# Handwritten Syntactic Analyzer

In this lab, we are going to apply what we knew from the lecture… but we'll have some stuff to read in order to be able to implement it in the required time (which is the lab's duration).

As we have already known, syntactic analyzers (parsers) read the input one at a time (receiving the tokens from the lexical analyzer one-by-one) and decide whether the input string is in the language or not. This is what we are going to do *manually*. This means we're not going to use a tool to generate the code for the syntactic analyzer (parser); we are going to write the code ourselves. But you can use JLex to implement the lexical analyzer if you want.

The lab problem will be principally depending on a lexical analyzer and a syntactic analyzer (parser) for a simple structure.

For understanding the theoretical part, please read the following chapters from the book I sent you: 2.1, 2.2 (basically for revision), but mainly 2.3, 2.4 and 2.5 must be read. Give special interest to understanding the code on page 51 and 55, the lab problem will be quite close to it.

Please try to start preparing the whole thing as early as possible – this lab *will not be easy*. Your questions are always welcome.

Awaiting you… ;-)

*Time: 180 minutes*

# MMTree

*Source File: MMTree.java*

We are to write a lexical analyzer and a parser for the general form of mathematical expressions. The following tokens are available:

**Number**

A non-empty series of digits. Floating point numbers are not supported.

**Whitespace**

All whitespace characters (`' '` `'\b'` `'\f'` `'\n'` `'\r'` `'\t'`) are to be ignored.

**Operators and Separators**

```
+          -          *          /          %          (          )          ;
```

## Input *(file name: MMTree.in)*

A series of valid mathematical expressions, each ending in a semicolon.

## Output *(file name: MMTree.out)*

Each line contains the solution of the corresponding mathematical expression, rounded to two decimal places in the shown format… which shows the precedence of the evaluation process. Brackets in the input are output as-is. All numbers are to be output as `doubles`. Positive numbers must be preceded in the output by a plus sign. Negative numbers must be preceded in the output by a negative sign. The abstract syntax tree for each expression must be built firstly and then traversed to produce the corresponding output.

## Sample Input

```
3; 5-6;1-2-3*4 +9%3 -(3-(4-2));
(5*3*10/25)%005;5*3*010/25%5;
6-3%2;(12-(9*5%6))/3;(161-(14-(0008-007) * (6-

2)));
7*7/7*8/8*9/9; 5+3;     5-((3) + (000*010*016*019     *012*017*018* 011));
```

## Sample Output

```
1) +3.0 = 3.0;
2) (+5.0-+6.0) = -1.0;
3) ((((+1.0-+2.0)-(+3.0*+4.0))+(+9.0%+3.0))-+((+3.0-+((+4.0-+2.0))))) = -14.0;
4) (+((((+5.0*+3.0)*+10.0)/+25.0))%+5.0) = 1.0;
5) ((((+5.0*+3.0)*+10.0)/+25.0)%+5.0) = 1.0;
6) (+6.0-(+3.0%+2.0)) = 5.0;
7) (+((+12.0-+(((+9.0*+5.0)%+6.0))))/+3.0) = 3.0;
8) +((+161.0-+((+14.0-(+((+8.0-+7.0))*+((+6.0-+2.0))))))) = 151.0;
9) ((((((+7.0*+7.0)/+7.0)*+8.0)/+8.0)*+9.0)/+9.0) = 7.0;
10) (+5.0++3.0) = 8.0;
```

```
11) (+5.0-+((+(+3.0)++(((((((+0.0*+10.0)*+16.0)*+19.0)*+12.0)*+17.0)*+18.0)*+11.0))))) =
2.0;
```