Universidad Nacional Autónoma de México

Computer Engineering

Compilers

Students

Names	**Account Numbers**
Cuevas Quintana Amir	320166866
Guerrero Lopez Uriel Ivan	320046045
Perez Rojo Abraham	320149517
Sandoval Hernandez Darinka	320137156
Sanchez Mayen Tristan Qesen	320187072

Group:
5
Semester:
2025 - 2

Mexico, CDMX. February 27th 2025

Contents

1 Introduction

- **Problem statement:** The problem we need to solve is to implement the FLEX app, which is a lexical analyzer. By providing it with a text and defining certain rules, we can analyze specific elements in a string, such as the number of certain characters, numbers, specific letters, and more.
- **Motivation:** Understanding how to use this tool allows us to gain a deeper understanding of how lexical analyzers work, offering two main benefits: better comprehension of this process and the ability to apply the acquired knowledge to our final project for the Compilers course.
- **Objectives:** Our objective is to learn how to use a lexical analyzer and, with this knowledge, build our own by researching the the specific techniques required to complete this task.

2 Theorical Framework

To understand how the lexical analyzer works, we use theoretical concepts learned in the Compilers and Formal Languages and Automata courses. The key concepts we are applying are:

- Flex first takes the regular expressions provided by the user. Each of these expressions describes a pattern that should match specific tokens in the input text. For example, if we provide [0-9] as input, Flex interprets this as a regular expression, and every match is considered a number token.
- Flex then creates an NFA (Nondeterministic Finite Automaton) to read the regular expressions given by the user and converts the NFA into a DFA (Deterministic Finite Automaton). This conversion makes the input analysis faster by enabling the creation of a transition table (similar to a symbol table), thus speeding up the translation process.
- Using the DFA, the lexer reads the input until it reaches a valid token as defined by the regular expressions.
- To avoid ambiguity, Flex uses a two-rule disambiguation strategy when multiple patterns match the same input:
  - Longest Match Rule (Maximal Munch): If two or more patterns can match the current input, Flex chooses the longest possible match.
  - Rule Order Priority: If multiple rules match the same length of input, Flex picks the one that appears first in the rule list.
- This explanation involves theoretical concepts from Formal Languages and Automata, as well as discrete structures. Additionally, from the Compilers perspective, the main concept used is the lexical analyzer itself, as this is the first stage where a compiler matches a given input with the previously created automaton.

3 Body

What we did to use the Flex analyzer was modify the source code. In the beginning, we added a global variable that counts the number of incidences the lexical analyzer gives us. Then, between the percentage symbols, we added the pattern that we wanted Flex to search for. In this case, we used these three patterns:

- $[-09]$
- $[w(x \mid y)z]$
- $[a+]$

3.1 How does Flex understand the [-09] pattern

The pattern $[-09]$ is read by Flex, it matches any single character from the set defined inside the square brackets, the way Flex interprets this pattern follows certain rules, first, the square brackets define a set of valid characters, so any single character inside them is considered a valid match according to the rules that we defined previously. The hyphen $-$ is treated as a literal character because it is placed at the beginning of the set. Normally, a hyphen inside a character class defines a range, but when it appears at the start or end, it is recognized as a normal character rather than a range operator, also, characters 0 and 9 are matched individually, as they do not make a numeric range, concluding, as a result, every time one of these three characters appears in the input, the corresponding Flex action associated with the pattern will be executed.

3.2 How does Flex understand the [w(x|y)z] pattern

The $[w(x \mid y)z]$ pattern detects the character group 'wxz' or 'wyz' and the symbol | works like an or, so it can be either an 'x' or a 'y' between the 'w' and 'z'. The pattern $[w(x \mid y)z]$ is a regular expression because it follows the structure of one. Use parentheses to refer to a concatenation and the $'|'$ symbol like an or, that is, it can be an 'x' or a 'y'. So it lets the FLEX accept or read two strings, these are 'wxz' and 'wyz'. This is a fairly easy pattern to understand since FLEX expects only two strings. Knowing that FLEX reads one character at a time from left to right, when it reads a 'w' then it can go to the next state, which can be either an 'x' or a 'y', in any other case then the string will not increment the counter. After having read any of the previously mentioned characters, it goes to the next state where it expects a 'z', fulfilling one of the strings it really expects, otherwise it will not increment the counter and will continue reading the entry.

