CMPT 432 - Spring 2025 - Dr. Labouseur

Regular Expressions and Test Cases for Lexer

Tyler DeLorey
tyler.delorey1@marist.edu

February 17th, 2025

Contents

1 Introduction

1.1 How the Lexer Works

Before showing the test cases and how I created my Lexer for this grammar, I want to briefly explain how I created the Lexer in the first place. Below is how I got the program from the file before even passing the program to the Lexer.

```cpp
72      // Copies all characters from the file to the code string
73      string code((istreambuf_iterator<char>(file)),
        istreambuf_iterator<char>());
74
75      // Close file
76      file.close();
77
78      // Remove trailing whitespace
79      code.erase(code.find_last_not_of(" \t\n\r\f\v") + 1);
80
81      // Vector that stores programs separated with $
82      char delimiter = '$';
83      vector<string> programs = splitPrograms(code, delimiter);
84
85      // Compile each program
86      for (int i = 0, size = programs.size(); i < size; i++)
87      {
88          cout << endl;
89
90          // LEXER
91          Lexer currentLex = Lexer(i + 1, programs[i], delimiter);
92          auto lexResult = currentLex.tokenize();
93          vector<Token> tokens = lexResult.first;
94          int errors = lexResult.second;
95          if (errors > 1)
96          {
97              continue;
98          }
99
100         // PARSER
101         // SEMANATIC ANALYSIS
102         // CODE GEN
103     }
```

Figure 1.1: Before Lexing (main.cpp)

After reading the code from the file, I passed the entire file into a variable named code, which is just a string that contains the ENTIRE contents of the file (Line 73). Then, trailing whitespace is removed for a bit of optimization because the trailing whitespace is useless and messes with the compiler for the delimiter (Line 79). After this, each program gets separated by the delimiter and gets put into a vector of strings named programs (Line 83). For this grammar, the delimiter is a $, so whenever a $ is seen, it will separate the programs. I'm not going to get into the function, but if a $ is seen inside of a comment, it will NOT separate the programs. However, it would still separate the program if it was seen in quotes, even though the programs would fail due to the unterminated string, since $ is not a valid character for this grammar. Then, each program will be compiled, starting with the Lexer tokenization, which returns a vector of Tokens and an error count, and if there were errors, it won't continue compiling that specific program (Lines 86-103).

The way I did tokenization for this compiler was use a lot of regular expressions, and I mean A LOT. In the next section I will go somewhat in depth for each regular expression I used, but basically RegEx is the reason why my Lexer works.

I will now briefly explain how the Lexing process works. The Lexer starts out my taking my entire program and it will try to match the first characters with my regular expressions (I will get into how to do that in the next section). If the first characters match with the regular expression, it will be added to a matches vector, which stores what regular expression it matched with and the actual value matched. After checking for keywords, IDs, symbols, digits, chars, and spaces (in that order), it will get the longest match. For example, if the program snippet starts with int, it will match an ID of "i" and the keyword "int", but since "int" is longer it will consume the token of keyword "int", not the ID. If for some reason they are the same length, it will take the value that is found earlier in the array to preserve rule order. Then the position "pointer" in the program will progress past that token and a new snippet of the program is checked for tokens. Of course, there are a lot of edge cases and errors/warnings to be checked when doing this, like unpaired comments, unpaired quotes, unrecognized tokens, and more. These errors/warnings are all handled by the compiler. One quick note I believe is important to mention is that even though the Lexer doesn't necessarily deal with token order and such, I decided to throw an error if quotes are unterminated because it would inevitably fail during the Parsing step. I thought to throw a warning, but the result still ends up being the same.

1.2 Test Files for Lexing

The repository has a ton of test cases in the testFiles folder specifically for testing the Lexer. More tests will be added when the parsing, semantic analysis, and code generation steps are completed, but for now, the testFiles folder contains a lot of tests dealing with really dumb edge cases for the Lexer. There are test cases for unterminated comments, multi-line comments, unterminated strings, unrecognized tokens, programs without spaces, multiple programs with spaces, and more. Section 3 will explore some of these test cases to see the results the Lexer outputs.

2 Regular Expressions

2.1 Comments and Whitespace

I will first be explaining the regular expressions used for dealing with comments and whitespace. These are special and important because the Lexer ignores comments and whitespace (unless its a space in quotes). Below are the regular expressions for comments and whitespace.

```
        const regex commentREGEX = regex(R"(^\/\*[\s\S]*?\*\/)");
        const regex commentBeginREGEX = regex(R"(^\/\*)");
        const regex commentEndREGEX = regex(R"(^\*\/)");

        const regex spaceREGEX = regex(R"(^[ \t])");
```

Figure 2.1: Comments and Whitespace RegEx (Lexer.h)

Before explaining each RegEx, I think it is important to explain what the ˆ (carat) sign entails. This character ensures that it finds a match in the BEGINNING of the string. This is why the entire Lexer works and this is how it matches the first characters with my regular expressions using the program snippets. It will only match if it occurs at the beginning of the snippet.

