This program can calculate the results of a sequence of expressions. The expressions can include variables and functions. Generally speaking, the program consists of a scanner, a parser, a variable table, and something miscellaneous.

**SCANNER** The scanner receives the input string (expression) and split them into tokens. I used a deterministic finite automaton (DFA) to accomplish this task. Here is the automaton:

![Diagram of a DFA](image)

Since the DFS is implemented by the “switch... case...” technique, the complexity is \( \Theta(n) \) for an input of length \( n \).

**PARSER** The parser analyzes the tokens from the scanner, and decide a parse tree to calculate the expression. Originally, I designed the following CFG
(context free grammar) for parsing:

\[
S \rightarrow \text{EXPR0};
\]

\[
\text{EXPR0} \rightarrow \text{EXPR1}
\]

\[
\text{EXPR1} \rightarrow \text{EXPR2}
\]

\[
\text{EXPR2} \rightarrow \text{EXPR3}
\]

\[
\text{EXPR3} \rightarrow \text{EXPR4}
\]

\[
\text{EXPR4} \rightarrow \text{EXPR4}^* \text{EXPR3} \quad /\!/^2^* 2^* 2 \text{ means } 2^* (2^* 2)
\]

\[
\text{EXPR4} \rightarrow \text{var} \quad /\!/^\text{variable}
\]

\[
\text{EXPR4} \rightarrow \text{var}(\text{EXPR1}) \quad /\!/^\text{function}
\]

\[
\text{EXPR4} \rightarrow \text{num} \quad /\!/^\text{constant}
\]

\[
\text{EXPR4} \rightarrow (\text{EXPR1})
\]

So, we have now some left recursions and common prefixes. First we’ll remove the common prefixes in \text{EXPR4}. The rule will be changed into:

\[
\text{EXPR4} \rightarrow (\text{EXPR1})
\]

\[
\text{EXPR4} \rightarrow \text{var}\text{EXPR4x}
\]

\[
\text{EXPR4} \rightarrow \text{num}
\]

\[
\text{EXPR4x} \rightarrow (\text{EXPR1})
\]

\[
\text{EXPR4x} \rightarrow \epsilon
\]

^ Then we’ll remove the common prefixes in \text{EXPR3}. The rule will be changed into:

\[
\text{EXPR3} \rightarrow \text{EXPR4EXPR3x}
\]

\[
\text{EXPR3x} \rightarrow ^* \text{EXPR3}
\]

Then we’ll remove the left recursion of \text{EXPR2}. The rule will be changed into:

\[
\text{EXPR2} \rightarrow \text{EXPR3EXPR2x}
\]

\[
\text{EXPR2x} \rightarrow (^/\!/)\text{EXPR3EXPR2x}
\]

\[
\text{EXPR2x} \rightarrow \epsilon
\]

It’s the similar case for \text{EXPR1}:

\[
\text{EXPR1} \rightarrow \text{EXPR2EXPR1x}
\]

\[
\text{EXPR1x} \rightarrow (+/\!-)\text{EXPR2EXPR1x}
\]

\[
\text{EXPR1x} \rightarrow \epsilon
\]

Last, we need to remove common prefixes in \text{EXPR0} since \text{EXPR0} \rightarrow ^* \text{EXPR4} \rightarrow \text{var}, and \text{EXPR0} \rightarrow \text{var} = \text{EXPR0}, thus we have common prefixes to remove. First, we’ll convert \text{EXPR0} \rightarrow \text{EXPR1} into a terminal-starting rule

\[
\text{EXPR0} \rightarrow (\text{EXPR1})\text{EXPR3xEXPR2xEXPR1x}
\]

\[
\text{EXPR0} \rightarrow \text{var}\text{EXPR4xEXPR3xEXPR2xEXPR1x}
\]

\[
\text{EXPR0} \rightarrow \text{num}\text{EXPR3xEXPR2xEXPR1x}
\]

