Laboratory 4

Jorge Adrian Padilla Velasco

***Abstract*— The objective of this laboratory is to understand the power of the LEX language. To do so, we compared two files (lex.c and compiler.l) that do a lexical analysis of a third file (random_code.ac).**

I. INTRODUCTION

LEX is a program designed to generate scanners, also known as tokenizers, which recognize lexical patterns in text. Lex is an acronym that stands for ”lexical analyzer generator.” It is intended primarily for Unix-based systems. The code for Lex was originally developed by Eric Schmidt and Mike Lesk.

Lex can perform simple transformations by itself but its main purpose is to facilitate lexical analysis, the processing of character sequences such as source code to produce symbol sequences called tokens for use as input to other programs such as parsers. Lex can be used with a parser generator to perform lexical analysis. It is easy, for example, to interface Lex and Yacc, an open source program that generates code for the parser in the C programming language.

Lex is proprietary but versions based on the original code are available as open source. These include a streamlined version called Flex, an acronym for ”fast lexical analyzer generator,” as well as components of OpenSolaris and Plan 9.

II. PROBLEM DESCRIPTION

We have to generate a LEX code (compiler.l) to parse an ac file (example.ac). For example, the ac file could be something like:

```
//basic code
//float b
f b

// integer a
i a

// a = 5
a = 5

// b = a + 3.2
b = a + 3.2

//print 8.5
p b
```

And the output should be something like:

```
COMMENT
COMMENT
floatdcl id
COMMENT
intdcl id
COMMENT
id assign inum
COMMENT
id assign id plus fnum
COMMENT
print id
```

The LEX code should compile with a Makefile as follows:

```
make
lex compiler.l
gcc lex.yy.c -o compiler -ll
```

In order to generate the ac file, we have to add –stress as a parameter when executing the python script that was given to us so it generates stress examples for us to try:

```
python3 code_generator.py --stress
```

This will generate a huge ac random code (random_code.ac). Now we have to run our solution (compiler.l) and check how much time it takes to do the LEX part of the compiler.

Once we have done that, now we have to compare that time with the time it takes for the code we generated (lex.c) for the previous laboratory (lab 3) to run, using the same random_code.ac file.

III. SOLUTION

First, we try our LEX file with the huge random ac code generated by running:

```
make
time ./compiler random_code.ac
```

The code can be checked out on: GitHub.

Now, we do the same thing with our C code (lex.c) we created for laboratory 3:

```
make
time ./lex random_code.ac
```

The code can be checked out on: GitHub.

IV. RESULTS

After executing our LEX code (compiler.l), we get the following output regarding the execution time taken:

```
real    0m3.218s
user    0m0.382s
```

```
sys     0m0.838s
```

After executing our C code (lex.c), we get the following output regarding the execution time taken:

```
real    0m2.753s
user    0m0.232s
sys     0m0.716s
```

To visualize better these outputs, we compared them using a graph:

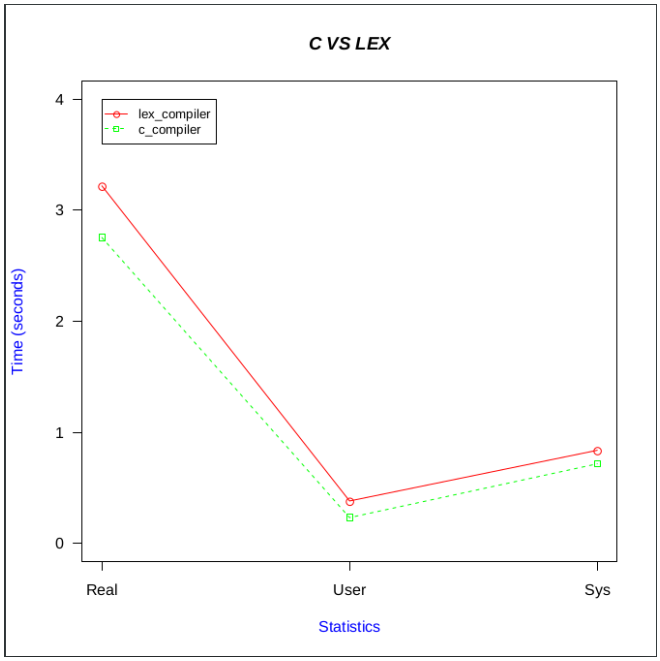

Fig. 1. C vs LEX execution time.

V. CONCLUSIONS

The token descriptions that LEX uses are known as regular expressions, extended versions of the familiar patterns used by the grep and egrep commands. LEX turns these regular expressions into a form that the lexer can use to scan the input text extremely fast, independent of the number of expressions that it is trying to match. A LEX lexer is almost always faster than a lexer that we might write in C. An example where this is not the case, is the one presented on this paper. As we can see in Fig. 1, on the RESULTS section, the C compiler written for laboratory 3 was a tiny bit faster than the LEX compiler written for laboratory 4.

As the input is divided into tokens, a program often needs to establish the relationship among the tokens. A C compiler needs to find the expressions, statements, declarations, blocks, and procedures in the program. This task is known as parsing and the list of rules that define the relationships that the program understands is a grammar. Yacc takes a concise description of a grammar and produces a C routine that can parse that grammar, a parser. The yacc parser automatically detects whenever a sequence of input tokens matches one of the rules in the grammar and also detects a syntax error whenever its input does not match any of the rules. A yacc parser is generally not as fast as a parser you could write by hand, but the ease in writing and modifying the parser is invariably worth any speed loss. The amount of time a program spends in a parser is rarely enough to be an issue anyway.

REFERENCES

[1] T. Mason, D. Brown, J. Levine, Lex & Yacc, 2nd edition.
[2] M. Rose, Lex (lexical analyzer generator). Recovered from: https://whatis.techtarget.com/definition/Lex-lexical-analyzer-generator