It is also important to explain what the R"(...)" means inside of the regex() function. This is for raw string literals, so I don't have to escape special characters an extra time. Normally there needs to be a double escape for special characters inside of RegEx. For example, if the regular expression wants to include a tab, it would be written as \\t, but now it could be written as \t since the RegEx uses R.

Let's break down each REGEX for comments and whitespace with these basics now explained.

- The commentREGEX will match a full comment at the start of the string (/* ... */). The / and the * need to be escaped, which is why it is written as \/*. The [\s\S]*? matches any characters within the comments, including whitespace. The *? matches as many as possible (until it reaches the next set of \/*). The ? signifies that it doesn't use a greedy approach when matching comments, meaning it will always match with the first set of \/*.
- The commentBeginREGEX will match the first part of a comment at the start of the string (/*). This may seem useless, but it is specifically created for error/warning checking. Since the Lexer first checks for the regular commentREGEX, if it doesn't match that, but it matches the commentBeginREGEX, that means there is an unterminated comment. This will return a warning.

- The commentEndREGEX will match the last part of a comment at the start of the string (*/). Just like commentBeginREGEX, this is created for error/warning checking. If the snippet doesn't match commentREGEX but it matches commentEndREGEX, that means there was no beginning to the comment, so it returns an error since this would be considered as unrecognized tokens anyways.
- The spaceREGEX will match a space or a tab at the start of the string. Note that this grammar doesn't necessarily support tabs. Even though it is whitespace and it doesn't really matter, there is a case where the whitespace and tabs do matter, which is in quotes. Inside of quotes, the spaceREGEX is used to test for the space character, and I decided to convert tabs into spaces for better writability for the code for the compiler.

2.2 The Regular Rules

Now it is time to analyze the regular expressions for the keywords, IDs, symbols, digits, and characters. Below is my implementation of these regular expressions.

```
        const regex keywordREGEX = regex("^(print|while|if|int|string|boolean|true|false)");
        const regex idREGEX = regex(R"(^[a-z])");
        const regex symbolREGEX = regex(R"(^(\{|\}|"|\(|\)|==|!=|\+|=|\$|\r?\n))");
        const regex boolOPREGEX = regex(R"(^(!\/\*[\s\S]*?\*\/=|=\/\*[\s\S]*?\*\/=))");
        const regex digitREGEX = regex(R"(^[0-9])");
        const regex charREGEX = regex(R"(^[a-z])");
```

Figure 2.2: Regular RegEx (Lexer.h)

Let's break down each REGEX

- The keywordREGEX will match a full keyword at the start of the string. The keywords for this grammar are print, while, if, int, string, boolean, true, and false. If one of these words are found, it matches this RegEx. In case if it wasn't obvious, the | operator represents OR, meaning it only has to match one of these for it to match the RegEx.
- The idREGEX will match an ID, which is any single lowercase letter from a-z at the start of the string. Notice how IDs can ONLY be one letter, which is why there isn't any + at the end of the RegEx (+ means one or more, but since the grammar only has one letter to be an ID, that means it doesn't need this).
- The symbolREGEX will match any symbol at the start of the string. The symbols for this grammar are , , ", (,), ==, !=, +, =, $, and a newline. Even though the $ and newline aren't really part of the grammar, they

are used for error/warning checking and making sure the line and column variables match up with what they are supposed to be. The RegEx looks very confusing because of the amount of | operators and the parenthesis and curly braces, but it works out at the end.

- The boolOPREGEX will be explained in the next section, since it's only purpose is for ONE STUPID TEST CASE.
- The digitREGEX will match a digit, which is any single digit from 0-9 at the start of the string. Notice how digits are singular, which, like the IDs, is why there isn't any + at the end of the RegEx.
- The charREGEX will match a char, which is any single lowercase letter from a-z at the start of the string. Notice how this is exactly the same as IDs. However, this is only checked if the Lexer detects that it is in quotes, so it will treated as a character instead of an ID.

2.3 Special Case For Bool Operators

There is a specific RegEx specifically for if a comment separates either the equality symbol (==) or inequality symbol (!=), as shown below.

```
        const regex boolOPREGEX = regex(R"(^(!\/\*[\s\S
]*?\*\/=|=\/\*[\s\S]*?\*\/=))");
```

Figure 2.3: BoolOp RegEx (Lexer.h)

- The boolOPREGEX will match if the bool operators are separated by a comment (!/*...*/= OR =/*...*/=). This will be treated as a normal bool operator, but there needed to be a special RegEx for this specific scenario. This uses syntax from the commentREGEX and the symbolREGEX.