2
Thus, we have common prefixes between the second rule and the assignment rule. So we’ll modify it as:

\[
\begin{align*}
EXPR0 & \rightarrow (EXPR1)EXPR3xEXPR2xEXPR1x \\
& \rightarrow numEXPR3xEXPR2xEXPR1x \quad // thesearethesame \\
& \rightarrow varEXPR0x \\
EXPR0x & \rightarrow = EXPR0 \\
& \rightarrow EXPR4xEXPR3xEXPR2xEXPR1x \\
\end{align*}
\]

So the grammar is done. The overall grammar would be like:

\[
\begin{align*}
S & \rightarrow EXPR0; \\
EXPR0 & \rightarrow (EXPR1)EXPR3xEXPR2xEXPR1x \\
& \rightarrow numEXPR3xEXPR2xEXPR1x \\
& \rightarrow varEXPR0x \\
EXPR0x & \rightarrow = EXPR0 \\
& \rightarrow EXPR4xEXPR3xEXPR2xEXPR1x \\
EXPR1 & \rightarrow EXPR2EXPR1x \\
EXPR1x & \rightarrow (+\mid-)EXPR2EXPR1x \\
& \rightarrow \epsilon \\
EXPR2 & \rightarrow EXPR3EXPR2x \\
EXPR2x & \rightarrow (\star\mid/)EXPR3EXPR2x \\
& \rightarrow \epsilon \\
EXPR3 & \rightarrow EXPR4EXPR3x \\
EXPR3x & \rightarrow \hat{\epsilon} EXPR3 \\
& \rightarrow \epsilon \\
EXPR4 & \rightarrow (EXPR1) \\
& \rightarrow varEXPR4x \\
& \rightarrow num \\
EXPR4x & \rightarrow (EXPR1) \\
& \rightarrow \epsilon \\
\end{align*}
\]

The \texttt{FIRST} and \texttt{FOLLOW} set would be:

\[
\begin{array}{ll}
\text{FIRST} & \text{FOLLOW} \\
S & \{\langle var, num \rangle\} \quad \{\epsilon\} \\
EXPR0 & \{\langle var, num \rangle\} \quad \{\epsilon\} \\
EXPR0x & \{\epsilon\} \quad \{\epsilon\} \\
EXPR1 & \{\langle var, num \rangle\} \quad \{\epsilon\} \\
EXPR1x & \{\langle +, - \rangle\} \quad \{\epsilon\} \\
EXPR2 & \{\langle *, / \rangle\} \quad \{\langle +, - \rangle\} \\
EXPR2x & \{\epsilon\} \quad \{\langle +, - \rangle\} \\
EXPR3 & \{\langle var, num \rangle\} \quad \{\langle +, - \rangle, \star, \hat{\epsilon}, \hat{\star} \} \\
EXPR3x & \{\langle \star, \epsilon \rangle\} \quad \{\langle var, num \rangle\} \\
EXPR4 & \{\langle var, num \rangle\} \quad \{\langle +, - \rangle, \star, \hat{\epsilon}, \hat{\star} \} \\
EXPR4x & \{\langle \epsilon \rangle\} \quad \{\langle +, - \rangle, \star, \hat{\epsilon}, \hat{\star} \}
\end{array}
\]

Thus, since we got a \texttt{LL}(1) grammar, we can implement it directly. Actually, in my program, there are functions called \texttt{parse\_xxx()}, where xxx stands for one
of those non-terminals. Since it’s a LL(1), the implementation will be in \( \Theta(n) \).

After we build the parse tree, we can perform calculation within the tree. Actually, it’s very easy to calculate using the original CFG, storing each intermediate value for each non-terminal. However, when we convert it into a LL(1) grammar, it’s not so easy to store the intermediate results.

So I passed a parameter for the non-terminals corresponding to the left-associative operations, i.e. EXPR1x, EXPR2x, as the value of the previous terminal. Then, I can carry out calculations with some rules.

**VARIABLE TABLE**  The variable table is used to store the names of variables and their values. I just use an dynamic allocated string array and double array to implement this.

**MISCELLANEOUS**  Since I can’t modify the reader.*, I have to read the whole input and make the results after the input. However, I want a program that can run on multiple cases without calling it from the prompt again and again. So I implement a shell which calls the core program again and again. So you can just call the main program once and carry out the calculation.