Parser for Core Language

part 2

In Part 1 of the project we did not consider the two productions:

expr -> expr aexpr
and
expr -> expr1 binop expr2

the problem with the first production is that it is left-recursive and a naive parser following it would cause an infinite recursion
we transform the production into:
expr -> aexpr_1...aexpr_n, for n>=1 which can be easily mimicked by the parser using the function *some, c*f. with Exercise 8 of Chapter 13 of the text.

for the application of the binop's we need to have many productions (recall Section 13.8 of the text book) that model the different precedences and associatovityof the different binop's. The precedences and associativity are summarized in the following table: notice that /, -, and relational op's are not associative. This means that they can be used only once. How to deal with the left associativity of application, is explained in the previous slide.

Precedence	Associativity	Operator
6	Left	Application
5	Right	*
	None	/
4	Right	+
	None	-
3	None	== ~= > >= < <=
2	Right	&
1	Right	|

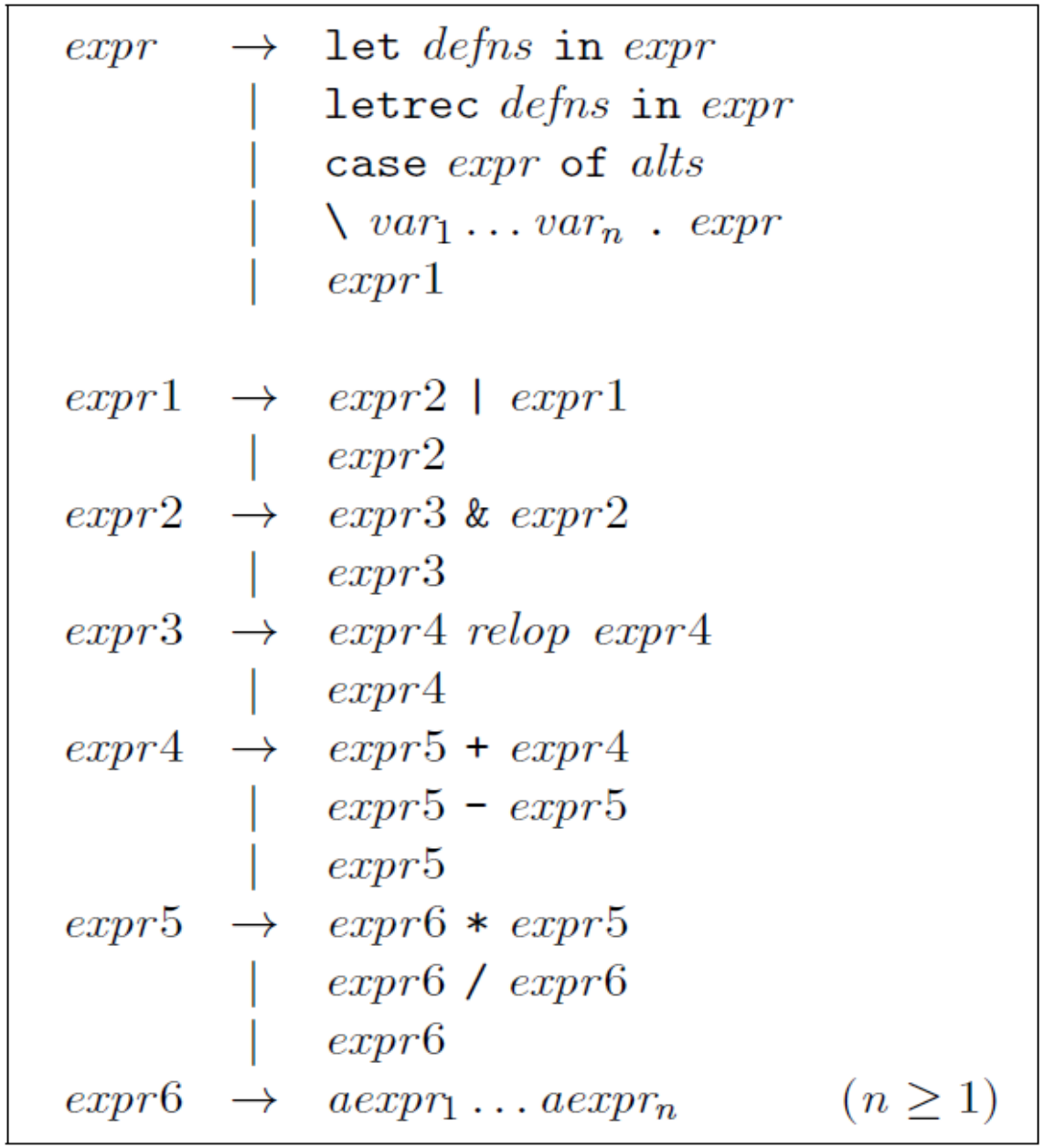

Figure 1.3: Grammar expressing operator precedence and associativity

In part 1, we have considered the production: expr :- aexpr, in the full program this is no longer needed as expr1 can generate (sequence of) aexpr.

important advice: since aexpr can be a simple variable, and, following the last production in the previous table, the parser will search the input for n aexpr, n >0, it is necessary that the function that recognizes variables, is able to distinguish variables from keywords of the Core Language, such as in, of, let, etc.
Otherwise, you risk that such keywords are mixed up with aexpr which may cause the failure of the program.