Grammar Specifications

Specifications of grammors in a type theoretic setting usually look like thir:

Term
$$\mathcal{M}, \mathcal{N} := \emptyset$$
 (zero)
 $| s(\mathcal{M}) | (successor)$
 $| \mathcal{M}+\mathcal{N} | (plus)$

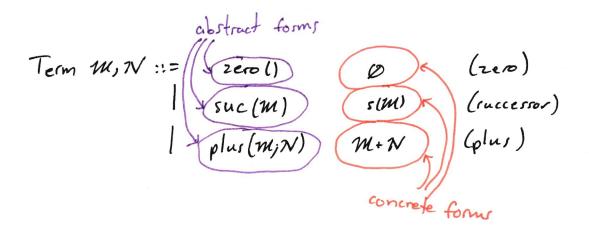
In such a grammar, there are many parts in different oler:

Metavariables used things rules that define (Term M,N) (0) (successor) (s(M) separator M+N (plus) that can be read as " is formed from" separator of rules that can be read as "or" and/or "and"

Explanations of each corresponding syntactic form (successor) (plus)

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 aght 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

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

:= (Zerol) suc (m) (successor plus (M; N) replaced of the Name of the hame of the can be used grammour type type when used abstract rules for names becomes on in a type of various things become constructor ADT's name or as comment definitions explaining a thing becomes Plus Tom Tem pretty printer and parce becomes lowercase names of variables with the type pretty : Term - String pretty Zero = "0" pretty (Suc m) = "5(" + pretty m + ")" pretty (Plus m n) = pretty m + "+" + pretty n parseterm :: Porser Term parke Term = parceZero </>
</>

> parce Successor

parce Play pase lero : Parser Term parse Zero = string "0" >> return Zero