National Autonomous University of Mexico
Faculty of Engineering

Compilers

Lexer Report

Students:
320280324 Balam Flores Gabriel Patricio
320338274 Solé Pi Arnau Roger
320182668 Hernández Domínguez Luis Carlos
320516467 Cervantes Mateos Leonardo

Group: #05

Semester: 2025-2

https://github.com/BalmyM4/unam.fi.compilers.g5.05.git

Index

Introduction

The compilation of a program is a process that translates source code so the computer can execute it correctly. The first stage of this process is called lexical analysis, which is responsible for breaking down the source code into tokens. This task is performed by a program called a lexer — a component of the compiler that reads the source code and produces a sequence of tokens.

For a program to execute correctly, it is essential to have a reliable lexer that performs its task accurately. In this document, we present a lexer developed using tools based on PLY. Through the development of this lexer, we aim to gain a better understanding of its functionality and importance within the compilation process. As well as achieve a fully functional lexer.

Theoretical Framework

To perform lexical analysis, the lexer uses regular expressions (regex). According to [1], "Regular expressions are a notation used to describe all the languages that can be built from applying the language operators union, concatenation, and closure to the symbols of some alphabet." The use of regular expressions instead of grammars makes the lexer more efficient.

The result of this process is a sequence of tokens. A token is defined as "a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier" [1]. Tokens are typically classified into the following categories:

- Identifier: variable names, functions, etc.
- Operator: +, -, *, etc.
- Keyword: return, int, etc.
- Constant: 1, 3, 5, etc.
- Literals: Anything surrounded by “ ”.
- Punctuators: {, }, (,), etc.

This classification is crucial for the next phase of the compilation process: syntax analysis. During syntax analysis, token names are often referred to as terminals, as they appear as terminal symbols in the grammar of the programming language.

Context-free grammars (CFG) are the primary type of grammar used to define the syntax of programming languages. They systematically describe language constructs such as expressions and statements.

A key part of this analysis is parsing, defined as "the process of determining how a string of terminals can be generated by a grammar" [1]. For parsing to succeed, the CFG mustn't be ambiguous — meaning the grammar must not produce more than one possible parse tree for the same string. Also, the grammar cannot have left recursion, a situation where the leftmost symbol of the body is the same as the non-terminal at the head of the production. which can cause the parser to enter an infinite loop.

In order to prepare the grammar for a top-down parsing it's necessary to apply left factoring. "When the choice between two alternative A productions is not clear, it's possible to rewrite the productions to defer the decisions until enough of the input has been seen to make the right decision." [1]

$$A \rightarrow \alpha\beta_1|\alpha\beta_2$$

Deferring the decision by expanding A to A' allows us to see the input derived from α, and expanding A' to β1 or β2. The original production becomes.

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1|\ \beta_2$$

Development

As mentioned previously the development of this lexer

Functions from and defined for PLY

- **lex()**

  This function is responsible for building the lexer taking into account the token rules and functions defined previously. Token rules are basically the regular expressions (regex) used in order for the lexer to recognize and identify the tokens and if they're a valid token type.

- **lex.input()**

  Feeds the lexer the input for tokenization.

- **lex.token()**

  Extracts the next token from the lexer.

- **t_error()**

  Handles the errors, if a token doesn't match with any of the rules defined then it's ignored and prints an error message.

- **t_identifier()**

  Determine whatever or not the token is a keyword or an identifier by comparing it with the keywords defined in the dictionary.

- **t_literal()**

  Identifies literals through a regex.

Functions from UI

The library used for the user interface was Tkinter, a simple and suitable option for small applications.

Functions

- **open_file()**

  Linked to the button of the same name. Allows searching for files in the operating system, opening them in the program, extracting their information, and displaying it in the first text box.

- **get_text()**

  Retrieves the content of the first text box and formats it by removing whitespace.

- **analyze_text()**

  Linked to the Analyze button. Connects the user interface with the lexer by calling lex.Lexer_analyzer() and displays the result in the second text box.

- **save_file()**

  Linked to the button of the same name. Saves the content of the second text box to a file in the operating system.

