Compiler Construction

Consider the following grammar for expressions (where Id is a terminal symbol representing an identifier resulting from lexical analysis):

\[
\text{Expr} ::= 1 | 2 | \text{Id} | \text{Expr} + \text{Expr} | \text{Expr} \div \text{Expr} |
\]

\[
\text{Expr} \cdot \text{Expr} | (\text{Expr})
\]

(a) Explain in what principal respect this grammar is unsatisfactory. [1 mark]

(b) Assuming further that + is to be left-associative, \(^\) is to be right-associative and \(/\) is to be non-associative (i.e. \(2/2/2\) is forbidden but \((2/2)/2\) and \(2/(2/2)\) are allowed), re-write the grammar to reflect this. [4 marks]

(c) List the terminal symbols and non-terminal symbols, and count the production rules both in the original grammar and in the grammar in your answer to part (b). Indicate the start symbol in both grammars. [2 marks]

(d) Define a type or types (in C, Java, or ML) suitable for holding an abstract syntax tree resulting from your answer to part (b). [2 marks]

(e) Give a brief and elementary explanation of the principles of how the grammar resulting from part (b) might be used to create a syntax analyser taking a token stream as input (via calls to function lex()) and giving as output an abstract syntax tree corresponding to part (d). Mention both hand-written and automatically-generated syntax analysers. [8 marks]

(f) Summarise any issues related to left- or right-associative operators in the two techniques (in implementing the parser and in constructing the tool) you outlined in part (e). [3 marks]