3 Test Cases

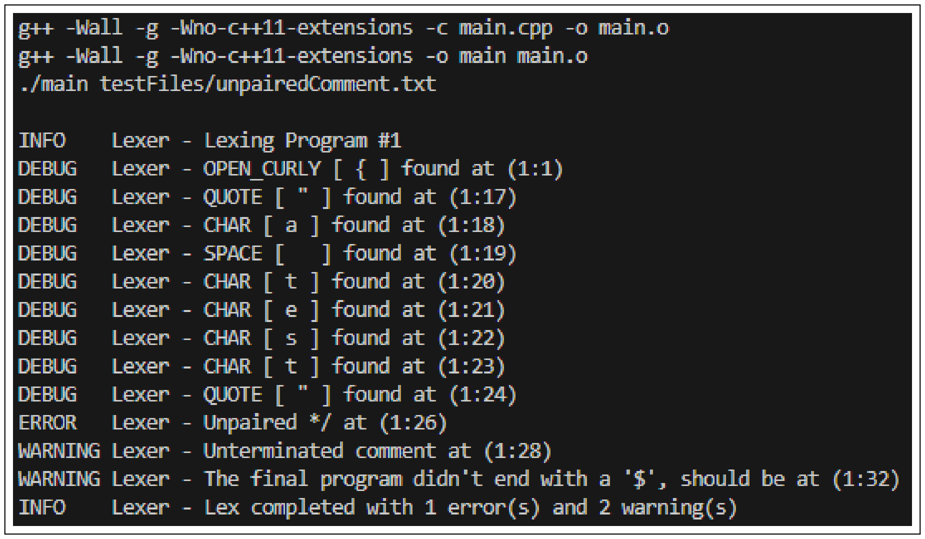
3.1 unpairedComment.txt

The first test case I will be going over is unpairedComment.txt. This deals with the edge case that comments are unterminated or unpaired, as shown below.

```
{/*/*This is */ "a test" *//*}$
```

Figure 3.1: unpairedComment.txt

Below is my Lexer output for this program.



```
g++ -Wall -g -Wno-c++11-extensions -c main.cpp -o main.o
g++ -Wall -g -Wno-c++11-extensions -o main main.o
./main testFiles/unpairedComment.txt

INFO    Lexer - Lexing Program #1
DEBUG   Lexer - OPEN_CURLY [ { ] found at (1:1)
DEBUG   Lexer - QUOTE [ " ] found at (1:17)
DEBUG   Lexer - CHAR [ a ] found at (1:18)
DEBUG   Lexer - SPACE [   ] found at (1:19)
DEBUG   Lexer - CHAR [ t ] found at (1:20)
DEBUG   Lexer - CHAR [ e ] found at (1:21)
DEBUG   Lexer - CHAR [ s ] found at (1:22)
DEBUG   Lexer - CHAR [ t ] found at (1:23)
DEBUG   Lexer - QUOTE [ " ] found at (1:24)
ERROR   Lexer - Unpaired */ at (1:26)
WARNING Lexer - Unterminated comment at (1:28)
WARNING Lexer - The final program didn't end with a '$', should be at (1:32)
INFO    Lexer - Lex completed with 1 error(s) and 2 warning(s)
```

Figure 3.2: unpairedCommentResults Output

Notice how ”This is” is in comments, while the ”a test” are counted as characters and spaces, and then when it reaches the */ it will throw an error that it is unpaired and those tokens wouldn't be recognized. Then, since a new comment starts, it will throw a warning saying there is an unterminated comment, and since the delimiter ($) is technically inside of comments, there will also be a warning about the program not ending with a $.

