# ICSI 311 Assignment 1 – The Lexer Part 1

**This assignment is extremely important – (nearly) every assignment after this one uses this one!**

**If you have bugs or missing features in this, you will need to fix them before you can continue on to new assignments. This is very typical in software development outside of school.**

**You must submit .java files. Any other file type will be ignored. Especially “.class” files.**

**You must not zip or otherwise compress your assignment. Blackboard will allow you to submit multiple files.**

***You must submit buildable .java files for credit.***

## Introduction

In this project, we will begin our lexer. Our lexer will start by reading the strings of the .shank file that the user wants to run. It will break the Shank code up into “words” or tokens and build a collection of these tokens. We can consider the lexer complete when it can take any Shank file and output a list of the tokens generated.

We will not be using the Scanner class that you may be familiar with for reading from a file; we are, instead, using Files.readAllLines. This is a much simpler way of dealing with files.

Example of readAllLines:

Path myPath = Paths.get(“someFile.shank”);

List <String> lines = Files.readAllLines(myPath, StandardCharsets.UTF\_8);

A second concept you may not be familiar with is “enum”. Enum, short for enumeration. This is a Java language construct that lets us create a variable that may be any one of a list of things. We will use this to define the types of tokens – a tokenType.

enum colorType { RED, GREEN, BLUE }

colorType myFavorite = colorType.BLUE;

System.out.println(myFavorite); // prints BLUE

## Details

To start with, we will build a lexer that accepts words and numbers and generates a collection of tokens. There are three types of tokens in this assignment – WORD, NUMBER and ENDOFLINE.

A word is defined as a letter (upper or lower case) and then any number of letters or numbers. In regular expression terms: [A-Za-z][A-Za-z0-9]\* - anything not a letter or number ends the word.

A number is defined as integer or floating point; in regular expressions: [0-9]\*[.]?[0-9]+ - anything else ends the number.

Any character that is not word or number is not a token (for now) but does show the end of a token.

For example:

word1 123.456 2

would lex to 4 tokens:

WORD (word1)

NUMBER (123.456)

NUMBER (2)

ENDOFLINE

The lexer class will hold a collection of tokens; it will start out empty and calls to lex() will add to it. You must use a state machine to keep track of what type of token you are in the middle of. Any character that is not a letter or number will reset the state machine and output the current token. The end of the line will also cause the current token to be output. Output an ENDOFLINE token at the end of each line.

This is the state machine that you should build into your lexer:

A screenshot of a computer

Description automatically generated with low confidence

## Requirements

This assignment **must** have three different source code files.

One file **must** be called Shank.java.

Shank.java **must** contain main. Your main **must** ensure that there is one and only one argument (args). If there are none or more than 1, it **must** print an appropriate error message and exit. That one argument will be considered as a filename. Your main **must** then use Files.ReadAllLines to read all of the lines from the file denoted by filename. Your main **must** instantiate one instance of your Lexer class (to be defined below). You **must** parse all lines using the lex method of the Lexer class (calling it repeatedly). If lex throws an exception, you **must** catch the exception, print that there was an exception. You **must** then print each token out (this is a temporary step to show that it works) once the lexing is complete.

One file must be called Token.java. This file **must** contain a Token class. The token class is made up of an instance of an enum (tokenType) and a value string. There **must** be a public accessor for both the enum and the value string; the underlying variables **must** be private. You may create whatever constructors you choose. The enum **must** be defined as containing values appropriate to what we will be processing. The definition of the enum should be public, but the instance inside Token must be private. We will add to this enum in the next several assignments. You will find it helpful to create an appropriate “ToString” overload. The enum should be inside the Token class so you will reference it as:

token.tokenType.WORD

The final file **must** be called Lexer.java. The Lexer class **must** contain a lex method that accepts a single string and returns nothing. The lex method **must** use one or more state machine(s) to iterate over the input string and create appropriate Tokens. Any character not allowed by your state machine(s) should throw an exception. The lexer needs to accumulate characters for some types (consider 123 – we need to accumulate 1, then 2, then 3, then the state machine can tell that the number is complete because the next character is not a number). You may **not** use regular expressions to do your lexical analysis – you must build your own state machine(s).

Some examples of valid input and the result output are:

|  |  |
| --- | --- |
| ***Input*** | ***Output*** |
| *an empty line* | ENDOFLINE |
| *5 goodbye* | NUMBER (5) WORD(goodbye) ENDOFLINE |
| *5.23 – 8.5 + 3* | NUMBER(5.23) NUMBER(8.5) NUMBER(3) ENDOFLINE |
| *8 he11o 4 world 99999* | NUMBER (8) WORD(he11o) NUMBER (4) WORD(world) NUMBER (99999) ENDOFLINE |

**HINTS**

Do not wait until the assignment is nearly due to begin. **Start early** so that you can ask questions.

You may find it useful to make some private helper methods in your lexer. **This is encouraged.**

Draw out your state machine(s) before you start coding.

Create some examples and walk through them with your state machine(s)

Test your work thoroughly before handing it in. Trade test cases with your friends to torture each other’s code.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Rubric | Poor | OK | Good | Great |
| Comments | None/Excessive (0) | “What” not “Why”, few (5) | Some “what” comments or missing some (7) | Anything not obvious has reasoning (10) |
| Variable/Function naming | Single letters everywhere (0) | Lots of abbreviations (5) | Full words most of the time (8) | Full words, descriptive (10) |
| Shank.java/main | Doesn’t exist or named wrong (0) |  |  | Exists and named correctly (5) |
| File reading | Non-Existent (0) | Uses some other mechanism for reading lines from a file(2) |  | Uses RealAllLines(5) |
| Calling lex | Non-Existent (0) | Doesn’t call lex appropriately(3) |  | Instantiates Lexer and calls lex for all lines (5) |
| Handling exceptions | Don’t handle exception (0) |  |  | Handles exception and prints message(5) |
| Printing results | Doesn’t print results |  |  | Prints results appropriately(5) |
| Token.java | Doesn’t exist(0) | One of: private enum, private value, has accessors (5) | Two of: Exists, private enum, private value, has accessors (7) | Exists, private enum, private value, has accessors (10) |
| Lexer.java | Doesn’t exist or no public lex method(0) |  |  | Exists, has public lex method(5) |
| State machine(s) – words handled | Nonexistent or never correct (0) |  | Most cases handled(10) | All cases handled(15) |
| State machine(s) – numbers handled | Nonexistent or never correct (0) | Some cases handled(7) | Most cases handled(13) | All cases handled(20) |
| End of Line output | None (0) |  |  | Outputs (5) |