COMP 141: Course Project
Phase 1.1: Scanner 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.

For the current phase you will construct the scanner module for L_{exp}. A scanner, also called a tokenizer or a lexer, processes the stream of characters that holds the expression to be evaluated. The result is a stream of tokens, where a token is a meaningful "word" in the language.

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

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

That is,

- identifiers consisting of strings where the first symbol is upper/lower case alphabetic character and the remaining are zero or more alphanumeric characters.
- numbers consisting of strings of one or more digits.
- symbols consisting of addition, subtraction, multiplication, and parentheses symbols.

Although the regular expressions defining `IDENIFIER` and `NUMBER` tokens indicate that these tokens can be infinitely long, you can assume that no token will be longer than 100 characters. This restriction is primarily to allow you to use fixed length arrays or strings to hold tokens if variable length arrays or strings would be unnecessarily complex.

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.

Examples

- `catfood` is the single identifier `catfood`.
- `catfood45` is the single identifier `catfood45`.

[1]tabs, spaces, new lines

- `catfood-45` is three tokens: the identifier `catfood`, the symbol `-`, and the number `45`.
- `catfood67.` is a syntax error. The identifier `catfood67` is followed by a single period, which is not a valid token.
- `45)` is two tokens, the number `45` and the symbol `)`.

Language Selection

- You can build your interpreter in any language that you like; Java or Python recommended due to richness of existing libraries for string manipulation.
- 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.
- You will need to add the parser and evaluator modules to your program later in the semester. This will require the use of trees and associative arrays (maps or dictionaries). You may want to consider how well your chosen language will handle these data structures.

Input and Output Requirements Your full program (scanner module and test driver) 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. For example, if your scanner is a python program in scanner.py, we will test it using a command like:

```
python scanner.py test_input.txt test_output.txt
```

Your test driver program should be a loop, as follows:

```
Loop:
  Read a line from input file.
  Print the input line to the output file.
  Pass the input line to the scanner and receive the list of tokens.
  Print the tokens to the output file, in order, one token per line. Each line should report the token type and the token value.
```

Error Handling Requirements If your scanner encounters an error while processing an input line, it should abort in a graceful manner, leaving the module ready to process a new line of input.The test driver should be constructed such that it recognizes a scanner error, reports the error, then continues to process the next line of input.

Example Assume the input file consists of the following:

```
34 + 89 - x * y23
cxo6y / 67z23 * 56& 34 * x
2 + x
```

Then, the output of the program may be as follows:

```
Line: 34 + 89 - x * y23
34 : NUMBER
+ : SYMBOL
89: NUMBER
- : SYMBOL
x : IDENTIFIER
* : SYMBOL
y23 : IDENTIFIER

Line: cxo6y / 67z23 * 56& 34 * x
```

```
cxo6y : IDENTIFIER
/ : SYMBOL
67 : NUMBER
z23 : IDENTIFIER
* : SYMBOL
56 : NUMBER
ERROR READING "&"

Line: 2 + x
2 : NUMBER
+ : SYMBOL
x : IDENTIFIER
```

Submission Format Requirements Your submission should include the source code for your scanner 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.

Notes on Program Structure Since the scanner will ultimately be one module in a larger system, it will be best to define the scanner itself as a single function or method that receives a string and returns a list of tokens (or some similar ordered data structure). The main program should be a test driver program that receives strings from the input file and prints the resulting tokens to the output output.