

# The Parser Type

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

```
instance Monad Parser where
    return a = Parser ( \cs -> [(a,cs)] )
    p >=> f = Parser ( \cs -> do { (a,cs') <- parse p cs; parse (f a) cs' } )

(+++) :: Parser a -> Parser a -> Parser a
p +++ q = Parser ( \cs -> case parse p cs of
    [] -> []
    x:xs -> [x] )

zero :: Parser a
zero = Parser ( \cs -> [] )
```

# Combinators - Char and String

```
item :: Parser Char
item = Parser f
  where f [] = []
        f (c:cs) = [(c,cs)]

sat :: (Char -> Bool) -> Parser Char
sat p = do { c <- item; if p c then return c else zero }

sat' :: (Char -> Bool) -> Parser String
sat' p = do { x <- sat p; return [x] }

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

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

next :: String -> Parser String
next cs = string cs +++ (item >> next cs)
```

# Combinators - Applying a Parser

```
space :: Parser String
space = sat' Char.isSpace
```

```
comments :: Parser String
comments = (string "/*" >> next "*/") +++ (string "//" >> next "\n")
```

```
white :: Parser String
white = asterisk (space +++ comments) >=> return . concat
```

```
token :: Parser a -> Parser a
token p = do { a <- p; white; return a }
```

```
apply :: Parser a -> String -> [(a,String)]
apply p = parse (white >> p)
```

# Combinators - Parsing Sequence

```
asterisk :: Parser a -> Parser [a]
asterisk p = plusSign p +++ return []

plusSign :: Parser a -> Parser [a]
plusSign p = do { x <- p; xs <- asterisk p; return (x:xs) }

-- ( (a op a) op a ) op a
lass :: Parser a -> Parser op -> (a -> b) -> (b -> op -> a -> b) -> Parser b
lass a op single cons = a >>= rest . single
    where rest x = ( do
                        y <- op
                        z <- a
                        rest $ cons x y z ) +++ return x

-- a op ( a op (a op a) )
rass :: Parser a -> Parser op -> (a -> b) -> (a -> op -> b -> b) -> Parser b
rass a op single cons = a >>= rest
    where rest x = ( do
                        y <- op
                        z <- a
                        r <- rest z
                        return $ (cons x y r) ) +++ return (single x)
```

# Parsing Arithmetic Expressions - Operators and Ints

```
data AddOp    = Plus | Minus
```

```
plus         = token $ string "+"  >> return Plus
```

```
minus        = token $ string "-"  >> return Minus
```

```
addOp        = plus +++ minus
```

```
data MulOp    = Times | Slash | Modulo
```

```
times        = token $ string "*"  >> return Times
```

```
slash        = token $ string "/"  >> return Slash
```

```
modulo       = token $ string "%"  >> return Modulo
```

```
mulOp        = times +++ slash +++ modulo
```

```
data BasicExpr = Bool Bool | Int Int | Nul | ...
```

```
int = token $ plusSign (sat Char.isDigit) >=> return . Int . read
```

# Parsing Arithmetic Expressions - Nodes in AST

```
data AddExpr = AMul MulExpr  
             | AddExpr AddExpr AddOp MulExpr
```

```
addExpr = lass mulExpr addOp AMul AddExpr
```

```
data MulExpr = AUnary UnaryExpr  
             | MulExpr MulExpr MulOp UnaryExpr
```

```
mulExpr = lass unaryExpr mulOp AUnary MulExpr
```

# Parsing Arithmetic Expressions - An Example

## Terminal

```
ghci> fst . head $ apply addExpr "10 / 3 - 4 * 5 % 7"
```

```
(AddExpr  
  (MulExpr  
    (int 10)  
    (op "/")  
    (int 3)  
  )  
  (op "-")  
  (MulExpr  
    (MulExpr  
      (int 4)  
      (op "*")  
      (int 5)  
    )  
    (op "%")  
    (int 7)  
  )  
)
```