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
Sanchez Mayen Tristan Qesen	320187072
Sandoval Hernandez Darinka	320137156

Group:
5
Semester:
2025 - 2

Mexico, CDMX. March 13th 2025

Contents

1 Introduction

- **Problem statement:** The first phase of compilation is lexical analysis, where source code is broken down into tokens that represent fundamental language units. This process is crucial for ensuring efficient syntax and semantic analysis. The first goal is to develop a lexical analyzer that processes an input file or code snippet, counts tokens, and correctly classifies them using Python tools for text and pattern analysis.
- **Motivation:** Token classification is a key component in compiler construction, as it facilitates structured code interpretation and debugging. This project not only applies theoretical concepts but also provides a practical tool to automate lexical analysis, reducing human error and improving software development efficiency.
- **Objectives:** To develop a Python-based lexical analyzer that processes input code, identifies and categorizes tokens efficiently and easily, utilizing regular expressions and specialized text-processing tools.

2 Theoretical Framework

The process of compiling a programming language involves an analysis phase and a Synthesis phase, awe are going to enfocate first of all in the **lexical analysis** which is fundamental in the creation of a compiler. Lexical analysis serves as the initial stage in transforming raw source code into a structured format that subsequent phases, such as syntax and semantic analysis, developing the process efficiently.

Lexical Analyzer

A lexical analyzer commonly known as a lexer, is a program responsible for reading the source code and converting it into a sequence of tokens. Tokens are atomic units that represent the syntactic components of a language, such as keywords, identifiers, operators, constants, and punctuation marks. The lexer operates by scanning the source code character by character, grouping these characters into meaningful sequences based on predefined rules and patterns.

The main objectives of a lexical analyzer are:

- **Scanning:** A lexical analyzer scans the raw code line by line in order to produce tokens.
- **Tokenization:** Breaking the input source code into tokens by recognizing specific patterns known as lexemes.
- **Filtering:** Removes irrelevant characters like whitespace and comments.

- **Error Handling:** Identifying and reporting lexical errors, such as identifiers that are wat too long or illegal characters that are out of the code´s source.

By performing these tasks, the lexer simplifies the source code, allowing the parser to focus on syntactic structure without being concerned with low-level details.

The Tokenization Process

The tokenization process involves analyzing the input source code to identify sequences that match predefined patterns. These patterns are usually expressed using regular expressions, which provide the way to describe character sequences. Each regular expression corresponds to a specific token type. In the context of the implemented lexer using the PLY (Python Lex-Yacc) library, tokenization follows a structured approach:

- **Pattern Definition:** Each token type is defined by a regular expression. For instance, the pattern `r'\d+(\.\d+)?'` matches numeric constants, capturing both integers and floating-point numbers, which begin with a letter or underscore and are followed by letters, digits, or underscores.
- **Character Scanning:** The lexer reads the source code sequentially, applying each regular expression to determine if the current sequence matches any token pattern.
- **Token Creation:** Upon identifying a match, the lexer creates a token object that encapsulates the type and value of the matched sequence.
- **Token Emission:** The token is then passed to the parser for further syntactic analysis.
- **Error Handling:** If no pattern matches the current input, the lexer invokes an error-handling routine to report the illegal character and advance the scanning process.

Error handling is a crucial aspect of lexical analysis. In PLY, this is achieved through the `t_error` function, which captures and processes unrecognized characters, preventing unexpected lexer crashes.

This systematic approach ensures that the lexer efficiently segments the source code, creating a clear and structured set of tokens essential for the parsing phase.

Chomsky Hierarchy and Lexical Analysis

An essential theoretical concept in language processing is the Chomsky hierarchy, introduced by Noam Chomsky in 1956. This hierarchy classifies formal grammars into four types, each with varying expressive power and complexity:

1. **Type 0 - Unrestricted Grammars:** These grammars have no restrictions on production rules and can describe any language that a Turing machine can recognize.
2. **Type 1 - Context-Sensitive Grammars:** Productions are of the form $\alpha A\beta \to \alpha\gamma\beta$, where γ is at least as long as A. These grammars are more powerful but computationally complex.
3. **Type 2 - Context-Free Grammars (CFGs):** These grammars have production rules where a single non-terminal symbol is replaced by a string of terminals and/or non-terminals. CFGs are widely used in programming language syntax.
4. **Type 3 - Regular Grammars:** The most restrictive type, regular grammars can be represented using regular expressions. These grammars are ideal for defining the lexical structure of programming languages.