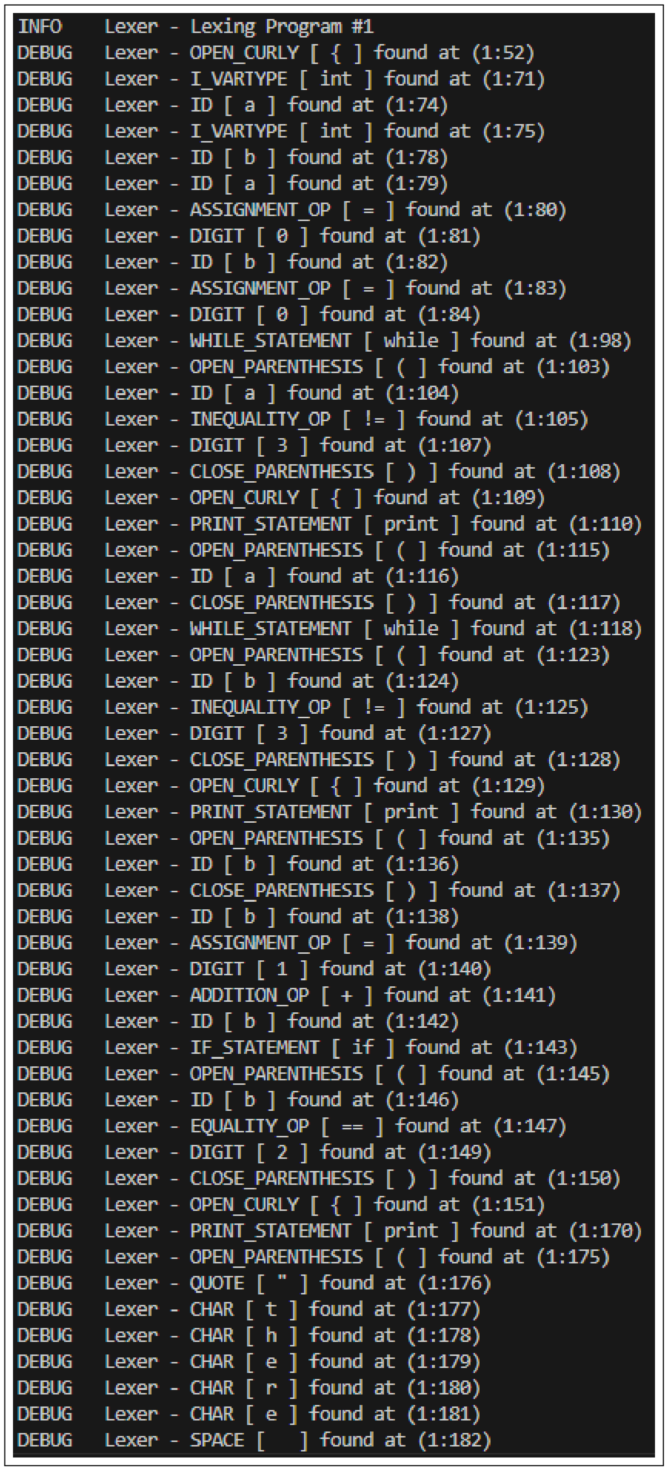
3.2 lexWithoutSpace.txt

The second test case I will be going over is lexWithoutSpace.txt. This deals with the edge case that there are no spaces between any keyword, IDs, and symbols.

```
/*LongTestCase-EverythingExceptBooleanDeclaration*/{/*
    IntDeclaration*/intaintba=0b=0/*WhileLoop*/while(a!=3){print(a)
    while(b!=3){print(b)b=1+bif(b==2){/*PrintStatement*/print("
    there is no spoon"/*Thiswilldonothing*/)}}b=0a=1+a}}$
```