- **clear_text_area()** and **clear_result_area()**

  Linked to the buttons of their respective text areas. Clears their content.

Tkinter Elements

- **tk.Tk()**

  Creates the main window.

- **tk.title()**

  Assigns a title to the window.

- **tk.geometry()**

  Sets the window size.

- **tk.config()**

  Modifies the window style.

- **move_window()**

  Allows moving the window.

- **tk.overrideredirect()**

  Removes the default top bar.

- **tk.Frame()**

  Creates frames where other interactive objects can be inserted.

- **tk.pack()**

  Defines object positioning parameters.

- **tk.Label()**

  Creates a label to display text.

- **tk.PhotoImage()**

  Allows loading images into the program.

- **tk.Button()**

  Creates interactive buttons.

- **bar.bind()**

Allows a frame to be draggable.

- **scrolledtext.ScrolledText()**

  Generates a text area with a scroll bar.

- **button.grid_columnconfigure()**

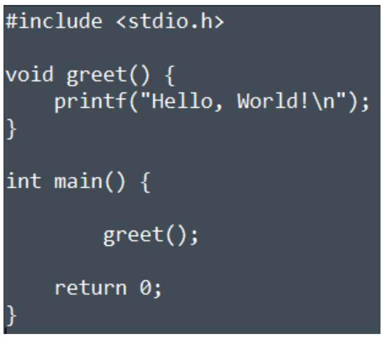
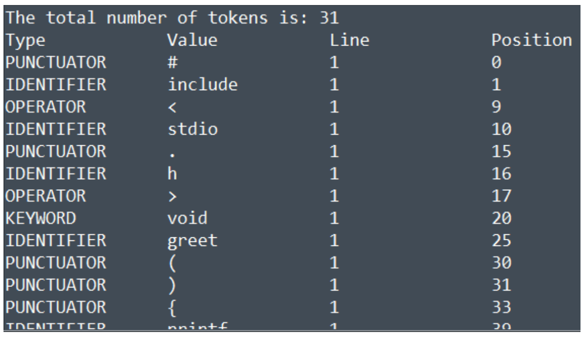
  Configures the positioning and size of buttons.

- **tk.mainloop()**

  Runs the main loop of the graphical interface, enabling the display of all created elements.

Testing

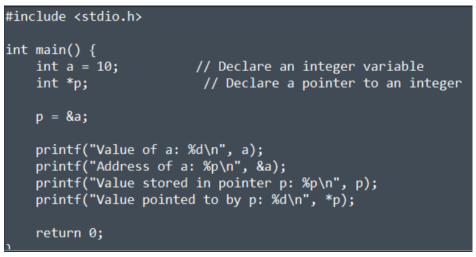
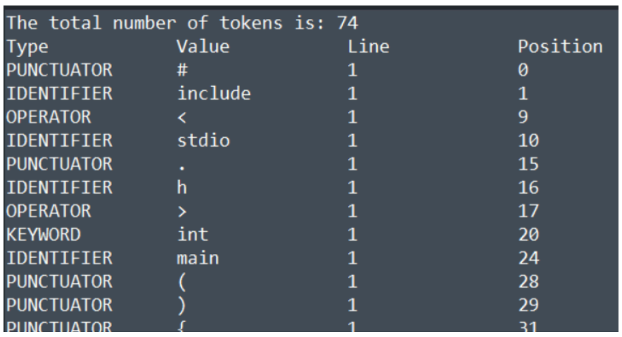
- Trial #1

INPUT	OUTPUT
	

```
#include <stdio.h>

void greet() {
    printf("Hello, World!\n");
}

int main() {

        greet();

    return 0;
}
```

```
The total number of tokens is: 31
Type            Value           Line            Position
PUNCTUATOR      #               1               0
IDENTIFIER      include         1               1
OPERATOR        <               1               9
IDENTIFIER      stdio           1               10
PUNCTUATOR      .               1               15
IDENTIFIER      h               1               16
OPERATOR        >               1               17
KEYWORD         void            1               20
IDENTIFIER      greet           1               25
PUNCTUATOR      (               1               30
PUNCTUATOR      )               1               31
PUNCTUATOR      {               1               33
IDENTIFIER      printf          1               39
```

