COMP 141: Course Project
Phase 3.1: Evaluator for L_{exp}

Over the course of the six phases, we will construct an interpreter for a small imperative language, that we call L_{imp}. In this phase we are interested in a subset of the language, called L_{exp}, i.e., the language of expressions that you have already built a scanner for it in phase 1.1, and a parser for it in phase 2.1.

For the current phase you will construct the evaluator module for L_{exp}. The evaluator receives the AST of a program and returns the result of evaluation.

Scanner Specification There are three types of tokens in L_{exp}, already specified in phase 1.1, defined by the following regular expressions.

```
IDENIFIER = ([a-z]  | [A-Z])([a-z] | [A-Z] | [0-9])*
   NUMBER = [0-9]+
   SYMBOL = \+  |  \-  |  \*  |  /  |  \(  |  \)
```

The following rules define how separation between tokens should be handled.

- White space[1] is not allowed in any token, so white space always terminates a token and separates it from the next token. Except for indicating token boundaries, white space is ignored.
- The principle of longest substring should always apply.
- Any character that does not fit the pattern for the token type currently being scanned immediately terminates the current token and begins the next token. The exception is white space, in which case the first rule applies.

Parser Specification Grammar of L_{exp} is defined as follows:

$$\begin{aligned}
\textit{expression} &::= \textit{term} \ \{ \ + \ \textit{term} \ \} \\
\textit{term} &::= \textit{factor} \ \{ \ - \ \textit{factor} \ \} \\
\textit{factor} &::= \textit{piece} \ \{ \ / \ \textit{piece} \ \} \\
\textit{piece} &::= \textit{element} \ \{ \ * \ \textit{element} \ \} \\
\textit{element} &::= \texttt{(} \ \textit{expression} \ \texttt{)} \ | \ \texttt{NUMBER} \ | \ \texttt{IDENTIFIER}
\end{aligned}$$

Note that *expression* is the starting nonterminal.

Abstract syntax trees Consider the following details regarding the generated abstract syntax tree.

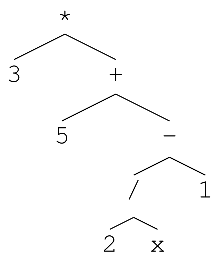
- All binary numerical operations (addition, multiplication, subtraction, and division) must denote a tree with two subtrees, and the operator at the root.
- All parentheses must be dropped in the abstract syntax trees.

[1]tabs, spaces, new lines

Example Consider the following expression in L_{exp}:

```
3 * (5 + 2 / x - 1)
```

The generated AST according to the defined grammar is as follows:



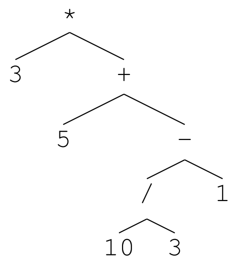
Evaluator Specification Evaluator for L_{exp} consists of one major data structure: a **stack** that represents the partially evaluated expression, e.

The AST resulted from the parsing phase must be passed to the evaluator, initially. Evaluator studies this tree in pre-order fashion and pushes the nodes of that AST to the stack. In each push of a node into the stack, check whether **the top three elements** of the stack can be evaluated. This happens only if the two top elements are numbers, and the third node from top is an operator. If this is the case, pop those top three elements and push back the result (which is the application of operator on those two numbers). This way, you can evaluate the expressions. Finally, the stack consisting of single node (which conveys the final value) is reported.

Consider the following for the implementation of the evaluator:

- Values evaluate as expected, e.g., `56` evaluates to number 56, and `26` evaluates to number 26.
- Operators are applied to the values, returned by their subtrees.
- Operators evaluate as expected[2], i.e.,
  - + denotes numerical addition.
  - $\star$ denotes numerical multiplication.
  - $-$ denotes numerical subtraction. There are no negative numbers in this language. Refer to lexical structure for clarification.
  - / denotes division on nonnegative integers, e.g., `3/2` evaluates to `1`. In the case of division by zero, the evaluator must stop and raise an exception.