Figure 3.3: lexWithoutSpace.txt

Below is my Lexer output for this program.



```
INFO    Lexer - Lexing Program #1
DEBUG   Lexer - OPEN_CURLY [ { ] found at (1:52)
DEBUG   Lexer - I_VARTYPE [ int ] found at (1:71)
DEBUG   Lexer - ID [ a ] found at (1:74)
DEBUG   Lexer - I_VARTYPE [ int ] found at (1:75)
DEBUG   Lexer - ID [ b ] found at (1:78)
DEBUG   Lexer - ID [ a ] found at (1:79)
DEBUG   Lexer - ASSIGNMENT_OP [ = ] found at (1:80)
DEBUG   Lexer - DIGIT [ 0 ] found at (1:81)
DEBUG   Lexer - ID [ b ] found at (1:82)
DEBUG   Lexer - ASSIGNMENT_OP [ = ] found at (1:83)
DEBUG   Lexer - DIGIT [ 0 ] found at (1:84)
DEBUG   Lexer - WHILE_STATEMENT [ while ] found at (1:98)
DEBUG   Lexer - OPEN_PARENTHESIS [ ( ] found at (1:103)
DEBUG   Lexer - ID [ a ] found at (1:104)
DEBUG   Lexer - INEQUALITY_OP [ != ] found at (1:105)
DEBUG   Lexer - DIGIT [ 3 ] found at (1:107)
DEBUG   Lexer - CLOSE_PARENTHESIS [ ) ] found at (1:108)
DEBUG   Lexer - OPEN_CURLY [ { ] found at (1:109)
DEBUG   Lexer - PRINT_STATEMENT [ print ] found at (1:110)
DEBUG   Lexer - OPEN_PARENTHESIS [ ( ] found at (1:115)
DEBUG   Lexer - ID [ a ] found at (1:116)
DEBUG   Lexer - CLOSE_PARENTHESIS [ ) ] found at (1:117)
DEBUG   Lexer - WHILE_STATEMENT [ while ] found at (1:118)
DEBUG   Lexer - OPEN_PARENTHESIS [ ( ] found at (1:123)
DEBUG   Lexer - ID [ b ] found at (1:124)
DEBUG   Lexer - INEQUALITY_OP [ != ] found at (1:125)
DEBUG   Lexer - DIGIT [ 3 ] found at (1:127)
DEBUG   Lexer - CLOSE_PARENTHESIS [ ) ] found at (1:128)
DEBUG   Lexer - OPEN_CURLY [ { ] found at (1:129)
DEBUG   Lexer - PRINT_STATEMENT [ print ] found at (1:130)
DEBUG   Lexer - OPEN_PARENTHESIS [ ( ] found at (1:135)
DEBUG   Lexer - ID [ b ] found at (1:136)
DEBUG   Lexer - CLOSE_PARENTHESIS [ ) ] found at (1:137)
DEBUG   Lexer - ID [ b ] found at (1:138)
DEBUG   Lexer - ASSIGNMENT_OP [ = ] found at (1:139)
DEBUG   Lexer - DIGIT [ 1 ] found at (1:140)
DEBUG   Lexer - ADDITION_OP [ + ] found at (1:141)
DEBUG   Lexer - ID [ b ] found at (1:142)
DEBUG   Lexer - IF_STATEMENT [ if ] found at (1:143)
DEBUG   Lexer - OPEN_PARENTHESIS [ ( ] found at (1:145)
DEBUG   Lexer - ID [ b ] found at (1:146)
DEBUG   Lexer - EQUALITY_OP [ == ] found at (1:147)
DEBUG   Lexer - DIGIT [ 2 ] found at (1:149)
DEBUG   Lexer - CLOSE_PARENTHESIS [ ) ] found at (1:150)
DEBUG   Lexer - OPEN_CURLY [ { ] found at (1:151)
DEBUG   Lexer - PRINT_STATEMENT [ print ] found at (1:170)
DEBUG   Lexer - OPEN_PARENTHESIS [ ( ] found at (1:175)
DEBUG   Lexer - QUOTE [ " ] found at (1:176)
DEBUG   Lexer - CHAR [ t ] found at (1:177)
DEBUG   Lexer - CHAR [ h ] found at (1:178)
DEBUG   Lexer - CHAR [ e ] found at (1:179)
DEBUG   Lexer - CHAR [ r ] found at (1:180)
DEBUG   Lexer - CHAR [ e ] found at (1:181)
DEBUG   Lexer - SPACE [   ] found at (1:182)
```