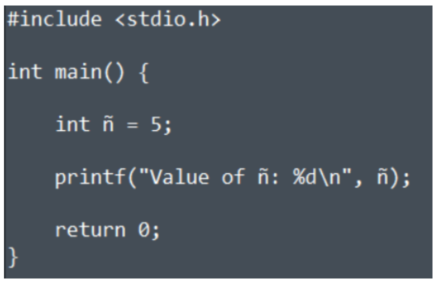
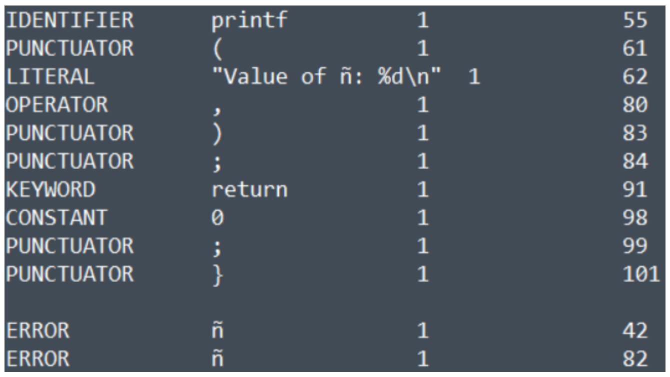
- Trial #2

INPUT	OUTPUT
	

```
#include <stdio.h>

int main() {
    int a = 10;           // Declare an integer variable
    int *p;                // Declare a pointer to an integer

    p = &a;

    printf("Value of a: %d\n", a);
    printf("Address of a: %p\n", &a);
    printf("Value stored in pointer p: %p\n", p);
    printf("Value pointed to by p: %d\n", *p);

    return 0;
}
```

```
The total number of tokens is: 74
Type            Value           Line            Position
PUNCTUATOR      #               1               0
IDENTIFIER      include         1               1
OPERATOR        <               1               9
IDENTIFIER      stdio           1               10
PUNCTUATOR      .               1               15
IDENTIFIER      h               1               16
OPERATOR        >               1               17
KEYWORD         int             1               20
IDENTIFIER      main            1               24
PUNCTUATOR      (               1               28
PUNCTUATOR      )               1               29
PUNCTUATOR      {               1               31
```

- Trial #3

INPUT	OUTPUT
	

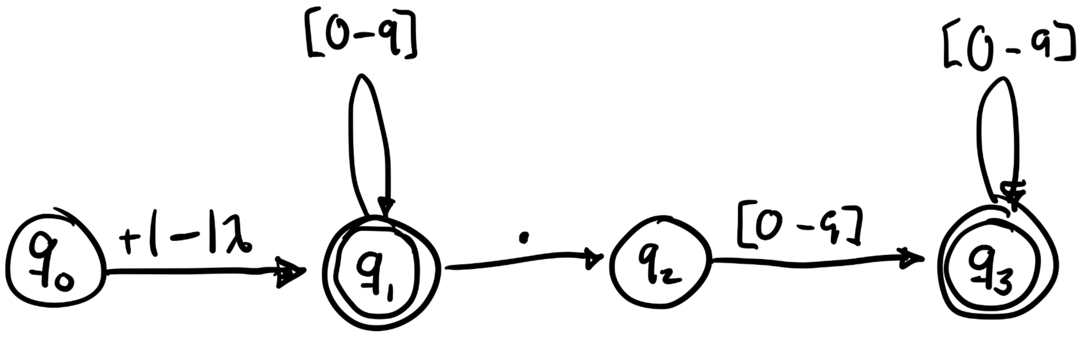
```
#include <stdio.h>

int main() {

    int ñ = 5;

    printf("Value of ñ: %d\n", ñ);

    return 0;
}
```

```
IDENTIFIER      printf          1               55
PUNCTUATOR      (               1               61
LITERAL         "Value of ñ: %d\n"  1           62
OPERATOR        ,               1               80
PUNCTUATOR      )               1               83
PUNCTUATOR      ;               1               84
KEYWORD         return          1               91
CONSTANT        0               1               98
PUNCTUATOR      ;               1               99
PUNCTUATOR      }               1               101

ERROR           ñ               1               42
ERROR           ñ               1               82
```

