Lexical Analysis — Haskell Example

```
module SimpleLexer where
import Char

data Token = Number Integer
           | Sep Char
           | Ident String        deriving ( Show )

simpleLexer :: String → [ Token ]
simpleLexer ( c : cs )
  | isDigit c  =  lexNumber [ c ] cs
  | isAlpha c  =  lexIdent [ c ] cs
  | isSep  c   =  Sep c : simpleLexer cs
  | isSpace c  =  simpleLexer cs
  | otherwise  =  error ("simpleLexer: illegal character: " ++ take 20 ( c : cs ) )
simpleLexer [ ] = [ ]

lexNumber , lexIdent :: String → String → [ Token ]
lexNumber prefix ( c : cs ) | isDigit c     =  lexNumber ( prefix ++ [ c ] ) cs
lexNumber prefix s                          =  Number ( read prefix ) : simpleLexer s
lexIdent  prefix ( c : cs ) | isAlphaNum c  =  lexIdent ( prefix ++ [ c ] ) cs
lexIdent  prefix s                          =  Ident prefix : simpleLexer s

isSep c = c `elem` "(){};,+-*/"
```


Abstract Syntax Example — Haskell

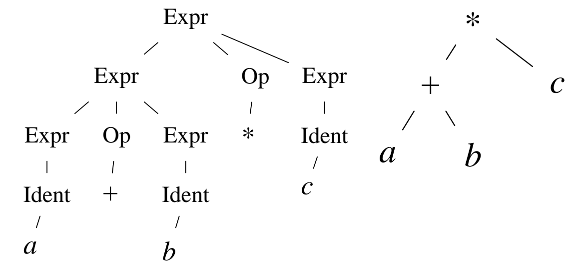
Expr = *Ident* | *Number* | *Expr Op Expr*

```
data Op = MkOp String

data Expr
  = Var String
  | Num Integer
  | Bin Expr Op Expr
```


Abstract Syntax Example — Haskell

Expr = *Ident* | *Number* | *Expr Op Expr*

```
data Op = MkOp String
  deriving Show
data Expr
  = Var String
  | Num Integer
  | Bin Expr Op Expr
 deriving Show

expr1 = Bin
        ( Bin ( Var "a")
           ( MkOp "+")
           ( Var "b") )
        ( MkOp "*")
        ( Var "c")
```




```
plus x y = Bin x ( MkOp "+") y
mult x y = Bin x ( MkOp "*") y

expr2 = ( Var "a" `plus` Var "b") `mult` Var "c"
```


Showing *Expr*

```
showExpr :: Expr → String
showExpr ( Var v ) = v
showExpr ( Num n ) = show n
showExpr ( Bin e1 op e2 ) =
  '(' : showExpr e1 ++ showOp op ++ showExpr e2 ++ ")"

showOp :: Op → String
showOp ( MkOp s ) = s
```

Type of One Kind of Simple Parsers in Haskell

```
parseExpr :: [Token] → Maybe (Expr, [Token])
```

A simple parser for N takes a token stream and returns:

- *Nothing* if no N could be parsed
- *Just* (*x* , *toks*) if the N value *x* could be parsed, leaving the tokens in *toks* unconsumed

Each parser function corresponds to a **concrete** nonterminal — (*parseExpr*, *parseTerm*, *parseFactor*)

Each data type corresponds to an **abstract** nonterminal — (*Op*, *Expr*)

Parsing *Factor* in Haskell

$$Factor \;\rightarrow\; Ident \;\mid\; Number \;\mid\; (\,Expr\,)$$

```
parseFactor :: [Token] → Maybe (Expr, [Token])
parseFactor ((Ident v)  : rest) = Just (Var v, rest)
parseFactor ((Number n) : rest) = Just (Num n, rest)
parseFactor ((Sep '(')  : rest) = case parseExpr rest of
  Just (e, Sep ')' : rest1) → Just (e, rest1)
  _                         → Nothing
parseFactor _                   = Nothing
```


Parsing Operators in Haskell

$$AddOp \;\rightarrow\; + \;\mid\; -$$
$$MultOp \;\rightarrow\; * \;\mid\; /$$

```
parseAddOp :: [Token] → Maybe (Op, [Token])
parseAddOp ((Sep '+') : rest) = Just (MkOp "+", rest)
parseAddOp ((Sep '-') : rest) = Just (MkOp "-", rest)
parseAddOp _                  = Nothing

parseMultOp :: [Token] → Maybe (Op, [Token])
parseMultOp ((Sep '*') : rest) = Just (MkOp "*", rest)
parseMultOp ((Sep '/') : rest) = Just (MkOp "/", rest)
parseMultOp _                  = Nothing
```


Recursive Descent Parsing in Haskell: *Expr* (1st attempt)

$$Expr \;\rightarrow\; Term\;\{\,AddOp\;Term\}^{*}$$

```
parseExpr1 :: [Token] → Maybe (Expr, [Token])
parseExpr1 input = case parseTerm input of
  Nothing → Nothing
  Just (e1, rest1) → case parseAddOp rest1 of
    Nothing → Just (e1, rest1)
    Just (op, rest2) → case parseExpr1 rest2 of
      Nothing → Nothing                  -- Syntax Error!
      Just (e2, rest3) → Just (Bin e1 op e2, rest3)
```

```
*Expr> pupd1 showExpr $ fromJust $ parseExpr1 $ simpleLexer "a-b+c"
```

Recursive Descent Parsing in Haskell: *Expr* (2nd attempt)

$$Expr \rightarrow Expr\ AddOp\ Term \mid Term$$

```
parseExpr2 :: [Token] → Maybe (Expr, [Token])
parseExpr2 input = case parseExpr2 input of
 Nothing → parseTerm input
 Just (e1, rest1) → case parseAddOp rest1 of
  Nothing → Just (e1, rest1)
  Just (op, rest2) → case parseTerm rest2 of
   Nothing → parseTerm input
   Just (e2, rest3) → Just ( Bin e1 op e2, rest3)
```

*Expr> parseExpr2 $ simpleLexer "a-b+c"

Recursive Descent Parsing in Haskell: *Term*

$$Term \rightarrow Factor\ \{MultOp\ Factor\}^*$$

```
parseTerm :: [Token] → Maybe (Expr, [Token])
parseTerm = parseFactors id

parseFactors :: (Expr → Expr) → [Token] → Maybe (Expr, [Token])
parseFactors wrap input = case parseFactor input of
 Nothing → Nothing
 Just (e1, rest1) → case parseMultOp rest1 of
  Nothing → Just (wrap e1, rest1)
  Just (op, rest2) → parseFactors (Bin e1 op) rest2
```


Recursive Descent Parsing in Haskell: *Expr*

$$Expr \rightarrow Term\ \{AddOp\ Term\}^*$$

```
parseExpr :: [Token] → Maybe (Expr, [Token])
parseExpr = parseTerms id

parseTerms :: (Expr → Expr) → [Token] → Maybe (Expr, [Token])
parseTerms wrap input = case parseTerm input of
 Nothing → Nothing
 Just (e1, rest1) → case parseAddOp rest1 of
  Nothing → Just (wrap e1, rest1)
  Just (op, rest2) → parseTerms (Bin (wrap e1) op) rest2
```

Frequently used ***Functional Programming Pattern:***

Auxiliary function with additional argument —

additional argument corresponds to imperative local variable.