Comments for `evaluate`

Karl Stroetmann

May 2022

We can describe what happens in the *reading phase* using rewrite rules that describe how the three stacks `mTokens`, `mArguments` and `mOperators` are changed in each *step*. Here, a *step* is one iteration of the first `while`-loop of the function `evaluate`. The following *rewrite rules* are executed until the token stack `mTokens` is empty.

1. If the token on top of the token stack is an integer, it is removed from the token stack and pushed onto the argument stack. The operator stack remains unchanged in this case.

$$\begin{array}{ll}
\texttt{mTokens} = \texttt{mTokensRest} + [\texttt{token}] & \wedge \\
\texttt{isInteger}(\texttt{token}) & \Rightarrow \\[0.5em]
\texttt{mArguments}' = \texttt{mArguments} + [\texttt{token}] & \wedge \\
\texttt{mTokens}' = \texttt{mTokensRest} & \wedge \\
\texttt{mOperators}' = \texttt{mOperators} &
\end{array}$$

   Here, the primed variable `mArguments`′ refers to the argument stack after `token` has been pushed onto it.

   In the following rules we implicitly assume that the token on top of the token stack is not an integer but rather a parenthesis or a proper operator. In order to be more concise, we suppress this precondition from the following rewrite rules.

2. If the operator stack is empty, the next token is pushed onto the operator stack.

$$\begin{array}{ll}
\texttt{mTokens} = \texttt{mTokensRest} + [\texttt{op}] & \wedge \\
\texttt{mOperators} = [] & \Rightarrow \\[0.5em]
\texttt{mOperators}' = \texttt{mOperators} + [\texttt{op}] & \wedge \\
\texttt{mTokens}' = \texttt{mTokensRest} & \wedge \\
\texttt{mArguments}' = \texttt{mArguments} &
\end{array}$$

3. If the next token is an opening parenthesis, this parenthesis token is pushed

onto the operator stack.

$$\begin{array}{ll}
\texttt{mTokens} = \texttt{mTokensRest} + [\texttt{'('}] & \Rightarrow \\[0.5em]
\texttt{mOperators}' = \texttt{mOperators} + [\texttt{'('}] & \wedge \\
\texttt{mTokens}' = \texttt{mTokensRest} & \wedge \\
\texttt{mArguments}' = \texttt{mArguments} &
\end{array}$$

4. If the next token is a closing parenthesis and the operator on top of the operator stack is an opening parenthesis, then both parentheses are removed.

$$\begin{array}{ll}
\texttt{mTokens} = \texttt{mTokensRest} + [\texttt{')'}] & \wedge \\
\texttt{mOperators} = \texttt{mOperatorsRest} + [\texttt{'('}] & \Rightarrow \\[0.5em]
\texttt{mOperators}' = \texttt{mOperatorsRest} & \wedge \\
\texttt{mTokens}' = \texttt{mTokensRest} & \wedge \\
\texttt{mArguments}' = \texttt{mArguments} &
\end{array}$$

5. If the next token is a closing parenthesis but the operator on top of the operator stack is not an opening parenthesis, the operator on top of the operator stack is evaluated. Note that the token stack is not changed in this case.

$$\begin{array}{ll}
\texttt{mTokens} = \texttt{mTokensRest} + [\texttt{')'}] & \wedge \\
\texttt{mOperatorsRest} + [\mathrm{op}] & \wedge \\
\texttt{op} \neq \texttt{'('} & \wedge \\
\texttt{mArguments} = \texttt{mArgumentsRest} + [\texttt{lhs}, \texttt{rhs}] & \Rightarrow \\[0.5em]
\texttt{mOperators}' = \texttt{mOperatorsRest} & \wedge \\
\texttt{mTokens}' = \texttt{mTokens} & \wedge \\
\texttt{mArguments}' = \texttt{mArgumentsRest} + [\texttt{lhs op rhs}] &
\end{array}$$

Here, the expression `lhs op rhs` denotes evaluating the operator `op` with the arguments `lhs` and `rhs`.

6. If the token on top of the operator stack is an opening parenthesis, then the operator on top of the token stack is pushed onto the operator stack.

$$\begin{array}{ll}
\texttt{mTokens} = \texttt{mTokensRest} + [\mathrm{op}] & \wedge \\
\texttt{op} \neq \texttt{')'} & \wedge \\
\texttt{mOperators} = \texttt{mOperatorsRest} + [\texttt{'('}] & \Rightarrow \\[0.5em]
\texttt{mOperator}' = \texttt{mOperator} + [\mathrm{op}] & \wedge \\
\texttt{mTokens}' = \texttt{mTokensRest} & \wedge \\
\texttt{mArguments}' = \texttt{mArguments} &
\end{array}$$

In the remaining cases neither the token on top of the token stack nor the operator on top of the operator stack can be a parenthesis. The following rules will implicitly assume that this is the case.

7. If the operator on top of the operator stack needs to be evaluated before the operator on top of the token stack, the operator on top of the operator stack is evaluated.

$$\begin{array}{ll}
\texttt{mTokens} = \texttt{mTokensRest} + [o_2] & \wedge \\
\texttt{mOperatorsRest} + [o_1] & \wedge \\
\texttt{evalBefore}(o_1, o_2) & \wedge \\
\texttt{mArguments} = \texttt{mArgumentsRest} + [\texttt{lhs}, \texttt{rhs}] & \Rightarrow \\[4pt]
\texttt{mOperators}' = \texttt{mOperatorRest} & \wedge \\
\texttt{mTokens}' = \texttt{mTokens} & \wedge \\
\texttt{mArguments}' = \texttt{mArgumentsRest} + [\texttt{lhs}\ o_1\ \texttt{rhs}] &
\end{array}$$

8. Otherwise, the operator on top of the token stack is pushed onto the operator stack.

$$\begin{array}{ll}
\texttt{mTokens} = \texttt{mTokensRest} + [o_2] & \wedge \\
\texttt{mOperators} = \texttt{mOperatorsRest} + [o_1] & \wedge \\
\neg\texttt{evalBefore}(o_1, o_2) & \Rightarrow \\[4pt]
\texttt{mOperators}' = \texttt{mOperators} + [o_2] & \wedge \\
\texttt{mTokens}' = \texttt{mTokensRest} & \wedge \\
\texttt{mArguments}' = \texttt{mArguments} &
\end{array}$$