Token's Automatas

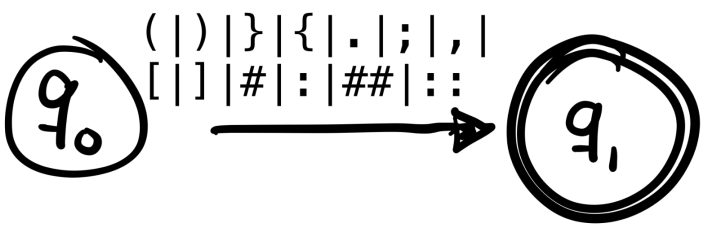
Automata design for each kind of token

Constants automata



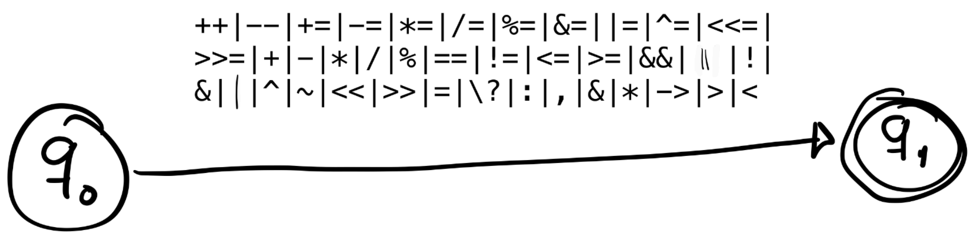
$$\begin{tikzcd} q_0 \arrow[r, "+|-|\lambda"] & \boxed{q_1} \arrow[loop above, "{[0-9]}"] \arrow[r, "."] & q_2 \arrow[r, "{[0-9]}"] & \boxed{q_3} \arrow[loop above, "{[0-9]}"] \end{tikzcd}$$

Punctuation automata

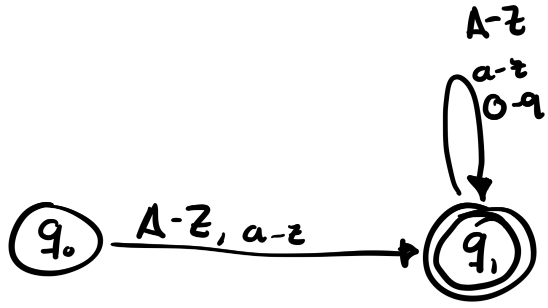


$$\begin{tikzcd} q_0 \arrow[r, "{(|)|\}|\{|.|;|,|[|]|\#|:|\#\#|::}"] & \boxed{q_1} \end{tikzcd}$$