Example evaluation Consider the following expression: `3 * (5 + 10 / 3 - 1)`. The AST returned by parser would be as follows, which is the input top the evaluator.



The evaluator does a pre-order traversal and evaluates the expression. The preo-order traversal of the AST pushed into the stack is as follows:

[2]There is one exceptions, however. Subtracting larger number from a smaller number results in 0, rather than a negative number, e.g., 3 - 6 should be evaluated to 0. This is due to the fact that the language is not supporting negative numbers.

(1) Push `*` to the empty stack

(2) Push `3` resulting in `* 3`

(3) Push + resulting in `* 3 +`

(4) Push `5` resulting in `* 3 + 5`

(5) Push − resulting in `* 3 + 5 -`

(6) Push / resulting in `* 3 + 5 - /`

(7) Push `10` resulting in `* 3 + 5 - / 10`

(8) Push `3` resulting in `* 3 + 5 - / 10 3`.

(9) Since **the top three elements** of the stack can be evaluated, pop them and push back the result, i.e., pop `/ 10 3`, and push back `3` since `10/3` is `3`. The result is `* 3 + 5 - 3`

(10) Push `1` resulting in `* 3 + 5 - 3 1`.

(11) Since **the top three elements** of the stack can be evaluated, pop them and push back the result, i.e., pop `- 3 1` and push back `2`, since `3 - 1` is `2`. The result is `* 3 + 5 2`.

(12) Since **the top three elements** of the stack can be evaluated, pop them and push back the result, i.e., pop `+ 5 2` and push back `7`, since `5 + 2` is `7`. The result is `* 3 7`.

(13) Since **the top three elements** of the stack can be evaluated, pop them and push back the result, i.e., pop `* 3 7` and push back `21`, since `3 * 7` is `21`. The result is `21`, hence 21 is the final value returned by the evaluator.

Language Selection

- You can build your interpreter in any language that you like.
- For more obscure languages, you will be responsible for providing instructions detailing how the program should be compiled or interpreted. Check with the instructor for the level of detail necessary for a particular language.

Input and Output Requirements Your full program (scanner module, parser module and evaluator) must read from an input file and write an output file. These files should be passed to the program as an argument in the command line. The input file would consist of a single expression without reference to any identifiers in this phase, despite the fact that identifiers are part of L_{exp}[3].

For example, if your parser is a python program in parser.py, we will test it using a command like:

```
python interpreter.py test_input.txt test_output.txt
```

The interpreter's output should be sent to the output file. The output should include:

1. The list of tokens produced by the scanner.
2. The abstract syntax tree produced by the parser.
3. The final value returned by the evaluator.

[3]We will evaluate expressions with identifiers in the next phase of the project.

Example Program I/O Assume that the input expression is as follows:

```
3 * (5 + 10 / 3 - 1)
```

Then, the output may be as follows:

```
Tokens:

3 : NUMBER
* : SYMBOL
( : SYMBOL
+ : SYMBOL
10 : NUMBER
/ : SYMBOL
3 : NUMBER
- : SYMBOL
1 : NUMBER
) : SYMBOL

AST:

* : SYMBOL
        3 : NUMBER
        + : SYMBOL
                5 : NUMBER
                - : SYMBOL
                        / : SYMBOL
                                10 : NUMBER
                                3 : NUMBER
                        1 : NUMBER

Output: 21
```

Error Handling Requirements If your scanner encounters an error while processing an input line, it should abort in a graceful manner. An error message and the input line that caused the error should be printed in the output file. After the error is reported, the program should shut down.

If the parser encounters an error, it should abort in a graceful manner. An error message and the token that caused the error should be printed in the output file. After the error is reported, the program should shut down.

If the evaluator encounters an error, e.g., when it wants to evaluate `2 / 0`, it should report a simple error message and shut down.

Submission Format Requirements Your submission should include the source code for your scanner module, parser module and test driver, along with a text file containing instructions for building and running your program. Every source code file should have your name and the phase number in a comment at the top of the file. The instruction file must at least identify the compiler that you used for testing your program.

Submit an archive file (.zip) containing the code and instruction file.