Parser ver. 0

```
expr0 =     do  {  e0  <-  expr0; keyw "+";
                   e1  <-  expr0; return(e0 :+: e1)}
```

NB!

Doesn't work, as the grammar is left recursive!!

```
                   e1  <-  expr0; return(e0 :^: e1)}
        <|> do  {i <- int; return (Num i)}
        <|> paren expr0
```

Arithmetic expressions

Parser ver. 1

```
expr1  =  do    a    <- atom1
                op   <- oper1
                e    <- expr1
                return (a `op` e)
          <|>   atom1

oper1  =        (keyw  "+"   >>  return (:+:))
          <|>   (keyw  "-"   >>  return (:-:))
          <|>   (keyw  "*"   >>  return (:*:))
          <|>   (keyw  "/"   >>  return (:/:))
          <|>   (keyw  "^"   >>  return (:^:))

atom1  =  do    {i <- int; return (Num i)}
          <|>   paren expr1
```


Arithmetic expressions

Parser ver. 1

```
expr1  =  do   a   <- atom1
               op  <- oper1
               e   <- expr1
```

NB!

"Works" but doesn't take account operators
priorities and associativities!

```
oper
          <|>  (keyw  "/"  >>  return (:/:))
          <|>  (keyw  "^"  >>  return (:^:))

atom1  =  do   {i <- int; return (Num i)}
          <|>  paren expr1
```

Arithmetic expressions

Parser ver. 2

```
expr2  ::  Parser Expr
expr2  =   chainl term2  (     (keyw "+"  >>  return (:+:))
                         <|>   (keyw "-"  >>  return (:-:)))

term2  ::  Parser Expr
term2  =   chainl fact2  (     (keyw "*"  >>  return (:*:))
                         <|>   (keyw "/"  >>  return (:/:)))

fact2  ::  Parser Expr
fact2  =   chainr atom2  (keyw "^" >> return (:^:))

atom2  ::  Parser Expr
atom2  =   do   {i <- int; return (Num i)}
           <|>  paren expr2
```

Arithmetic expressions

Evaluator

```
expr3  ::  Parser Int
expr3  =   chainl term3  (    (keyw "+"  >>  return (+))
                         <|>  (keyw "-"  >>  return (-)))

term3  ::  Parser Int
term3  =   chainl fact3  (    (keyw "*"  >>  return (*))
                         <|>  (keyw "/"  >>  return div))

fact3  ::  Parser Int
fact3  =   chainr atom3 (keyw "^" >> return (^))

atom3  ::  Parser Int
atom3  =   int <|> paren expr3
```

Applicative parsing: `ParserA.hs`

Parsec

```
data ParsecT s u m a
```

is a parser with

- stream type `s`,
- user state type `u`,
- underlying monad `m`, and
- return type `a`.

Type

```haskell
data ParsecT s u m a
  = ParsecT {unParser :: forall b .
                State s u
             -> (a -> State s u -> ParseError -> m b) -- consumed ok
             -> (ParseError -> m b)                   -- consumed err
             -> (a -> State s u -> ParseError -> m b) -- empty ok
             -> (ParseError -> m b)                   -- empty err
             -> m b
            }
```

Parsec

- Text.Parsec

```
runParser :: Stream s Identity t => Parsec s u a -> u -> SourceName -> s
                                                   -> Either ParseError a

(<|>) :: ParsecT s u m a -> ParsecT s u m a -> ParsecT s u m a
try   :: ParsecT s u m a -> ParsecT s u m a
many  :: Stream s m t => ParsecT s u m a -> ParsecT s u m [a]
many1 :: Stream s m t => ParsecT s u m a -> ParsecT s u m [a]
sepBy :: Stream s m t => ParsecT s u m a -> ParsecT s u m sep -> ParsecT s u m [a]
```

- Text.Parsec.Char

```
letter  :: Stream s m Char => ParsecT s u m Char
string  :: Stream s m Char => String -> ParsecT s u m String
anyChar :: Stream s m Char => ParsecT s u m Char
oneOf   :: Stream s m Char => [Char] -> ParsecT s u m Char
satisfy :: Stream s m Char => (Char -> Bool) -> ParsecT s u m Char
```

Monad Transformer

```
runParserT :: Stream s m t => ParsecT s u m a -> u -> SourceName
              -> s -> m (Either ParseError a)

class MonadTrans (t :: (* -> *) -> * -> *) where
  lift :: Monad m => m a -> t m a

instance MonadTrans (ParsecT s u)
```

Monad Transformer

```
runParserT :: Stream s m t => ParsecT s u m a -> u -> SourceName
              -> s -> m (Either ParseError a)

class MonadTrans (t :: (* -> *) -> * -> *) where
  lift :: Monad m => m a -> t m a

instance MonadTrans (ParsecT s u)
```

Example

```
okP = do
  string "ok"
  lift $ putStrLn "read ok!"

parseOk = runParserT okP () "test source" "ok"
```

Monad Transformer

```
runParserT :: Stream s m t => ParsecT s u m a -> u -> SourceName
              -> s -> m (Either ParseError a)

class MonadTrans (t :: (* -> *) -> * -> *) where
  lift :: Monad m => m a -> t m a

instance MonadTrans (ParsecT s u)
```

Example

```
okP = do
  string "ok"
  lift $ putStrLn "read ok!"

parseOk = runParserT okP () "test source" "ok"
```

```
*Hw4_2> parseOk
read ok!
Right ()
```

State

```
getState    :: Monad m => ParsecT s u m u
putState    :: Monad m => u -> ParsecT s u m ()
modifyState :: Monad m => (u -> u) -> ParsecT s u m ()
```

State

```
getState    :: Monad m => ParsecT s u m u
putState    :: Monad m => u -> ParsecT s u m ()
modifyState :: Monad m => (u -> u) -> ParsecT s u m ()
```

Example

```
stateP :: ParsecT [Char] Int Identity Int
stateP = do
  x <- getState
  putState 10
  return x

parseSt = runParser stateP 5 "test source" "ok"
```

State

```
getState    :: Monad m => ParsecT s u m u
putState    :: Monad m => u -> ParsecT s u m ()
modifyState :: Monad m => (u -> u) -> ParsecT s u m ()
```

Example

```
stateP :: ParsecT [Char] Int Identity Int
stateP = do
  x <- getState
  putState 10
  return x

parseSt = runParser stateP 5 "test source" "ok"
```

```
*Hw4_2> parseSt
Right 5
```