While lexical analysis commonly relies on **regular grammars** (Type 3) due to their simplicity and efficiency, **context-free grammars** (CFGs) are considered more expressive and are better suited for more complex syntactic structures. CFGs enable the representation of nested and hierarchical patterns which are prevalent in programming languages.

PLY constructs the syntax tree through a combination of tokenization and parsing(Using our CFG), the syntax tree is bukt bottom-up as the parser reduces rules into hierarchical structures.

Application in Lexer Design

In the design of the lexer using PLY, the application of the Chomsky hierarchy is evident in the use of regular expressions to define token patterns. Each token type corresponds to a pattern that can be expressed within the constraints of regular grammars. For example:

- **Keywords:** Defined by explicit string patterns, such as `r'int|return'`.
- **Identifiers:** Recognized by the pattern `r'[a-zA-Z_][a-zA-Z_0-9]*'`, which matches typical variable or function names.
- **Operators:** Defined by patterns like `r'==|!=|<=|>=|<|>|=|\+|\-|\*|/'`, covering common arithmetic and comparison operators.
- **Constants and Literals:** Captured using patterns that account for numeric values and string literals, such as `r'\d+(\.\d+)?'`

Using these patterns, the lexer efficiently processes the input code, segmenting it into meaningful tokens. The regular grammar foundation ensures that this process is both reliable and computationally efficient.

3 Body

Grammar

Our proposal of a CFG is the following:

S $\rightarrow i|in|int|i(a-m|o-z|A-Z|\_|0-9)T|in(a-s|u-z|A-Z|\_|0-9)T|int(a-z|A-Z|\_|0-9|)T$

Applying left factoring

$S \rightarrow iS'$

$S' \rightarrow \backslash s|n|nt|(a-m|o-z|A-Z|\_|0-9)T|in(a-s|u-z|A-Z|\_|0-9)T|int(a-z|A-Z|\_|0-9|)T$

Applying left factoring

$S' \rightarrow \backslash s|nS''|(a-m|o-z|A-Z|\_|0-9)T$

$S'' \rightarrow \backslash s|t|(a-s|u-z|A-Z|\_|0-9)T|t(a-z|A-Z|\_|0-9|)T$

We apply Left Factoring

$S'' \rightarrow \backslash s|tS'''|(a-s|u-z|A-Z|\_|0-9)T$

$S'' \rightarrow \backslash s|(a-z|A-Z|\_|0-9|)T$

$T \rightarrow (a-z|A-Z|\_|0-9|)T|\backslash s$

$U \rightarrow (==|!=|<=|>=|<|>|=|+|-|*|/)$

$V \rightarrow ((|)|[|]|||.|,|:|;|!|)$

$W \rightarrow (0-9)|(0-9)W|(0-9).W$

To tokenize the expression int a = 9;, it is essential to define a corresponding context-free grammar (CFG). During this process, potential ambiguities arise in the generation of the symbol S, which could lead to multiple parsing interpretations. To address this issue, left factoring is applied multiple times, restructuring the grammar to ensure a more deterministic parsing process.

As a consequence of this transformation, a right recursion is introduced, helping to maintain a structured and predictable parsing sequence. This modification reduces ambiguity and makes the grammar more suitable for implementation in a lexical or syntactic analyzer. Additionally, the symbol is incorporated to explicitly represent whitespace, ensuring proper handling of spaces, particularly after recognizing the keyword int. These adjustments enhance token recognition accuracy and contribute to a more robust parsing mechanism.

