CS335A Assignment 1

Deepak Sangle, 200860
sangleds20@iitk.ac.in

January 24, 2023

Toy Programming Language Lexer

Overview

The Lexical Analyzer takes as input a file with .335 extension, analyzes its content, implement the regular expressions in the form of Finite Automata following basic principles of Lexical Analysis, extract correct Tokens and lexemes out of it, stores them and output the result in a comma separated(csv) file.

Codebase Structure

1. The definition and syntactic rules of the toy programming language is given in the ”assignment1.pdf” in root directory. These rules are strictly followed.
2. The regular expressions are implemented using flex in ”la.l” file of ”Toy-Programming-Lexer” directory.
3. There is also a ”helper.h” file in root directory to implement macros and other function definition.
4. Upon compilation of the .l file, we get a ”lex.yy.c” file which implements the regular expressions of .l file using some finite automata definition.
5. The output of the Lexer is present in ”output.csv” file.
6. There are some pre-defined test cases present in the concerned directory already which can be use to evaluate some test cases.
7. The ”README.md” file contains detailed instructions on how to run the lexer using various commands.

Some Regular Expressions

Some of the basic regular expressions are given below. All of them are implemented in "la.l" file's definition section which you can refer.

- NUMBER: $\{0-9\}^+|\{0-9\}^+.\{0-9\}^*$
- IDENTIFIER: $\{a-zA-Z\}\{a-zA-Z0-9\}^*$
- EXPONENT NUMBER: $\{\text{NUMBER}\}\{E|e\}\{+|-\}?\{0-9\}^+$

Assumptions

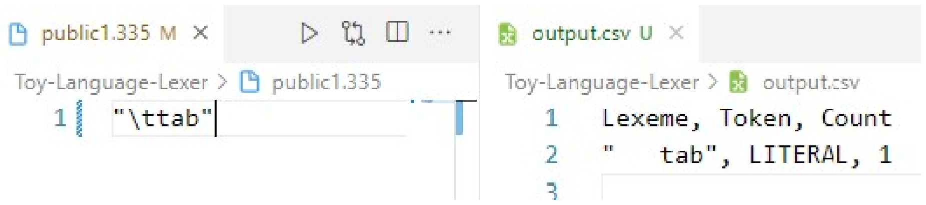
The major part of the lexer goes into assuming a lot of technical details. I have tried to summarize them all in the following points.

1. I have assumed that for signed numbers (say -2), the lexical analyzer will give '-' as a binary operator and '2' as a numeric literal. I have assumed this because I am saying that -2 is essentially same as 0-2, and thus I will treat such signed numbers during lexical parsing of this language.
2. As you have specified that "2." is a valid literal and ".2" is not, I have assumed the same and coded the regular expressions accordingly.
3. Since the assignment doesn't mentions all the excape sequences, I have assumed the following escape sequences for the toy language. \n, \r, \t, \f, \v.
4. I have assumed that syntax errors like 1234abc will give two seperate tokens as numberic literal(1234) and identifier(abc), and such errors will be taken into consideration during lexical parsing of the language.
5. Apart from the given set of characters, all the other input characters are treated as illegal characters in this language.
6. Multiline strings are not allowed in this language and throws an error.
7. All the other assumptions which are mentioned in the assignment pdf are also reflected in the lexer file.

Important Point to Know

Below are some of the points which should be taken into consideration before evaluating different test cases.

1. The escape sequence present in the lexeme string is replaced with the actual escape character. For eg., if the input string is of the form "\ttab", then the lexeme actually stores '\t' in the first position, instead of storing '\' in first position and 't' in second position of the lexeme string. Thus, in the output CSV file, the lexer actually gives the exact string expressions by evaluating these escape sequences.


public1.335 M
Toy-Language-Lexer > public1.335
output.csv U
Toy-Language-Lexer > output.csv

```
"\ttab"
```

```
Lexeme, Token, Count
"   tab", LITERAL, 1
```

The same functionality is applicable to all the escape sequences mentioned.

2. The lexer gives an error if it finds the first unbalanced comment or string delimiter, i.e. { or ”.
3. The Lexer terminates as soon as it encounters any of the lexical error and outputs the error in the terminal and thus the CSV file remains unedited.
4. For input like 2..56, the lexer correctly identifies 2 and 56 as separate numeric literals and .. as a delimiter, even though longest matching rule says to match 2. as a numeric literal. This feature is applicable for floating point numbers as well. For eg. 10.5..50.5 gives 10.5 and 50.5 as two numeric literals and .. as a delimitor.

