University of Kasdi Merbeh Ouargla

Faculty of New Technologies of Information and Communication.

Department of Computer Science and Information Technologies

Artificial Intelligence, Engineering Cycle

Project Report: Lexical Analysis

Presented by:

Nidhal KHAZENE **as Developer**
Khaled RAHMA **as Developer**

As part of the Compiler Design Module.

Index

1) Introduction

Lexical analysis is a fundamental process in compiler and interpreter design that transforms raw source code into a structured sequence of tokens, serving as the critical first phase of language processing. In this intricate transformation, the lexical analyzer, often called a scanner or tokenizer, meticulously breaks down the input text character by character, identifying and categorizing fundamental language elements such as keywords, identifiers, literals, operators, and punctuation symbols while simultaneously stripping away non-essential components like whitespace and comments. Your lexical analysis project will involve creating a sophisticated system that reads source code, recognizes different token types through carefully defined rules and patterns, and generates a token stream that represents the lexical structure of the input, providing a clean, organized representation that subsequent compilation stages like parsing and semantic analysis can effectively process. The core of your project will require developing robust token definitions, implementing a flexible lexer with advanced recognition strategies, constructing comprehensive error handling mechanisms to detect and report lexical inconsistencies, and designing a system that can accurately convert complex source code inputs into a meaningful sequence of tokens that capture the essential syntactic elements of the programming language being analyzed.

2) Why we choose Python

Here are 6 key advantages of choosing Python for a lexical analysis program:

1. Powerful Regular Expression Support: Python's `re` module provides extremely robust and flexible pattern matching capabilities, allowing developers to create sophisticated token recognition rules with minimal, readable code. This makes defining complex lexical patterns significantly easier compared to other programming languages, enabling precise and efficient tokenization of source code.

2. Rich String Manipulation Features: Python offers comprehensive built-in string manipulation methods and libraries that simplify character-level processing and text analysis. These features enable quick and efficient breaking down of input text, handling different character types, and performing advanced string transformations essential for lexical analysis.

3. Object-Oriented Design Flexibility: The language's strong object-oriented programming capabilities allow for clean, modular design of lexical analyzers. Developers can easily create hierarchical token classes, implement inheritance, define custom token types, and build sophisticated lexer architectures with minimal complexity and maximum code reusability.

4. Rapid Prototyping and Development: Python's interpreted nature and dynamic typing enable extremely fast development cycles. Programmers can quickly prototype, test, and modify lexical analysis algorithms without lengthy

compilation processes, making it ideal for experimental and educational lexical analysis projects.

5. Extensive Libraries and Frameworks: Python boasts specialized libraries like `ply` (Python Lex-Yacc) and `sly` (Sly Lex Yacc) that provide ready-made tools for lexical and syntactic analysis. These libraries offer pre-built components and utilities that can significantly accelerate the development of complex lexical analyzers.

6. High Readability and Maintainability: Python's clean, intuitive syntax allows lexical analysis code to be written in a way that closely resembles natural language descriptions of parsing rules. This characteristic makes the code easier to understand, debug, and maintain, which is crucial for complex language processing projects and collaborative development environments.

3) Language Defined

1) Identifiers: $L(\alpha)^\wedge < 40$

$L \in \{A..Z\}$, $\alpha \in \{ L, N, - \}$

α: not ending with a hyphen, cannot contain an even number of consecutive hyphens.

2) Integer Constant: $\pm B(\beta)^\wedge < 7$, $\beta \in \{0..9\}$, its value < 1657634

3) Real Constant: $\pm \lambda(\gamma)^\wedge < 9$, $\{\lambda \in \{0..9\}$ & $\gamma \in \{.,0..9\}\}$

Does not accept numbers in the form -.66 or 76.