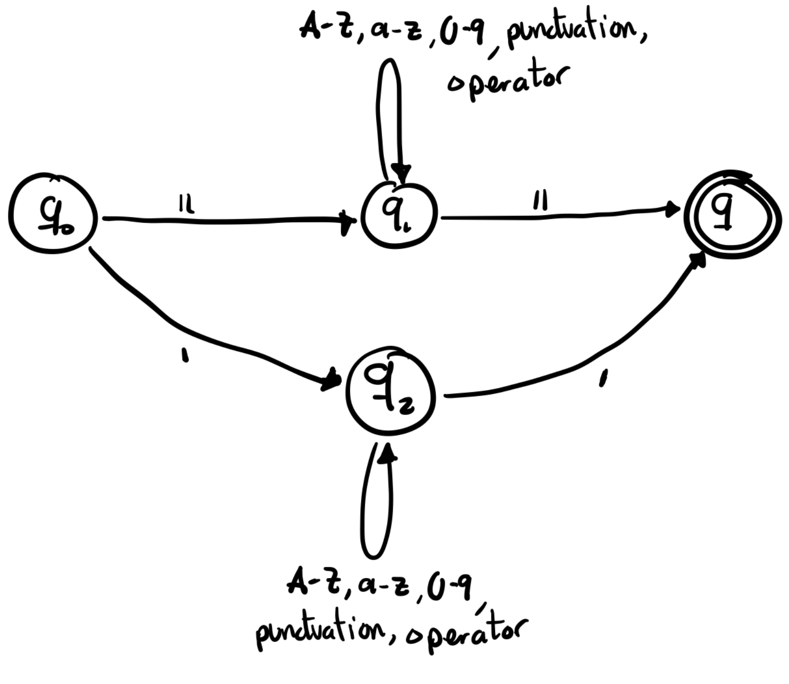
Operators automata



Identifiers automata



Identifiers automata



A-Z, a-z, 0-9, punctuation, operator
"
"
'
'
q0
q1
q
q2
A-Z, a-z, 0-9, punctuation, operator

Grammar

<u>Left factoring and left recursive case example:</u>

When designing the grammar for the compiler, we need to be careful when encountering left factoring or left recursion. One common scenario where this can happen is with the if-else structure, which is widely used in the C programming language. In this example, we will demonstrate how to handle both left factoring and left recursion in such cases.

A → A = E | if (E) A else A | if (E) A| E

Eliminate left recursion:

A → if (E) A else A A’ | if (E) A A’| E
A’ → =EA’|ε

Apply left factoring

A → if (E) A A’’|E
A’ → =EA’|ε
A’’→ else A A’ | A’

<u>Example of grammar with one of the tests</u>

Non terminal symbols : A , Token, KEYWORD, IDENTIFIER, PUNCTUATOR, CONSTANT, OPERATOR, LITERAL.

A → Token A |ε

Token → KEYWORD|IDENTIFIER| PUNCTUATOR|CONSTANT|OPERATOR| LITERAL

KEYWORD→ int | void | return
IDENTIFIER→ include | stdio | h | greet
PUNCTUATOR→ # | (|) | { | } | ;
CONSTANT→ 0
OPERATOR→ < | >
LITERAL→ String

Example:

```
#include <stdio.h>

void greet() {
   printf("Hello, World!\n");
}

int main() {

   greet();

   return 0;
}
```

A → Token A
Token A → Token Token A
Token Token A → Token Token Token A
Token Token Token A → Token Token Token Token A
.
.
.

Token Token . . . Token Token A → Token Token . . . Token Token ε - (Token 31 times)

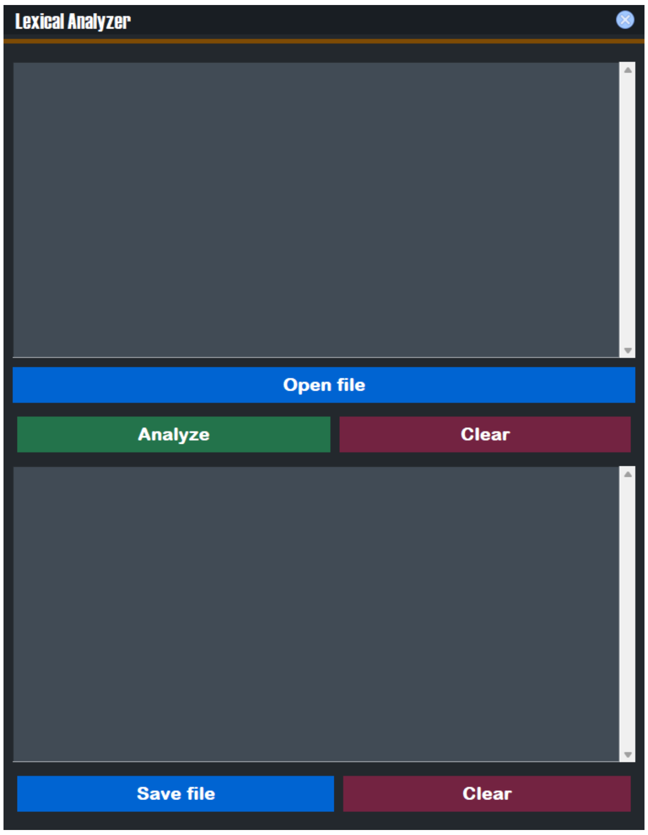
Token Token . . . Token Token ε → PUNCTUATOR IDENTIFIER OPERATOR IDENTIFIER PUNCTUATOR IDENTIFIER OPERATOR KEYWORD IDENTIFIER PUNCTUATOR PUNCTUATOR PUNCTUATOR IDENTIFIER PUNCTUATOR LITERAL PUNCTUATOR PUNCTUATOR PUNCTUATOR KEYWORD IDENTIFIER PUNCTUATOR PUNCTUATOR PUNCTUATOR IDENTIFIER PUNCTUATOR PUNCTUATOR PUNCTUATOR KEYWORD CONSTANT PUNCTUATOR PUNCTUATOR

