Language Engineering | Lecture 11

exprs

```
expr : term ('+' expr)*
term : number
     | '(' expr ')'
```

Using parser combinators, we have a few steps to follow:

1. Create a datatype that corresponds to the non-terminals (and groups).

```
-- expr : term ('+' expr)*
data Expr = Expr Term [Exprs]
data Exprs = Add Expr
data Term = Number Int
          | Parens Expr
```

This process was achieved by creating a new datatype for each non-terminal, whose constructors correspond to the RHS of the rule.
A new datatype should also be made for each group in the same way.

2. For each datatype we create a corresponding parser, usually named after the datatype.

For instance

```
data Expr = Expr Term [Exprs]
```

will create the parser:

```
expr :: Parser Expr
expr = Expr <$> term <*> many exprs
```

This corresponds to the rule:

```
expr : term (exprs)*
exprs : '+' expr
```

[here we named the group explicitly, but it's essentially the same grammar]

So we must now define the parsers term and exprs.

For the rule:

```
term : number
     | '(' expr ')'
```

We have

```
data Term = Number Int
          | Parens Expr
```

The corresponding parser is:

```
term :: Parser Term
term = (Number <$> number)
   <|>(Parens <$ tok "(" <*> expr
              <* tok ")" )
```

For exprs we do this:

rule: `exprs : '+' expr`

```
data Exprs = Add Expr

exprs :: Parser Exprs
exprs = Add <$ tok "+" <*> expr
```