4) Comments: Comments are written as: $ characters $ while characters ≠ $

5) Keywords: Keywords are written in uppercase:

List = {PROGRAM, BEGIN, END, INT, REAL, BOOLEAN, CHAR, STRING, CONST, WHEN, DO, OTHERWISE, EXECUTE, FOR}

4)Week 01: User Interface

4.1 Desktop Application with Tkinter

```
root = tk.Tk()
root.title("Compiler Design TP")
menuBar = tk.Menu(root)
root.config(menu=menuBar)
file_menu = tk.Menu(menuBar, tearoff=0)
menuBar.add_cascade(label="File", menu=file_menu)
file_menu.add_command(label="Open", command=open_file)
file_menu.add_command(label="Save", command=save_file)
file_menu.add_command(label="Exit", command=root.quit)
analysis_menu = tk.Menu(menuBar, tearoff=0)
menuBar.add_cascade(label="Analysis", menu=analysis_menu)
analysis_menu.add_command(label="Lexical Analysis", command=lexical_analysis)
analysis_menu.add_command(label="Syntax Analysis", command=lambda:
print("Syntax Analysis"))
analysis_menu.add_command(label="Semantic Analysis", command=lambda:
print("Semantic Analysis"))
main_frame = tk.Frame(root)
main_frame.pack(fill=tk.BOTH, expand=True)
left_frame = tk.Frame(main_frame)
left_frame.pack(side=tk.LEFT, fill=tk.Y)
right_frame = tk.Frame(main_frame)
right_frame.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True)
text_box1 = tk.Text(right_frame, width=50, height=25)
text_box1.pack(side=tk.LEFT, padx=40)
text_box2 = tk.Text(right_frame, width=50, height=25)
text_box2.pack(side=tk.RIGHT, padx=40)
root.mainloop()
```

4.2 Open File:

```
def open_file():
   file_path = filedialog.askopenfilename(filetypes=[("Text files", ".txt"),
("All files", ".*")])
   if file_path:
       with open(file_path, "r") as file:
           content = file.read()
           text_box1.delete(1.0, tk.END)
           text_box1.insert(tk.END, content)
```

4.3 Save File:

```
def save_file():
   file_path = filedialog.asksaveasfilename(defaultextension=".txt",
                                            filetypes=[("Text files", ".txt"),
("All files", ".*")])
   if file_path:
       with open(file_path, "w") as file:
           content = text_box1.get(1.0, tk.END)
           file.write(content)
```

This Python code is a basic Tkinter desktop application designed for text file handling and some compiler-related analysis. Here's a summary:

1. File Handling:

   The application includes Open and Save functionalities:

   Open: Allows the user to open a text file, display its content in text_box1.

   Save: Saves the content of text_box1 to a specified file.

   These operations use filedialog to handle file selection.

2. Menu Bar:

   A File menu with:

   Open, Save, and Exit options.

   An Analysis menu with:

   Options for Lexical Analysis, Syntax Analysis, and Semantic Analysis.

3. GUI Layout:

   The main window (root) contains:

   A menu bar at the top for navigation.

   A main frame divided into:

   left_frame (currently unused).

right_frame, which holds two text boxes (text_box1 and text_box2), side-by-side for displaying or editing text.

5)Week 02: Remove the superfluous characters.

```
def clean_paragraph(input_string):
   input_string = re.sub(r'\$[^$]*\$(?![^\s])', ' ', input_string)
   input_string = re.sub(r'\$[^$]*\$', ' ', input_string)
   input_string = re.sub(r'[\t\n]', ' ', input_string)
   return ' '.join(input_string.split())
```

The clean_paragraph function processes a string to clean specific patterns and ensure consistent formatting:

1. Remove Comments: Deletes content enclosed in $...$.
2. Replace Tabs/Newlines: Converts tabs (\t) and newlines (\n) into space.
3. Remove Extra Spaces: Collapses multiple spaces into single space.