Following the grammar with the following input: int a= 9; we produce the subsequence DFA:

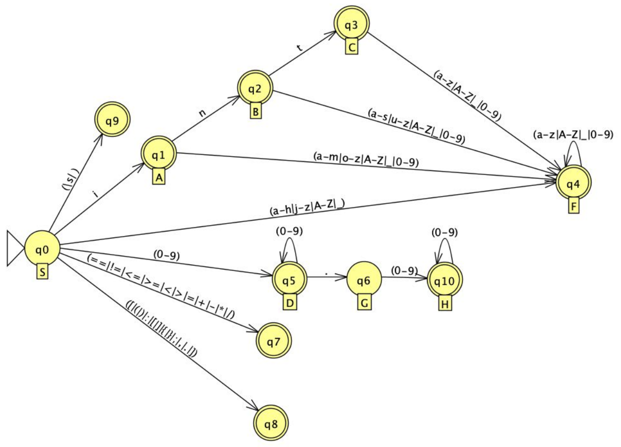


Figure 1: DFA, input int a=9;

We will be using the PLY (Python Lex-Yacc) library.

The next step is to define a tuple in our code containing the names of the tokens that our lexer will recognize. In this case, they are as follows:

- **Keywords**
- **Identifiers**
- **Operator**
- **Constant**
- **Punctuation**
- **Literal**

Once the tokens are defined, we establish the **regular expressions** for each token type.

-In the case of Keywords, whose regular expression is

```
r'\b(const|double|float|int|short|char|long|struct|break|for|if|else
        |switch|case|do|while|default|goto|void|return)\b'
```

we will only consider the words that are in the expression.

-For Identifiers, which will have the regular expression

r’[a-zA-Z_][a-zA-Z_0-9]*’

it will recognize **variable** and **function names**. The rule is that they must start with a letter or an underscore, followed by letters, numbers, or underscores.

-For Operators, we will use the following regular expression.

```
r'==|!=|<=|>=|<|>|=|\+|\-|\*|/'
```

which will identify **comparison**, **assignment**, and **arithmetic operators**.

-In the case of Constants, their regular expression will be

r’\d+(\.\d+)?’

which will identify both **integers** and **decimals**.

-For Punctuation, we will use the regular expression

r’[\(\):\[\]\\;,.]’

, and we will only consider **punctuation marks**.

-Finally, for Literals, we will use the regular expression

r’”([ˆ¨\\]—\\.)*”’

which will essentially identify **text strings** enclosed in quotation marks.

Additionally, we will configure the lexer to ignore spaces, tabs, and line breaks.

If we encounter unknown characters during the process, a warning will be displayed, and the character will be ignored. To handle these cases, we will use the following functions from the PLY library:

- **t.lexpos:** Indicates the position of the error within the source code.
- **t.lexer.skip(1):** Makes the lexer ignore a single character and continue analyzing the code.

After defining these parameters, we create the lexer using the **lex.lex()** command, which processes the **t_*** functions to generate the lexical analyzer.

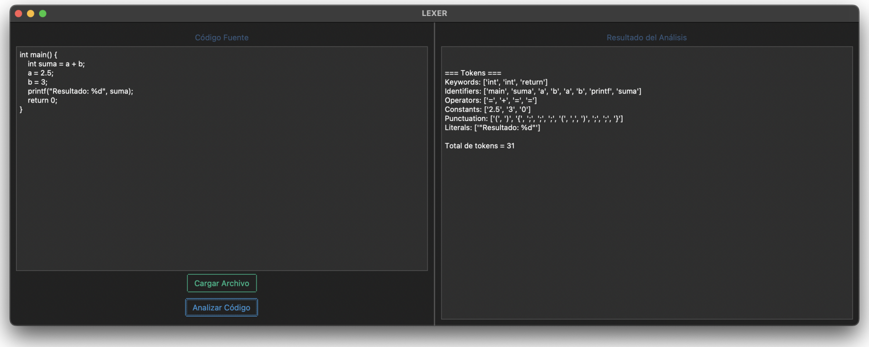
The function responsible for performing the lexical analysis will take a fragment of the source code as a parameter. The code is passed to the lexer using **lexer.input(code)**, and a for loop will iterate through the generated tokens according to their categories while simultaneously counting the total number of tokens.

The output will be a formatted string that displays the detected tokens and their total count.

