completing the parser for Core Language

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 -> aexpr1...aexprn, for n>=1 which can be easily mimicked by the parser using the function *some*

for the application of the binop’s we need to have many productions that model the different precedences of the different binop’s. The precedences and associativity are summarized in the following table:

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

in order to model the associativity and precedence of the different binop’s we follow the idea that we have discussed in chapter 13.8 of the text

the new productions for expr are in the next slide

$$
\begin{array}{lcll}
expr & \rightarrow & \texttt{let}\ defns\ \texttt{in}\ expr & \\
 & | & \texttt{letrec}\ defns\ \texttt{in}\ expr & \\
 & | & \texttt{case}\ expr\ \texttt{of}\ alts & \\
 & | & \texttt{\textbackslash}\ var_1 \ldots var_n\ \texttt{.}\ expr & \\
 & | & expr1 & \\
 & & & \\
expr1 & \rightarrow & expr2\ \texttt{|}\ expr1 & \\
 & | & expr2 & \\
expr2 & \rightarrow & expr3\ \texttt{\&}\ expr2 & \\
 & | & expr3 & \\
expr3 & \rightarrow & expr4\ relop\ expr4 & \\
 & | & expr4 & \\
expr4 & \rightarrow & expr5\ \texttt{+}\ expr4 & \\
 & | & expr5\ \texttt{-}\ expr5 & \\
 & | & expr5 & \\
expr5 & \rightarrow & expr6\ \texttt{*}\ expr5 & \\
 & | & expr6\ \texttt{/}\ expr6 & \\
 & | & expr6 & \\
expr6 & \rightarrow & aexpr_1 \ldots aexpr_n & (n \geq 1)
\end{array}
$$

Figure 1.3: Grammar expressing operator precedence and associativity

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.