→ # include < stdio . h > void greet ( ) { printf ("Hello, World!\n") ; } int main ( ) { greet ( ) ; return 0 ; }

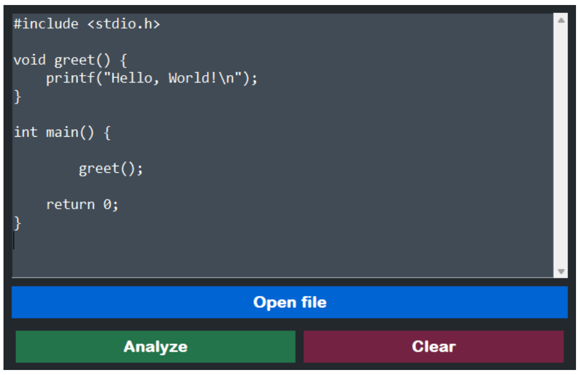
Results

Below are the results of the lexer implementation. The images illustrate the graphical user interface, the input file used for tokenization, and the corresponding output after processing.

User interface corresponding to the lexer.

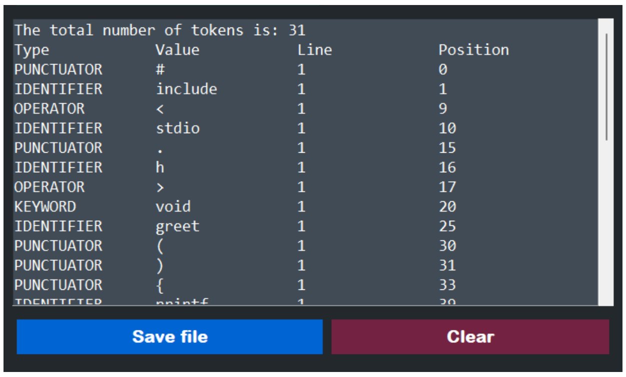


Visualization of the file used as input.



```
#include <stdio.h>

void greet() {
    printf("Hello, World!\n");
}

int main() {

        greet();

    return 0;
}
```

Output generated after tokenizing the input.



```
The total number of tokens is: 31
Type            Value           Line            Position
PUNCTUATOR      #               1               0
IDENTIFIER      include         1               1
OPERATOR        <               1               9
IDENTIFIER      stdio           1               10
PUNCTUATOR      .               1               15
IDENTIFIER      h               1               16
OPERATOR        >               1               17
KEYWORD         void            1               20
IDENTIFIER      greet           1               25
PUNCTUATOR      (               1               30
PUNCTUATOR      )               1               31
PUNCTUATOR      {               1               33
```

Conclusion

Developing a lexer required the use of multiple theoretical concepts such as regular expressions or context free grammars. It also implied comprehending what was the purpose behind the lexical analysis, which is basically to tokenize the source code to prepare it for the parsing process. It’s important as well for the grammar that defines the language to not present any of the next problems: left recursion and ambiguity.

In the end, the lexer is only the first step towards the compiler, but it’s important for the future development of the parser and eventual development of the compiler that the lexer, as well as all the theoretical bases are well thought out to prevent any major problems in the future.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston, MA: Pearson, 2007.