6) Week 03: Separate the words with the finite word character (Hashtag).

```
def separate_tokens(text):
   result = []
   words = text.split()
```

6.1 Sign Case

```
if re.match(r'^[-,+]?\d*\.?\d*$', word):
   result.append(word)
   continue
```

6.2 Variable Case

```
if re.match(r'^[a-zA-Z]+\d+$', word):
   result.append(word)
   continue
```

6.3 Loop Case

```
if re.match(r'^[a-zA-Z]\+\+$', word):
   result.append(word[0])
   result.append('++')
   continue
```

6.4 Special Cases

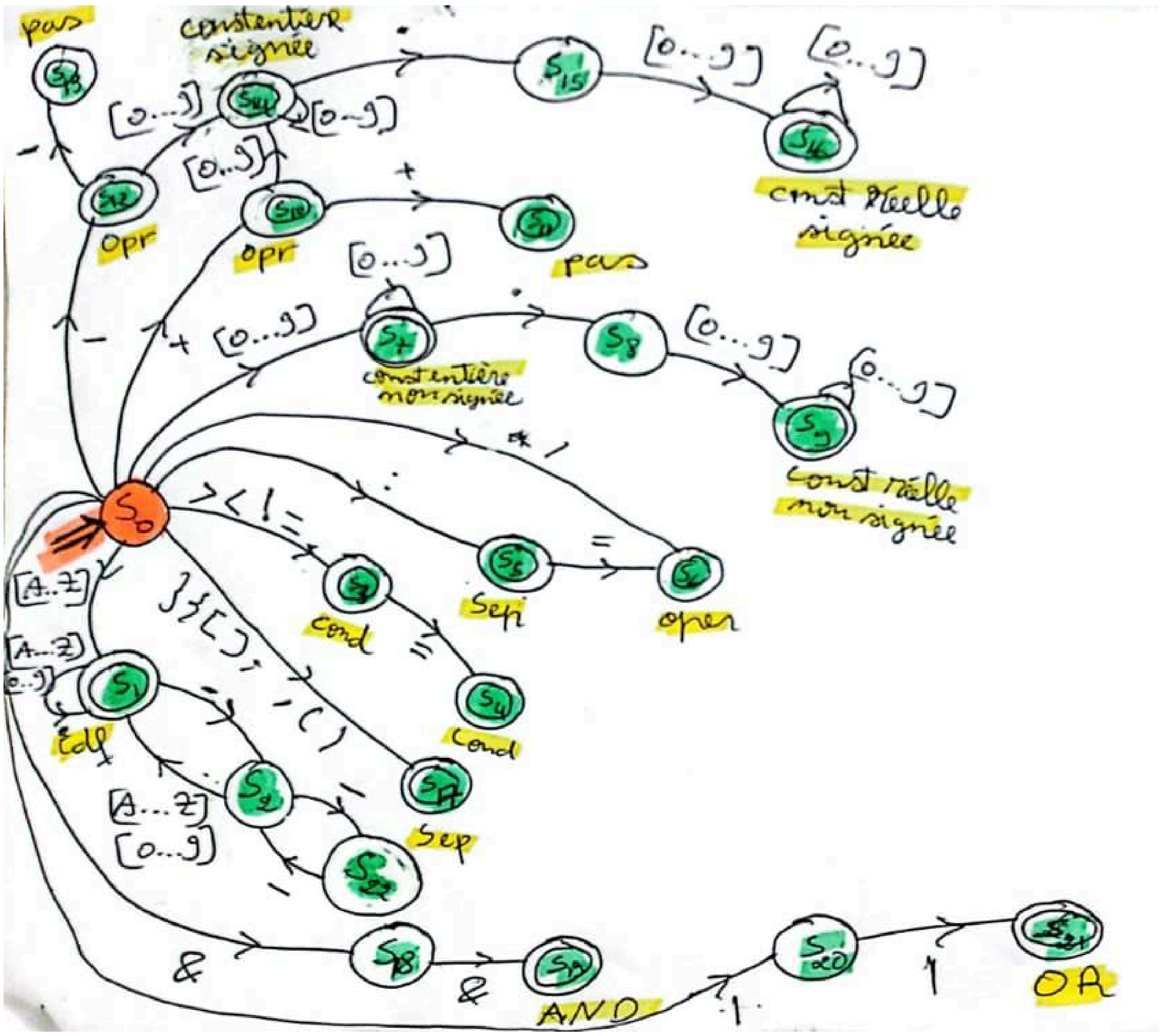
```
current_token = ''
   i = 0
   while i < len(word):
       char = word[i]
       # := Case
       if char == ':' and i + 1 < len(word) and word[i + 1] == '=':
           if current_token:
               result.append(current_token)
               current_token = ''
           result.append(':=')
           i += 2
           continue

       if (char in '+-') and i + 1 < len(word) and word[i + 1].isdigit():
           if i == 0 or not word[i - 1].isalnum():
               num = char
               i += 1
               # example +3.
               while i < len(word) and (word[i].isdigit() or word[i] == '.'):
                   num += word[i]
                   i += 1
               result.append(num)
               current_token = ''
               continue
       # example i++
       if char == '+' and i + 1 < len(word) and word[i + 1] == '+':
           if current_token:
               result.append(current_token)
               current_token = ''
           if i > 0 and word[i - 1].isalpha():
               result.append('++')
           else:
               result.append('+')
               result.append('+')
           i += 2
           continue
       if char in '=+-*/();':
           if current_token:
               result.append(current_token)
               current_token = ''
           result.append(char)
       else:
           current_token += char
       i += 1
   if current_token:
       result.append(current_token)
return '#'.join(result)
```

