The Parser Type

```haskell
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 ++ parse q cs of
                  [] -> []
                  x:xs -> [x] )

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

Combinators - Char and String

```haskell
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

```haskell
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

```haskell
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

```haskell
data AddOp    = Plus | Minus

plus      = token $ string "+"  >> return Plus
minus     = token $ string "-"  >> return Minus

data MulOp    = Times | Slash | Modulo

times     = token $ string "*"  >> return Times
slash     = token $ string "/"  >> return Slash
modulo    = token $ string "%"  >> return Modulo

data UnaryOp  = No | Nega

no        = token $ string "!"  >> return No
nega      = token $ string "-"  >> return Nega

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
data MulExpr   = AUnary UnaryExpr      | MulExpr MulExpr MulOp UnaryExpr
data UnaryExpr = APrimary PrimaryExpr | UnaryExpr UnaryOp UnaryExpr

data PrimaryExpr = ACall Call
                 | Calls PrimaryExpr Dot Call

data Call = Call BasicExpr (Maybe Calling)
```