Java Language Lexer

Overview

The Lexical Analyzer takes as input a java file, analyzes its content, implement the regular expressions in the form of Finite Automata following basic principles of Lexical Analysis, extract correct Tokens and lexemes out of it, stores them and output the result in a comma separated(csv) file.

Codebase Structure

The codebase structure is exactly similar as that of the Toy programming Language. It contains ”la.l” file, ”lex.yy.c” file, ”output.csv” and some test cases files. Important point to not is that I have the same ”helper.h” file for both the lexers.

Regular Expressions

There are many regular expressions which are implemented in the ”la.l” file. All of this regex follows the exact rules as that of the Java Manual. All this regex can be found in the definition section of the lexer file.

Assumptions

The major part of the lexer goes into assuming a lot of technical details. I have tried to summarize them all in the following points.

1. The Java Manual doesn’t mentioned anything about signed numbers. It follows the same principles that -2 can be treated as 0-2, and thus treating ”-” as binary operator only.

2. I have assumed that syntax errors like 1234abc will give two seperate tokens as numberic literal(1234) and identifier(abc), and such errors will be taken into consideration during lexical parsing of the language.

3. Apart from the given set of characters, all the other input characters are treated as illegal characters in this language.

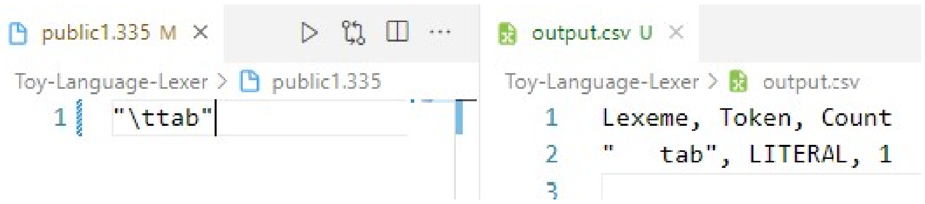
4. Numbers like 2.5.5 will return two numeric literal tokens, viz 2.5 and .5. Error handling of such numbers will be done during lexical parsing of the language.

5. Characters like ‘(backtick), #(hash), $(dollar) are some characters that are assumed to be illegal when present outside any of the string, character or text block literal since they are not included in any of the regular expressions.

Important Point to Know

Below are some of the points which should be taken into consideration before evaluating different test cases.

1. The escape sequence present in the lexeme string is replaced with the actual escape character. For eg., if the input string is of the form ”\ttab”, then the lexeme actually stores ’\t’ in the first position, instead of storing ’\’ in first position and ’t’ in second position of the lexeme string. Thus, in the output CSV file, the lexer actually gives the exact string expressions by evaluating these escape sequences.


public1.335 M
Toy-Language-Lexer > public1.335
output.csv U
Toy-Language-Lexer > output.csv

```
"\ttab"
```

```
Lexeme, Token, Count
"   tab", LITERAL, 1
```

The same functionality is applicable to all the escape sequences mentioned as well as to character, string and text block literals.

2. The lexer gives an error if it finds the first unbalanced comment or string delimiter, i.e. */, ””” or ”.

3. The Lexer terminates as soon as it encounters any of the lexical error (thus does not continue any further) and outputs the error in the terminal. Also the CSV file remains unchanged.

Boundary Cases

1. The textblock literals can handle characters like ’, ”, and \ with or without the use of escape sequences. The Lexer efficiently identifies both such cases and insert correct escape sequence in the string lexeme (refer 1st point of *Important points to know*).

2. The textblock literal (even though it supports \ without the use of double backslash), should give error if there is any character after a backslash which in combination does not forms any escape sequence. For eg. textblock literal containing \m, \j or \<space> should give error. My Lexer indeed handles

such cases and reports if any bad escape sequence is present in it or not. This functionality is available for string literals as well.

3. However there is one special boundary case when a single backslash followed by a newline character should not give any error. The lexer identifies this as well.

4. Also, even though textblocks supports double quotes(") without the use of escape sequence, three continuous double quotes are not allowed inside a textblock section. This error is also handled by the lexer.

5. I have implemented a dynamic buffer memory allocation in C, and thus the string or textblock's literals can grow indefinitely in size as well as the amount of tokens.

6. Many other lexical errors like bad escape sequences, unbalanced delimiters (like ', ", """ or */), multiline strings, multiple characters in a character literals, etc. are taken care of during lexical analysis only.

7. The Lexer handles **all** types of numeric literals, viz. integer, floating, hexadecimal, binary, octal sequences along with the latest underscore feature of Java as well.

References

I have taken help from only the below mentioned resources.

- A flex in a nutshell
- Flex Manual
- An overview of flex
- Stackoverflow