Compilers Lab 04: Lexical analyzer using lex

A00226860 - Gerardo Juarez

***Abstract*— Coding a compiler from scratch is really hard if you use only the theory behind it, however, there are many tools that makes the development of a compiler easier for a software developer. One example of those tools is the scanner generator, as it names implies, it is a tool that generates a lexical analyzer or a scanner for a language. The code created by this tool matches strings with regular expressions instead of the naive approach of if-else statements all over the code.**

The generated code and the naive approach were compared in time consumption to figure out whether the automated code is not only easier to implement but faster for a complete compiler.

I. INTRODUCTION

The lexical analyzer or scanner, is the first step of a compiler that generates a stream of tokens from the source code. In order to do this, the scanner must be able to identify and match specific sequences of characters to tag them with different identifiers that will be used for the next phases of the compiler. This process might be annoying if it is coded from scratch using many if-else statements in any programming language and slow since many comparisons will be executed for each character of the stream received as input.

There exists tools that helps the software developer in the construction of the compiler. One of this example is the Lex program that creates a scanner in C language, it only needs the set of regular expressions to match the defined tokens. This saves the time of coding the complete scanner.

II. PROBLEM DESCRIPTION

Compilers must be efficient and fast because large software products will consume lot of time in just the compiling. This is why the optimization of each phase of the compiler is wanted.

Lex is a tool that makes the life of a compiler developer easier, but this might be slower than the naive approach, or not. Whether Lex is used or not, the important thing to compare is if the matching of strings with regular expressions is faster than the use of if-else statements.

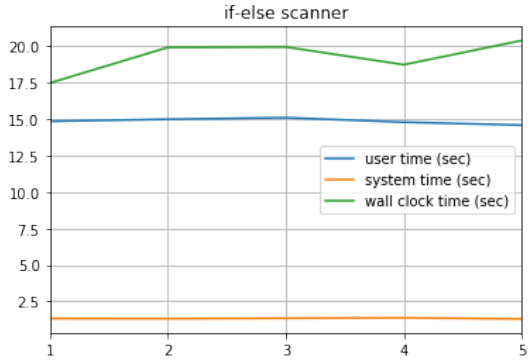
III. SOLUTION

Two scanners were coded for the experiment, one using Lex and the other using if-else statements. Both of them were run five different files with 10 millions of lines. The scanners detects tokens of the AC Language described in the book ”Crafting a Compiler”.

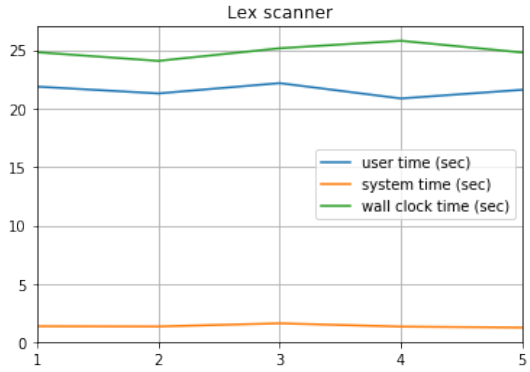
In order to measure the time spent for each program, the command ”time” of Linux was used like this: time -a -v -o output-file ./compiler random-code.ac, where the ’.ac’ file is the code for the AC language and the ’compiler’ is the name of the executable generated when compiling the scanners.

IV. RESULTS

The plot below shows the run time in seconds of the naive scanner using if-else statements. The x-axis represents one file, and the y-axis the duration of execution of the program. The wall clock time, or real time, represents the total amount of time, including both user time and kernel time spent.



On the other hand, the plot below shows the run time in seconds of the Lex scanner. It was expected to be faster than the naive scanner, however, it had worse performance. While the naive scanner time was around 17.5 to 20.0 seconds, the Lex scanner was around 25.0 seconds.



V. CONCLUSIONS

The Lex scanner was expected to be faster, but according to the results, it was not. There are many variables to take into consideration to be certain. For example, the compiler may have optimized the naive code to make it faster, but this would mean that the code generated by Lex was optimized too, hence, the compiler might not be the problem.

Another reason could be that the ’fprintf’ function inside the ’compiler.l’, used to write the tokens into a file, made it slower. This because a scanner should not do this kind of thing, it should pass the token stream into the parser without writing into disk or it would run slower. In other words, the C file generated by Lex may not perform well when

the program needs to perform complex operations, such as writing to a file, for each string of characters that is converted into a token.