



University of Asia Pacific

Course Title: Compiler Design Lab

Course Code: CSE 430

Project Title: Lexical Analyzer

Submitted by

Md. Samiur Rahman (20101094)

Md. Atik Shams (20101113)

Ziaul Karim Asfi (20101115)

Submitted to

Fabliha Haque

Lecturer

Department of Computer Science &
Engineering

University of Asia Pacific

Introduction:

We have made a lexical analyzer which recognizes 'Identifier', 'Keyword', 'Operator', 'Parenthesis', 'Punctuation', 'Constant' and 'Header' tokens accurately. Previously, there were issues with recognizing 'Identifier' and 'Keyword' tokens accurately. We've managed to fix these problems and also added some new values to the 'Operator' token. Additionally, we've introduced new tokens such as 'Parenthesis', 'Punctuation', 'Constant' and 'Header'. This report aims to outline these changes and their impact on our lexical analyzer's performance.

Header Token:

A notable addition to our updated code is the 'Header' token, a feature absent in previous code. In the prior codebase, headers were erroneously categorized as 'Identifier' tokens, leading to potential parsing inaccuracies. To rectify this, we implemented a dedicated recognition mechanism for headers. Within our code, a conditional check for preprocessor directives, denoted by the '#' symbol, triggers the identification and extraction of header values. Each identified header is stored in a designated data structure, facilitating efficient access and utilization during lexical analysis. This enhancement not only ensures accurate tokenization of headers but also contributes to the overall robustness and reliability of our lexical analyzer.

```
initialize header_count to 0
initialize headers array with size 100
while (not end of file) {
    read character from file
    if (character is '#') {
        read string from file until whitespace or newline
        store the string as a header in headers array at index header_count
        increment header_count
    }
}
print all headers identified
```

Identifier Token:

Improving how we identify identifiers was crucial for refining our lexical analysis. The previous code had several issues, like mistakenly labeling 'Header' as an 'Identifier' and missing the underscore character ('_') in identifiers. It also wrongly

identified the keyword 'char' and combined tokens like 'keywordlibrarychar', causing parsing errors. To fix these problems, we made significant improvements.

The updated code now uses a better method to recognize identifiers. It checks for alphanumeric characters and underscores using character functions and loops. Plus, we added a validation step to make sure keywords and constants aren't misclassified as identifiers. These changes help our code more accurately distinguish identifiers, reducing parsing mistakes.

```
initialize identifier_count to 0
initialize identifiers array with size 100
while (not end of file) {
    read character from file
    if (character is alphanumeric or underscore) {
        read characters from file until non-alphanumeric or underscore character or
        maximum identifier length is reached
        store the string as an identifier in the identifiers array at index identifier_count
        increment identifier_count
    }
}
```

Keyword Token:

In previous code, the keyword 'char' was mistakenly categorized as an identifier due to coding errors. However, in our updated code, this issue has been effectively resolved, and 'char' is now correctly identified as a keyword. The new code utilizes an array to store keywords and a function to check whether a given string matches any of the keywords. This approach ensures robust identification of keywords, including 'char' among others, enhancing the accuracy of our lexical analysis.

```
initialize keywords list with 32 predefined keywords
initialize is_keyword function to check if a given string is a keyword
function is_keyword(temp):
    for each keyword in keywords list:
        if keyword matches temp:
            return 1 // It's a keyword
    return 0 // Not a keyword
```

Operator Token:

Operators were successfully recognized in previous code, encompassing arithmetic operators such as "=", "+", "%", "*", "/", "-". However, in our updated code, we have expanded the repertoire to include logical operators like "!=", ">", "<", "=", "!", "==". Our code employs a more comprehensive approach to operator recognition. It utilizes a combination of character matching and a loop construct to identify both arithmetic and logical operators. Each operator encountered is validated against existing entries to avoid duplicate counting, ensuring accurate representation within the operator token set.

```
initialize operator_count to 0
initialize operators array with size 100
while (not end of file) {
    read character from file
    if (character is operator) {
        store the character as an operator in the operators array at index operator_count
        increment operator_count
    }
}
```

Punctuation Token:

In the previous code, there was a lack of capability to identify punctuation tokens, resulting in a gap in the lexical analysis. However, in our new code iteration, we've introduced a new token category specifically for punctuations, encompassing symbols like ',', ';', and ':'. The updated code incorporates a dedicated mechanism to recognize punctuation characters. Through conditional checks and a validation loop, each encountered punctuation is efficiently identified and stored in a punctuator list. This ensures that each punctuation symbol is accurately represented within the lexical stream.

```
initialize punctuator_count to 0
initialize punctuators array with size 100
while (not end of file) {
    read character from file
    if (character is punctuation) {
        store the character as a punctuator in the punctuators array at index punctuator_count
        increment punctuator_count
    }
}
```