The separate_tokens function takes a string (text) and separates it into individual tokens based on various patterns, applying custom handling for special cases. Here's a breakdown:

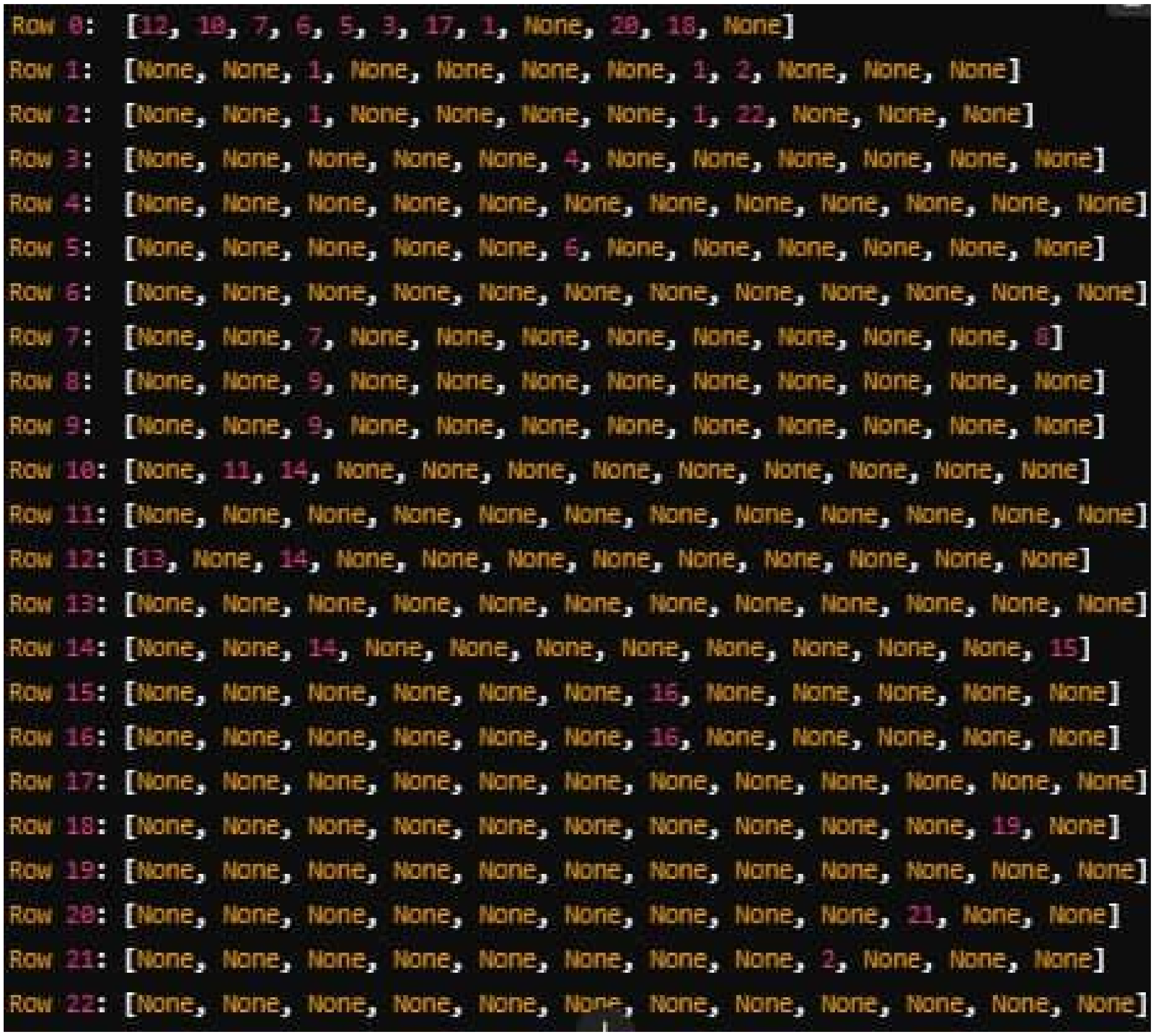
1. Word-by-Word Processing:
   Splits the input text into words using split() and processes each word based on specific rules.
2. Pattern-Based Token Handling:
   Period Ending: Words ending in a period (e.g., `word.`) are added directly.
   SignCase: Numeric tokens with optional signs and decimals (e.g., `+3.14`, `-5`) are added.
   VariableCase: Alphanumeric tokens ending with digits (e.g., `var1`) are added.
   LoopCase: Increments like `i++` are split into separate tokens (`i` and `++`).
3. Special Cases:
   Handles compound tokens like `:=`, and numerical expressions with `+` or `-` as leading characters.
   Separates symbols (=, `+`, `-`, `*`, `/`, `;`, `(`, `)`) from other content.
4. Tokenization Logic:
   Uses a loop to iterate through each character in a word:
   - Accumulates regular characters into a `current_token`.

