BnfCalculator

An Exercise in Parsing

John Till

1 Numeric Expressions

One of the quickest tasks to test our parsing abilities is to build a calculator, which requires us to parse and evaluate mathematical expressions. There are three components we will use– a scanner, parser, and abstract syntax tree (AST). The types of expressions which are parsed are summarized in the following BNF grammar:

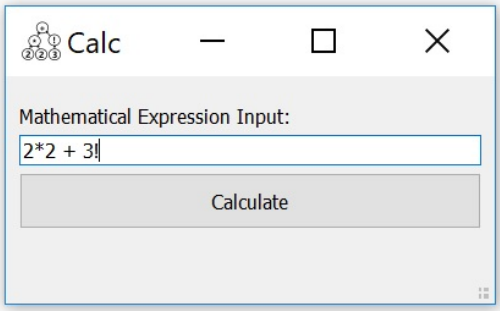
$$
\begin{aligned}
\text{Expression} &\Rightarrow \text{Below} \\
\text{Addition} &\Rightarrow \text{Below } ((\text{`+'} \mid \text{`-'})\ \text{Below})* \\
\text{Multiplication} &\Rightarrow \text{Below } ((\text{`*'} \mid \text{`/'} \mid \text{`\textbackslash'} \mid \text{`\%'})\ \text{Below})* \\
\text{LeftUnary} &\Rightarrow \text{`-'} * \text{ Below} \\
\text{RightUnary} &\Rightarrow \text{Below `!'}* \\
\text{Exponent} &\Rightarrow \text{Below } (\text{`\^{}'}\ \text{LeftUnary})? \\
\text{Primitive} &\Rightarrow \text{Grouping} \mid \text{AbsoluteValue} \mid \text{NumberLiteral} \mid \text{Call} \\
\text{Grouping} &\Rightarrow \text{`('}\ \text{Expression`)'} \\
\text{AbsoluteValue} &\Rightarrow \text{`|'}\ \text{Expression `|'} \\
\text{Call} &\Rightarrow \text{Identifier CallArgs}? \\
\text{CallArgs} &\Rightarrow \text{`('}\ (\text{Expression } (\text{`,'}\ \text{Expression})*)?\ \text{`)'}
\end{aligned}
$$

$$
\begin{aligned}
\text{NumberLiteral} &\Rightarrow (\text{`0'} - \text{`9'}) + \ (\text{`.'}\ (\text{`0'} - \text{`9'})+)? \\
\text{Identifier} &\Rightarrow (\text{`a'} - \text{`z'} \mid \text{`A'} - \text{`Z'})\ (\text{`a'} - \text{`z'} \mid \text{`A'} - \text{`Z'} \mid \text{`0'} - \text{`9'})*
\end{aligned}
$$

Fortunately the only rules with repeated symbols are numbers, started with a digit, or variables, started with a letter. Any other symbol encountered can be directly

mapped to a token type by the scanner. There are just a few keywords– “pi”, “cos”, “sin”, “tan”, “e”, “ln”, and “log”. Tokens for each keyword are actually built into the grammar, although that detail is ignored in the prior BNF grammar. The parser is a simple recursive descent parser.

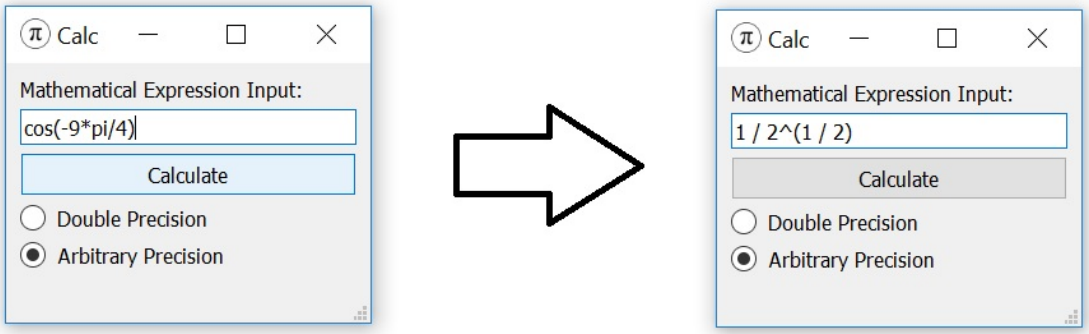
The AST uses an OOP design with a pure virtual “eval” function. While this isn’t a great design for parsing numeric expressions, it will be a useful abstraction later as the number of pure virtual functions grows.



2 Symbolic Expressions

There are a couple more drills to practice by tacking on a symbolic mode to the calculator. First, big-num arithmetic is implemented in “numbers.h”. A big unsigned integer is represented by a STL vector of uint8_t entries. Each element represents a single digit of the big integer, and the order is so that the first element corresponds to the rightmost digit, i.e. forward traversal of the vector visits the big int from right to left. It is a fairly naive implementation, but the implementation details are encapsulated well, and the solution is straightforward.

The second skill exercised is rewriting. Transformations are implemented directly in the AST as a pure virtual method to be defined by each specific node. This is not ideal to allow the AST to scale or to implement complicated transformation rules relying on many nodes (e.g. $\sin(x)^2 + \cos(x)^2 \Rightarrow 1$), but it allows us to play around with a few transformations without painstakingly defining a generalization of rewriting rules.



3 Conclusions

This is a good exercise to work on parsing, rewriting, and big-num arithmetic. The calculator app is free from any statement grammar or symbol table, and the symbolic mode is kept simple by allowing parent nodes to transform based on child types.