4 Results

We will be implementing a graphic interface to make it more friendly for the user to use and understand our lexer. This are the results obtained from 3 examples:

- 1. This code was written within the window, only using the main function, describing a basic algorithm for adding two integers and displaying the result on the screen.

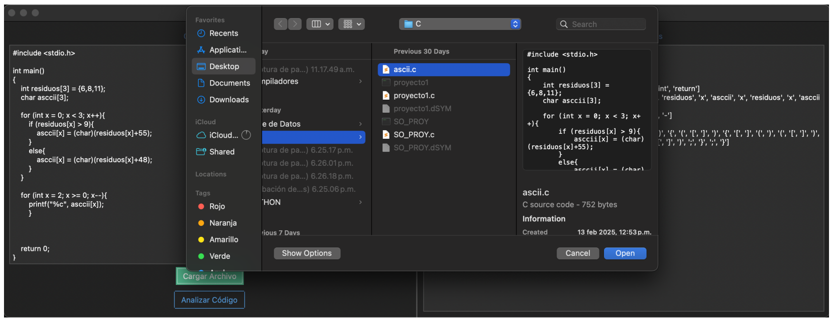


```
int main() {
    int suma = a + b;
    a = 2.5;
    b = 3;
    printf("Resultado: %d", suma);
    return 0;
}
```

```
=== Tokens ===
Keywords: ['int', 'int', 'return']
Identifiers: ['main', 'suma', 'a', 'b', 'a', 'b', 'printf', 'suma']
Operators: ['=', '+', '=', '=']
Constants: ['2.5', '3', '0']
Punctuation: ['(', ')', '{', ';', ';', ';', '(', ',', ')', ';', ';', '}']
Literals: ['"Resultado: %d"']

Total de tokens = 31
```

Figure 2: Example 1

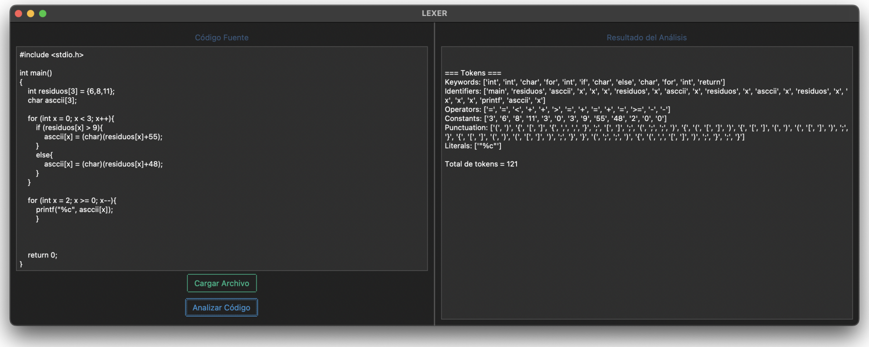
- 2. In this case, the code was loaded from the files for analysis within the app, describing the algorithm to convert decimal numbers to hexadecimal using the ASCII code. In this case, we have a description of the libraries used,unlike the previous exercise, but this will not count as a token, because we will take the headers as comments or saying it in another way, we will ignore them.



```
#include <stdio.h>

int main()
{
    int residuos[3] = {6,8,11};
    char asccii[3];

    for (int x = 0; x < 3; x++){
        if (residuos[x] > 9){
            asccii[x] = (char)(residuos[x]+55);
        }
        else{
            asccii[x] = (char)(residuos[x]+48);
        }
    }

    for (int x = 2; x >= 0; x--){
        printf("%c", asccii[x]);
        }



    return 0;
}
```

ascii.c
C source code - 752 bytes
Information
Created 13 feb 2025, 12:53 p.m.