   Splits tokens when encountering special symbols or specific patterns (e.g., `:=`, `i++`).
5. Final Output:
   Joins all tokens with a # separator and returns the resulting string.

7) **Week 4:** The corresponding Automate.



7) **Week 5:** The transition Matrix



```
Row 0:  [12, 10, 7, 6, 5, 3, 17, 1, None, 20, 18, None]
Row 1:  [None, None, 1, None, None, None, None, 1, 2, None, None, None]
Row 2:  [None, None, 1, None, None, None, None, 1, 22, None, None, None]
Row 3:  [None, None, None, None, None, 4, None, None, None, None, None, None]
Row 4:  [None, None, None, None, None, None, None, None, None, None, None, None]
Row 5:  [None, None, None, None, None, 6, None, None, None, None, None, None]
Row 6:  [None, None, None, None, None, None, None, None, None, None, None, None]
Row 7:  [None, None, 7, None, None, None, None, None, None, None, None, 8]
Row 8:  [None, None, 9, None, None, None, None, None, None, None, None, None]
Row 9:  [None, None, 9, None, None, None, None, None, None, None, None, None]
Row 10: [None, 11, 14, None, None, None, None, None, None, None, None, None]
Row 11: [None, None, None, None, None, None, None, None, None, None, None, None]
Row 12: [13, None, 14, None, None, None, None, None, None, None, None, None]
Row 13: [None, None, None, None, None, None, None, None, None, None, None, None]
Row 14: [None, None, 14, None, None, None, None, None, None, None, None, 15]
Row 15: [None, None, None, None, None, None, 16, None, None, None, None, None]
Row 16: [None, None, None, None, None, None, 16, None, None, None, None, None]
Row 17: [None, None, None, None, None, None, None, None, None, None, None, None]
Row 18: [None, None, None, None, None, None, None, None, None, None, 19, None]
Row 19: [None, None, None, None, None, None, None, None, None, None, None, None]
Row 20: [None, None, None, None, None, None, None, None, None, 21, None, None]
Row 21: [None, None, None, None, None, None, None, None, 2, None, None, None]
Row 22: [None, None, None, None, None, None, None, None, None, None, None, None]
```