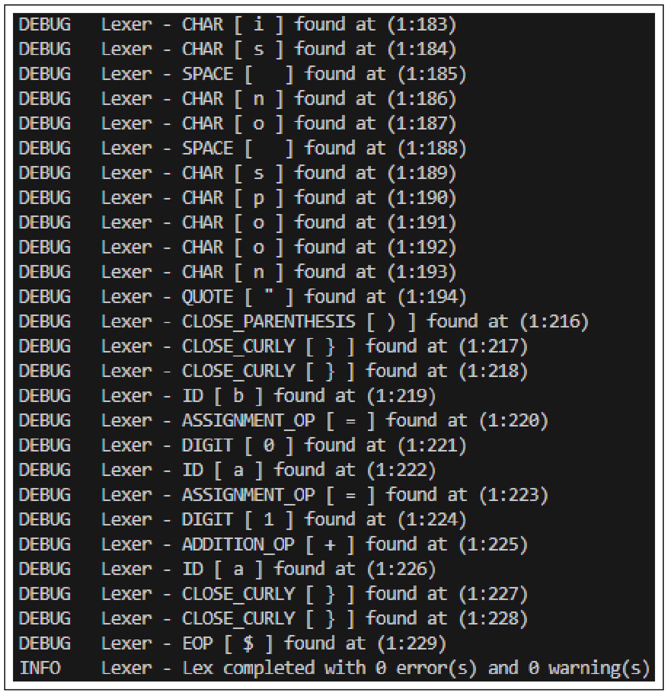

```
DEBUG   Lexer - CHAR [ i ] found at (1:183)
DEBUG   Lexer - CHAR [ s ] found at (1:184)
DEBUG   Lexer - SPACE [   ] found at (1:185)
DEBUG   Lexer - CHAR [ n ] found at (1:186)
DEBUG   Lexer - CHAR [ o ] found at (1:187)
DEBUG   Lexer - SPACE [   ] found at (1:188)
DEBUG   Lexer - CHAR [ s ] found at (1:189)
DEBUG   Lexer - CHAR [ p ] found at (1:190)
DEBUG   Lexer - CHAR [ o ] found at (1:191)
DEBUG   Lexer - CHAR [ o ] found at (1:192)
DEBUG   Lexer - CHAR [ n ] found at (1:193)
DEBUG   Lexer - QUOTE [ " ] found at (1:194)
DEBUG   Lexer - CLOSE_PARENTHESIS [ ) ] found at (1:216)
DEBUG   Lexer - CLOSE_CURLY [ } ] found at (1:217)
DEBUG   Lexer - CLOSE_CURLY [ } ] found at (1:218)
DEBUG   Lexer - ID [ b ] found at (1:219)
DEBUG   Lexer - ASSIGNMENT_OP [ = ] found at (1:220)
DEBUG   Lexer - DIGIT [ 0 ] found at (1:221)
DEBUG   Lexer - ID [ a ] found at (1:222)
DEBUG   Lexer - ASSIGNMENT_OP [ = ] found at (1:223)
DEBUG   Lexer - DIGIT [ 1 ] found at (1:224)
DEBUG   Lexer - ADDITION_OP [ + ] found at (1:225)
DEBUG   Lexer - ID [ a ] found at (1:226)
DEBUG   Lexer - CLOSE_CURLY [ } ] found at (1:227)
DEBUG   Lexer - CLOSE_CURLY [ } ] found at (1:228)
DEBUG   Lexer - EOP [ $ ] found at (1:229)
INFO    Lexer - Lex completed with 0 error(s) and 0 warning(s)
```

Figure 3.4: lexWithoutSpacesResults Output

Notice how even though there are no spaces, the compiler doesn't care about it and it all works perfectly well. There is another file named lexWithSpaces in the testFiles folder and the output for the Lexer for both of these files are the exact same. Spaces don't matter for this grammar.