3.3 How does Flex understand the [a+] pattern

The pattern a+ is a regular expression because it defines a clear pattern and follows the rules of a regular expression. It represents one or more consecutive

occurrences of the letter ‘a’. Now, we know that Flex scans the input text from left to right, reading character by character. In this case, when it encounters a sequence of a characters, it triggers the a+ rule and executes the associated action (which, in the code, would be incrementing the $no_{ap}lus$ counter)

Flex always tries to make the longest possible match. So, if there are multiple consecutive ‘a’ characters, Flex groups them as a single match of a+. For example, in the input “holaaa”, Flex scans the first three characters ‘h’, ‘o’, and ‘l’, which do not match the a+ rule. However, it then finds three grouped ‘a’ characters that match a+, so it increments the counter by 1 (applying the associated rule). For the input “aaa”, Flex increments the counter only once, because, as mentioned earlier, it always makes the longest match. Finally, with “banana” (which also applies to “manzana”), Flex reads the first character ‘b’, which does not match the rule. Then, it moves to the next character, ‘a’, which does match. However, when it moves to the next character, ‘n’, it no longer matches the rule, so it increments the counter at that point. This process repeats as Flex continues scanning each line character by character.

4 Results

Using screenshots and a short description after the screenshot, we show the final results.

Inputs:

- oaso0 ashd- hashdsa9 ajdjsajd90—dasd

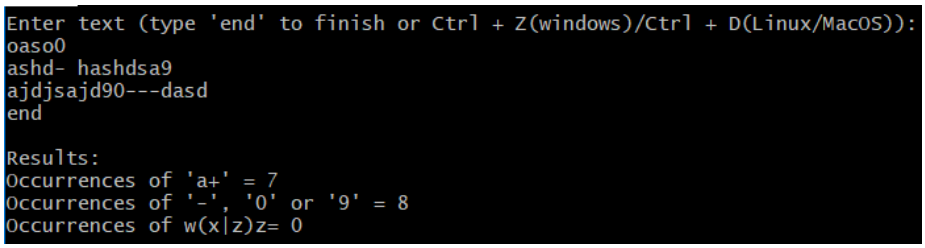

Figure 1: Screenshot of the pattern [-09] execution

- holaaa, aaa, a, banana, manzana

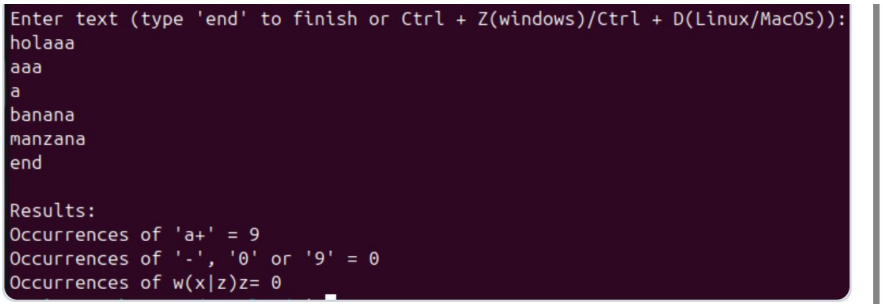

Figure 2: Screenshot of the pattern [a+] execution

– wxz, wxy

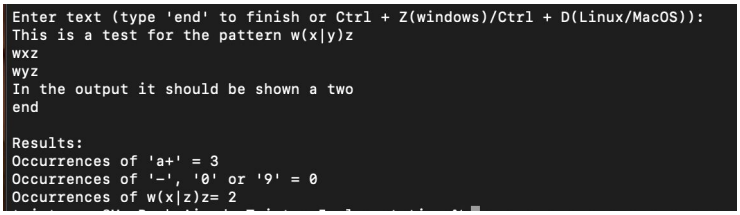

Figure 3: Screenshot of the pattern [w($x|y$)z] execution

Outputs:

- Ocurrences of '-', '0' or '9' = 9
- Ocurrences of 'a+' = 9
- Ocurrences of 'w(x—y)z' = 2

5 Conclusions

The analysis of the obtained results highlights the significance of applying theoretical concepts to effectively solve problems. The implementation demonstrated how automata theory and formal language concepts contribute to lexical analysis, ensuring precise and efficient token recognition. It also underscores the importance of prior knowledge acquired in subjects like Discrete Structures and new concepts introduced in the Compilers course.

Using a DFA-based approach enabled systematic pattern identification and counting within the input text. This process illustrates the relevance of automata-based methods in lexical analysis and demonstrates the efficiency of using tools like Flex for structured and optimized token processing. It also emphasizes the importance of avoiding ambiguity, as ambiguity leads to nondeterministic results. This validates the importance of strong theoretical foundations and highlights their role in developing robust computational solutions.

References

[1] "Lexical Analysis With Flex, for Flex 2.6.4". flex. Accedido el 28 de febrero de 2025. [En línea]. Disponible: https://docs.jade.fyi/gnu/flex.html.

[2] "Lenguajes Formales y Autómatas". Ivan Vladimir Meza Ruiz Blog. Accedido el 28 de febrero de 2025. [En línea]. Disponible: https://turing.iimas.unam.mx/ ivanvladimir/page/curso$_lfya$/.