9) **Week 6:** The analysis lexical Program

```
#LexicalAlgorithm
def lexicalAlgorithm(str):
   # str = input("enter: ")
   Ec = 0
   tc = str[0]
   cpt = 1
   word = tc
   while Ec is not None and tc != '#':
       Ec = getEcMatrix(Ec, getTcIndex(tc))
       tc = str[cpt]
       cpt += 1
       word = word + tc

   if Ec == 0:
       print(f"{word} incorrect")
   elif Ec == 1:
       if keyword(word):
           print(f"{word} is a keyword")
       elif cpt > 40:
           print("the identifier is > 40")
       else:
           print(f"{word} is a {typeFunction(1)}")
   elif Ec == 3 or Ec == 4:
       print(f"{word} is {typeFunction(3)}")
   elif Ec == 5 or Ec == 17:
       print(f"{word} is {typeFunction(5)}")
   elif Ec == 6 or Ec == 10 or Ec == 12:
       print(f"{word} is {typeFunction(6)}")
   elif Ec == 7:
       if cpt > 7:
           print("the number is > 7.")
       else:
           print(f"{word} is {typeFunction(7)}")
   elif Ec == 9:
       if cpt > 7:
           print("the number is > 7.")
       else:
           print(f"{word} is {typeFunction(9)}")
   elif Ec == 11 or Ec == 13:
       print(f"{word} is {typeFunction(11)}")
   elif Ec == 14:
       if cpt > 7:
           print("the number is > 7.")
       else:
           print(f"{word} is {typeFunction(14)}")
   elif Ec == 16:
       if cpt > 7:
           print("the number is > 7.")
       else:
           print(f"{word} is {typeFunction(16)}")
   elif Ec == 19:
       print(f"{word} is {typeFunction(19)}")
   elif Ec == 21:
       print(f"{word} is {typeFunction(21)}")
   else : print(f"{word} is incorrect")
```

The lexicalAlgorithm function simulates a **lexical analyzer**, classifying a string (str) based on its structure and transitions in a state machine. Here's a short explanation:

1. Initialization:

   Starts at an initial state (Ec = 0) and processes the first character (tc).

Builds the token word by iterating through str until a # (termination symbol) is encountered or no valid state (Ec) exists.

2. State Transitions:

   Uses getEcMatrix to transition between states (Ec) based on the current character (tc) and its type (getTcIndex).

   Tracks the length of the token (cpt) and appends characters to word.

3. Classification:

   After reading the token, determines its type based on the final state (Ec) and additional conditions:

   Keywords: Checked with the keyword function.

   Identifiers: Must be valid and less than or equal to 40 characters.

   Numbers: Must be valid and less than or equal to 7 digits.

   Other tokens are classified using typeFunction(Ec) for specific states.

4. Output:

   Prints the classification of the token (word) or flags it as **incorrect** if the state machine ends in an invalid state.

10) Conclusion

The lexical analysis project successfully demonstrates a sophisticated approach to transforming source code into a structured token sequence using advanced computational techniques. By implementing a robust lexical analyzer in Python, the project effectively breaks down complex textual inputs into fundamental syntactic elements, recognizing and categorizing diverse token types with precision. The implementation provides a flexible framework for parsing programming languages, showcasing the critical first step in compiler design by converting human-readable code into a systematically processable format. This project not only achieves core lexical analysis objectives but also establishes a foundation for more advanced language processing and interpretation.