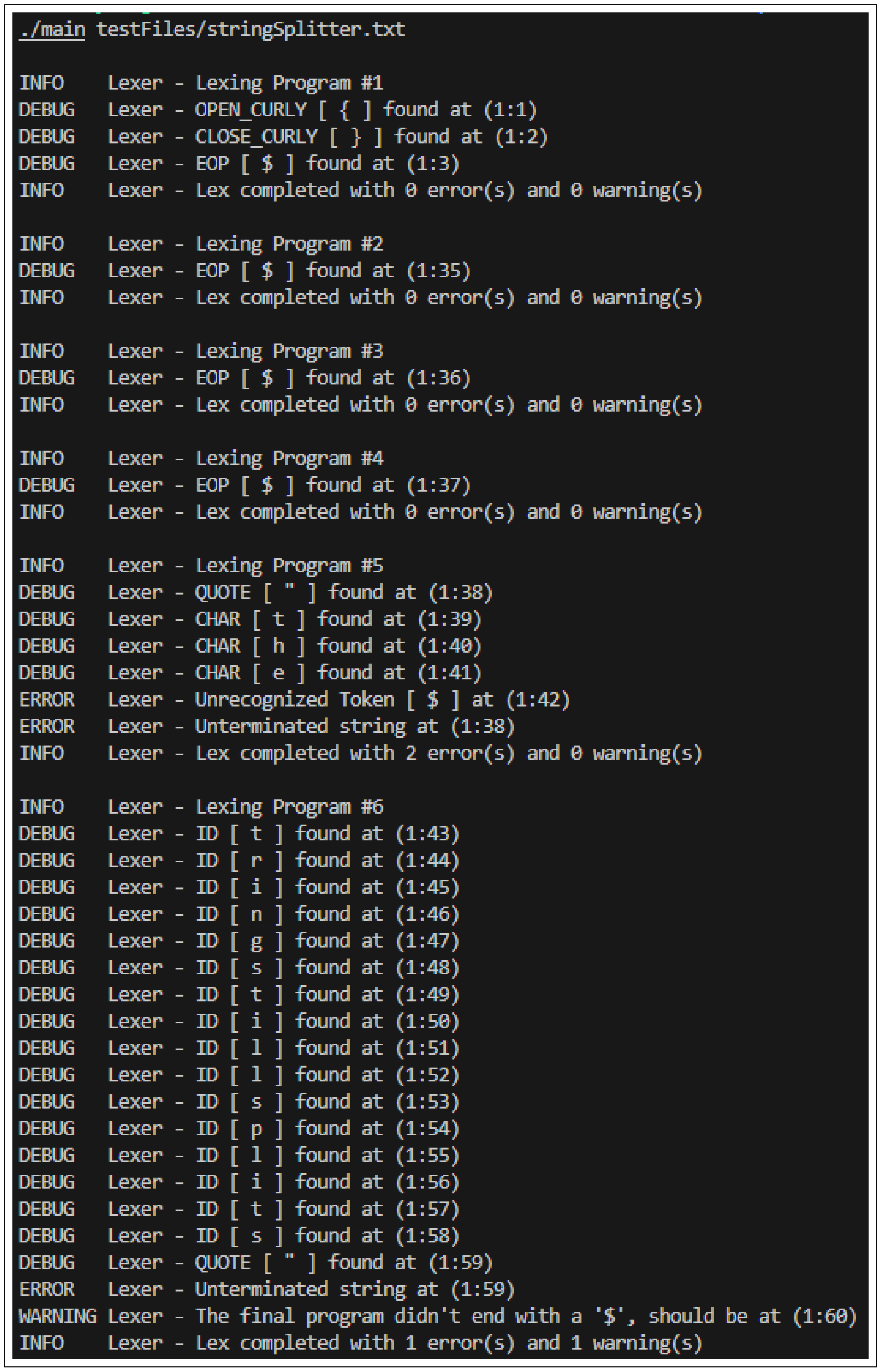
3.3 stringSplitter.txt

The third test case I will be going over is stringSplitter.txt. This deals with the edge case that the $ delimiter is inside of comments.

```
{}$/*$$$ This should be ignored */$$$"the$tringstillsplits"
```

Figure 3.5: stringSplitter.txt

Below is my Lexer output for this program.



```
./main testFiles/stringSplitter.txt

INFO    Lexer - Lexing Program #1
DEBUG   Lexer - OPEN_CURLY [ { ] found at (1:1)
DEBUG   Lexer - CLOSE_CURLY [ } ] found at (1:2)
DEBUG   Lexer - EOP [ $ ] found at (1:3)
INFO    Lexer - Lex completed with 0 error(s) and 0 warning(s)

INFO    Lexer - Lexing Program #2
DEBUG   Lexer - EOP [ $ ] found at (1:35)
INFO    Lexer - Lex completed with 0 error(s) and 0 warning(s)

INFO    Lexer - Lexing Program #3
DEBUG   Lexer - EOP [ $ ] found at (1:36)
INFO    Lexer - Lex completed with 0 error(s) and 0 warning(s)

INFO    Lexer - Lexing Program #4
DEBUG   Lexer - EOP [ $ ] found at (1:37)
INFO    Lexer - Lex completed with 0 error(s) and 0 warning(s)

INFO    Lexer - Lexing Program #5
DEBUG   Lexer - QUOTE [ " ] found at (1:38)
DEBUG   Lexer - CHAR [ t ] found at (1:39)
DEBUG   Lexer - CHAR [ h ] found at (1:40)
DEBUG   Lexer - CHAR [ e ] found at (1:41)
ERROR   Lexer - Unrecognized Token [ $ ] at (1:42)
ERROR   Lexer - Unterminated string at (1:38)
INFO    Lexer - Lex completed with 2 error(s) and 0 warning(s)

INFO    Lexer - Lexing Program #6
DEBUG   Lexer - ID [ t ] found at (1:43)
DEBUG   Lexer - ID [ r ] found at (1:44)
DEBUG   Lexer - ID [ i ] found at (1:45)
DEBUG   Lexer - ID [ n ] found at (1:46)
DEBUG   Lexer - ID [ g ] found at (1:47)
DEBUG   Lexer - ID [ s ] found at (1:48)
DEBUG   Lexer - ID [ t ] found at (1:49)
DEBUG   Lexer - ID [ i ] found at (1:50)
DEBUG   Lexer - ID [ l ] found at (1:51)
DEBUG   Lexer - ID [ l ] found at (1:52)
DEBUG   Lexer - ID [ s ] found at (1:53)
DEBUG   Lexer - ID [ p ] found at (1:54)
DEBUG   Lexer - ID [ l ] found at (1:55)
DEBUG   Lexer - ID [ i ] found at (1:56)
DEBUG   Lexer - ID [ t ] found at (1:57)
DEBUG   Lexer - ID [ s ] found at (1:58)
DEBUG   Lexer - QUOTE [ " ] found at (1:59)
ERROR   Lexer - Unterminated string at (1:59)
WARNING Lexer - The final program didn't end with a '$', should be at (1:60)
INFO    Lexer - Lex completed with 1 error(s) and 1 warning(s)
```

Figure 3.6: stringSplitterResults Output

Notice how whenever the $ are in comments, they don't actually split the program at all. Also notice how when the quotes start and the program ends it gives an unterminated string error and the next program is all messed up. This is intentional because this should fail, and it does.