Figure 3: Loading File Example 2

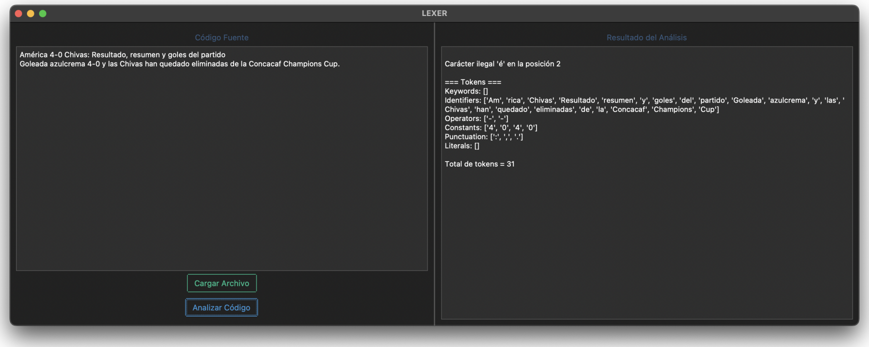


```
#include <stdio.h>

int main()
{
   int residuos[3] = {6,8,11};
   char asccii[3];

   for (int x = 0; x < 3; x++){
      if (residuos[x] > 9){
         asccii[x] = (char)(residuos[x]+55);
      }
      else{
         asccii[x] = (char)(residuos[x]+48);
      }
   }

   for (int x = 2; x >= 0; x--){
      printf("%c", asccii[x]);
      }



   return 0;
}
```

```
=== Tokens ===
Keywords: ['int', 'int', 'char', 'for', 'int', 'if', 'char', 'else', 'char', 'for', 'int', 'return']
Identifiers: ['main', 'residuos', 'asccii', 'x', 'x', 'x', 'residuos', 'x', 'asccii', 'x', 'residuos', 'x', 'asccii', 'x', 'residuos', 'x', '
x', 'x', 'x', 'printf', 'asccii', 'x']
Operators: ['=', '=', '<', '+', '+', '>', '=', '+', '=', '+', '=', '>=', '-', '-']
Constants: ['3', '6', '8', '11', '3', '0', '3', '9', '55', '48', '2', '0', '0']
Punctuation: ['(', ')', '{', '[', ']', '{', ',', ',', '}', ';', '[', ']', ';', '(', ';', ';', ')', '{', '(', '[', ']', ')', '{', '[', ']', '(', ')', '(', '[', ']', ')', ';',
'}', '{', '[', ']', '(', ')', '(', '[', ']', ')', ';', '}', '}', '(', ';', ';', ')', '{', '(', ',', '[', ']', ')', ';', '}', ';', '}']
Literals: ['"%c"']

Total de tokens = 121
```

Figure 4: Example 2

- 3. In this last case, we will only use an arbitrary text as an example since the lexer only considers the tokens we select, BASED on the C language.



```
América 4-0 Chivas: Resultado, resumen y goles del partido
Goleada azulcrema 4-0 y las Chivas han quedado eliminadas de la Concacaf Champions Cup.
```

```
Carácter ilegal 'é' en la posición 2

=== Tokens ===
Keywords: []
Identifiers: ['Am', 'rica', 'Chivas', 'Resultado', 'resumen', 'y', 'goles', 'del', 'partido', 'Goleada', 'azulcrema', 'y', 'las', '
Chivas', 'han', 'quedado', 'eliminadas', 'de', 'la', 'Concacaf', 'Champions', 'Cup']
Operators: ['-', '-']
Constants: ['4', '0', '4', '0']
Punctuation: [':', ',', '.']
Literals: []

Total de tokens = 31
```

Figure 5: Example 3

5 Conclusions

As we can see in the previous report, the results obtained in this project validate the practical application of lexical analysis in programming language processing. As was proven in the examples, the lexical analyzer successfully identified and categorized tokens; additionally, the integration of regular expression proved essential to accurately detect patterns in the source code, confirming the theoretical importance of formal language principles in token recognition.

In addition, the results highlight key challenges in the development of lexical analysis, such as handling lexical ambiguities and ensuring that token classification remains consistent across different input formats. The distinction between reserved words and identifiers, as well as the correct parsing through our GFC of numeric values and special symbols, was effectively managed, reinforcing the importance of structured lexical processing. These findings are a crucial foundation for subsequent phases for the creation of our compiler.

6 References

D. Clapham, "PLY (Python Lex-Yacc)," *PLY: Lex and Yacc for Python*, [On line]. Available: `https://ply.readthedocs.io/en/latest/ply.html`. [Accesed: Mar. 12, 2025].