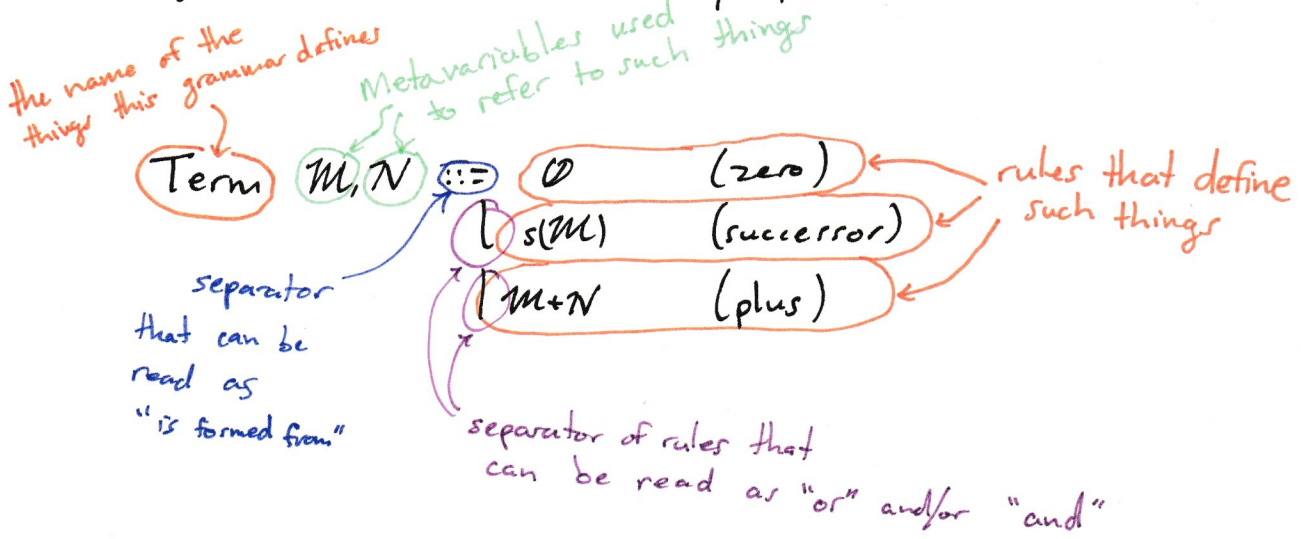
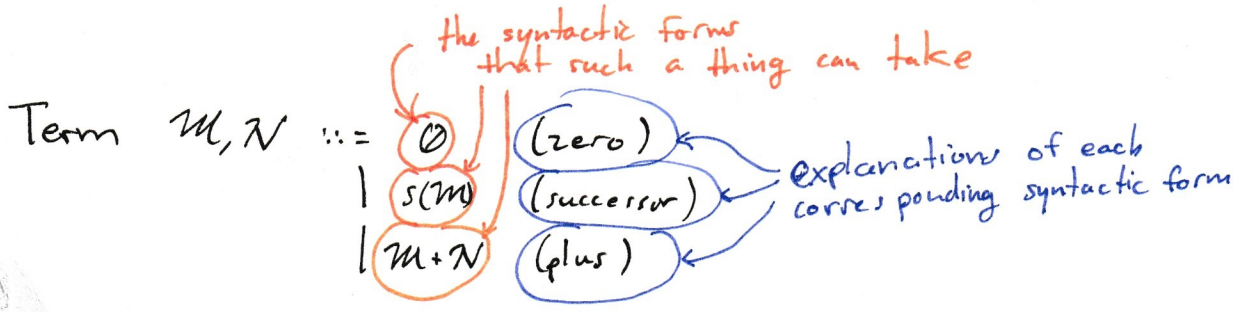
# Grammar Specifications

Specifications of grammars in a type theoretic setting usually look like this:

$$\begin{array}{lll} \text{Term } \mathcal{M}, \mathcal{N} ::= & \mathcal{O} & (\text{zero}) \\ \mid & s(\mathcal{M}) & (\text{successor}) \\ \mid & \mathcal{M}+\mathcal{N} & (\text{plus}) \end{array}$$

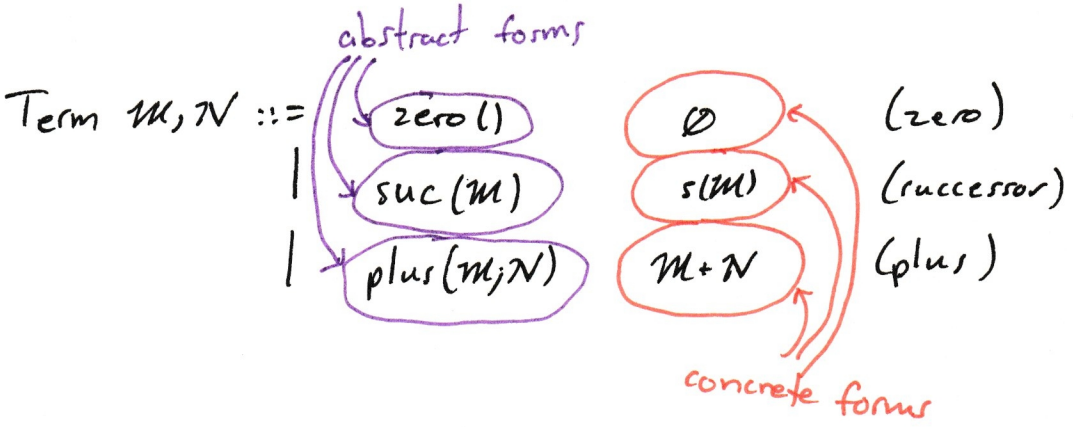
In such a grammar, there are many parts in different roles:



# Grammar Specifications (cont.)

In some cases, two sets of syntactic forms are given, written side by side. Typically, the left one is abstract syntax, while the right one is concrete:

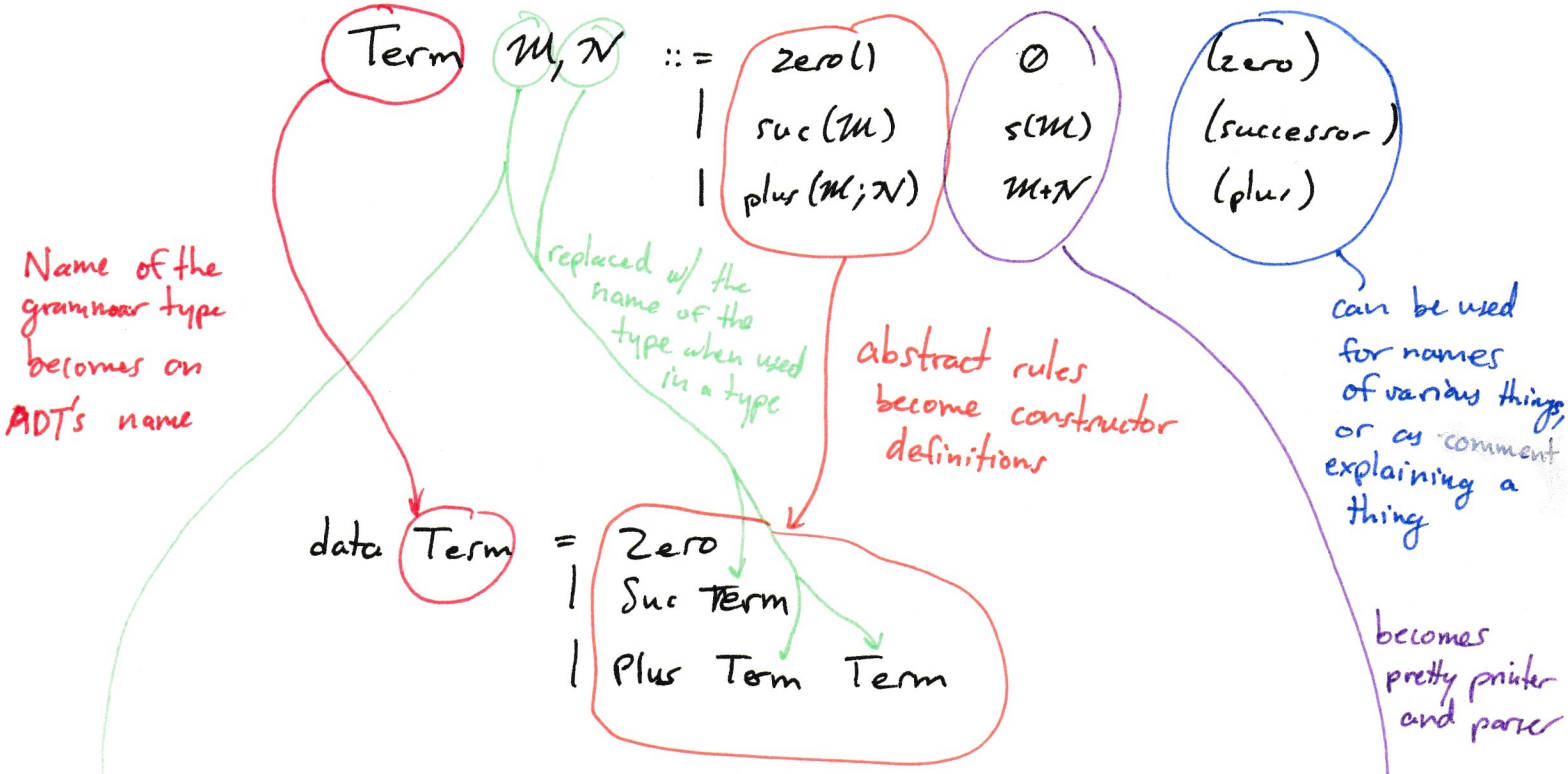



The metavariables are the same in either case, and we understand from context whether we should think of it as abstract or concrete.

We translate a grammar into a data type. The above grammar corresponds to

```
data Term = Zero
          | Suc Term
          | Plus Term Term
```

The correspondence of the two-form grammar works like so:




```
pretty :: Term → String
pretty Zero = "∅"
pretty (Suc m) = "s(" ++ pretty m ++ ")"
pretty (Plus m n) = pretty m ++ "+" ++ pretty n
```

```
parseTerm :: Parser Term
parseTerm = parseZero <|> parseSuccessor <|> parsePlus

parseZero :: Parser Term
parseZero = string "∅" >> return Zero

⋮
```