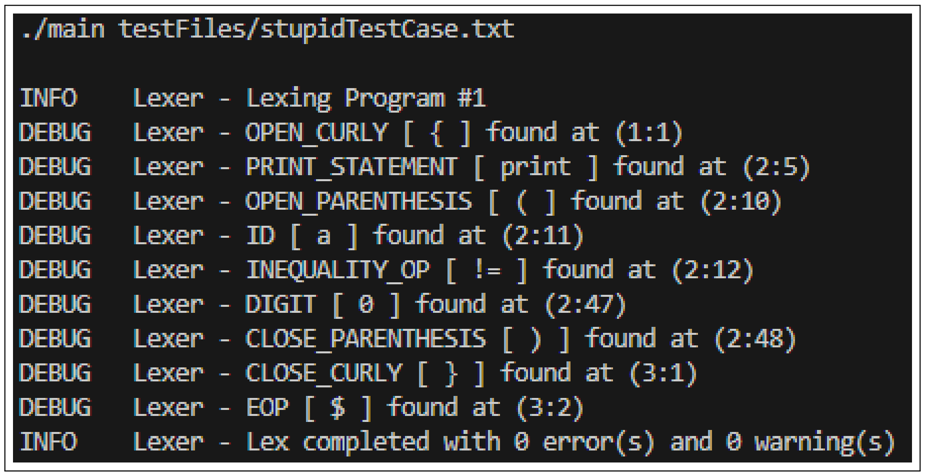
3.4 stupidTestCase.txt

The last test case I will be going over is stupidTestCase.txt. This deals with the edge case that the boolean operator is split by the string, as was mentioned with the special regular expressions that had to be created to deal with this issue.

```
{
    print(a!/*Comment in between inequality*/=0)
}$
```

Figure 3.7: stupidTestCase.txt

Below is my Lexer output for this program.



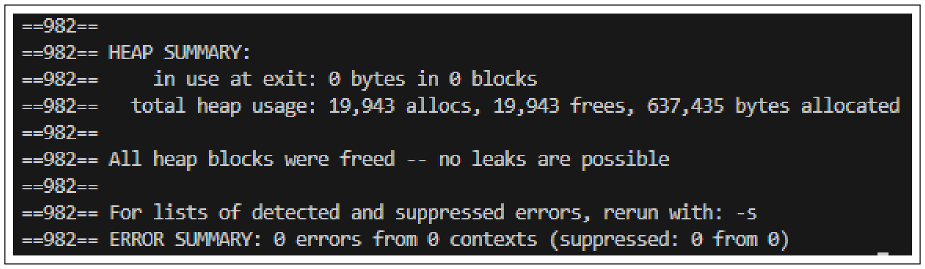
```
./main testFiles/stupidTestCase.txt

INFO    Lexer - Lexing Program #1
DEBUG   Lexer - OPEN_CURLY [ { ] found at (1:1)
DEBUG   Lexer - PRINT_STATEMENT [ print ] found at (2:5)
DEBUG   Lexer - OPEN_PARENTHESIS [ ( ] found at (2:10)
DEBUG   Lexer - ID [ a ] found at (2:11)
DEBUG   Lexer - INEQUALITY_OP [ != ] found at (2:12)
DEBUG   Lexer - DIGIT [ 0 ] found at (2:47)
DEBUG   Lexer - CLOSE_PARENTHESIS [ ) ] found at (2:48)
DEBUG   Lexer - CLOSE_CURLY [ } ] found at (3:1)
DEBUG   Lexer - EOP [ $ ] found at (3:2)
INFO    Lexer - Lex completed with 0 error(s) and 0 warning(s)
```

Figure 3.8: stupidTestCaseResults Output

Notice how even though there is a comment in between the inequality symbol, it still recognizes the symbol and the program finishes with no errors or warnings.

4 Conclusion

There are many more test cases that I have in the testFiles folder, but these are all the special ones to me. It took me a lot of time to deal with the edge cases and to deal with the regular expressions to work correctly. However, I got it working at the end. :)



```
==982==
==982== HEAP SUMMARY:
==982==     in use at exit: 0 bytes in 0 blocks
==982==   total heap usage: 19,943 allocs, 19,943 frees, 637,435 bytes allocated
==982==
==982== All heap blocks were freed -- no leaks are possible
==982==
==982== For lists of detected and suppressed errors, rerun with: -s
==982== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 4.1: Valgrind Output