Parenthesis Token:

The previous code lacked the capability to identify parenthesis tokens, which are crucial for syntactic analysis. However, in our latest code revision, we've introduced a dedicated token category for parentheses, encompassing symbols such as '(', ')', '{', '}', '[', and ']'. The updated code features a specialized mechanism to detect parenthesis characters. Through conditional checks, each encountered parenthesis is promptly recognized and added to a list for further analysis. Notably, parentheses are counted in pairs, ensuring proper nesting and structure within the code.

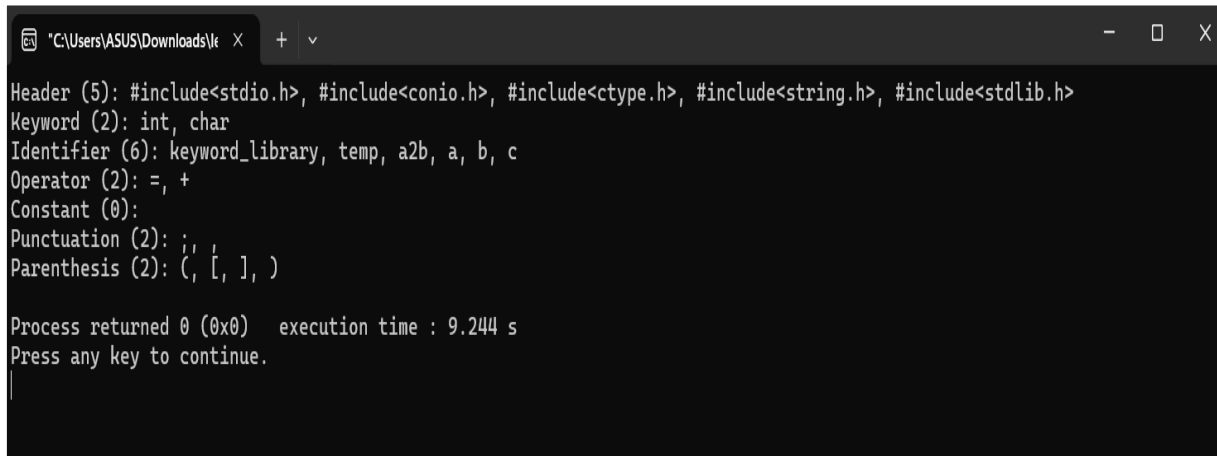
```
initialize punctuator_count to 0
initialize punctuators array with size 100
while (not end of file) {
    read character from file
    if (character is punctuation) {
        store the character as a punctuator in the punctuators array at index punctuator_count
        increment punctuator_count
    }
}
```

Constant Token:

It was not possible to detect constant letters in the prior code, hindering comprehensive lexical analysis. However, in our new code update, we've introduced a dedicated token category for constants. The newly implemented constant token is designed to recognize integer constants within the code. Through a systematic parsing process, the code iterates through each character of the input string, identifying digits and validating their sequence. If all characters are successfully consumed and found to be digits, the token is classified as an integer constant.

```
initialize constant_count to 0
initialize constants array with size 100
while (not end of file) {
    read character from file
    if (character is digit) {
        read characters from file until non-digit character or maximum constant length is reached
        store the string as a constant in the constants array at index constant_count
        increment constant_count
    }
}
```

Conclusion:



```
*C:\Users\ASUS\Downloads\le X + v
Header (5): #include<stdio.h>, #include<conio.h>, #include<ctype.h>, #include<string.h>, #include<stdlib.h>
Keyword (2): int, char
Identifier (6): keyword_library, temp, a2b, a, b, c
Operator (2): =, +
Constant (0):
Punctuation (2): ;, ,
Parenthesis (2): (, [, ], )

Process returned 0 (0x0)   execution time : 9.244 s
Press any key to continue.
```

Fig: Final Output

Our updated lexical analyzer demonstrates significant improvements in accurately recognizing and categorizing tokens within the code. Through the introduction of new token categories such as 'Header', 'Punctuation', 'Parenthesis', and 'Constant', along with enhancements to existing categories like 'Identifier', 'Keyword', and 'Operator', our code now offers enhanced precision and versatility in syntactic analysis. These advancements contribute to a more reliable and comprehensive lexical analysis process.