

Lab: 1

Lab Report: Setting up Environment for Lex and Flex Programs

Objective:

The main objective of this lab is to:

1. Follow the tutorial to set up the environment for developing Lex and Flex programs.
2. Install and configure the required tools, including **Flex** and **Bison**, to run lexical analysis programs.

Tools:

- **Flex (Fast Lexical Analyzer Generator)**: A tool for generating programs that perform lexical analysis.
- **Bison**: A parser generator that works with Flex to perform syntactic analysis.
- **Operating System**: Windows

Step-by-Step Installation of Flex and Bison

Step 1: Download the Necessary Files

- **Flex** and **Bison** executables can be downloaded from the provided URLs:
 - **Flex**: *flex-2.5.4a-1.exe*
 - **Bison**: *bison-2.4.1-setup.exe*

These files are used to install the Flex lexical analyzer and Bison parser generator.

Step 2: Installation of Flex

1. Download the *flex-2.5.4a-1.exe* file from the source.
2. Double-click on the file to start the installation process.
3. Follow the prompts and choose a location for installation (e.g., *C:\Flex*).
4. Complete the installation and ensure that the Flex directory is added to the system's PATH environment variable.

Step 3: Installation of Bison

1. Download the ***bison-2.4.1-setup.exe*** file.
2. Run the setup and follow the installation prompts, selecting a destination folder (e.g., *C:\Bison*).
3. Complete the installation and make sure that the Bison installation directory is also added to the system's PATH.

Step 4: Setting the PATH Environment Variable

1. Open **System Properties** by right-clicking **This PC** and choosing **Properties**.
2. Click on **Advanced system settings** on the left.
3. Under the **System Properties** window, click the **Environment Variables** button.
4. In the **Environment Variables** window, locate the **Path** variable under **System Variables**, select it, and click **Edit**.
5. In the **Edit Environment Variable** window, click **New** and add the paths for both Flex and Bison:
 - For Flex: *C:\Flex\bin*
 - For Bison: *C:\Bison\bin*
6. Click **OK** to save the changes.

Step 5: Verifying the Installation

1. Open **Command Prompt** (*cmd*).
2. Type the following commands to verify that Flex and Bison are installed correctly:
 - For Flex: *`flex --version`*
 - For Bison: *`bison --version`*

A Sample Lex Program

After the successful installation, we can write a basic Lex program to verify that the setup is correct.

Sample Lex Program to Identify Keywords and Identifiers:

1. Create a text file with the `.l` extension (e.g., `keywords.l`).
2. Write the following Lex code:

```
%{  
#include <stdio.h>  
%}  
%%  
  
int|float|char    { printf("Keyword: %s\n", yytext); }  
[a-zA-Z_][a-zA-Z0-9_]* { printf("Identifier: %s\n", yytext); }  
%%  
  
int main() {  
    yylex();  
    return 0;  
}
```

Step 6: Compiling and Running the Lex Program

1. Open **Command Prompt** and navigate to the directory where our Lex file is located.
2. Run the following command to compile the Lex program:

```
flex keywords.l
```

This will generate a file named `lex.yy.c`.

3. Compile the generated C file using a C compiler (e.g., GCC):

```
gcc lex.yy.c -o keywords
```

4. Run the compiled program:

```
./keywords
```

5. Input some test data (e.g., `int myVariable`) and check if it identifies keywords and identifiers.

Conclusion:

In this lab, we successfully installed and configured **Flex** and **Bison** on a Windows system. A basic Lex program was written, compiled, and executed to identify keywords and identifiers in the input source code, verifying that the setup works as expected. The installation process and the sample program provide a foundation for writing more complex lexical analyzers in future labs.

References:

- Lex Programs Tutorial: <https://silcnitc.github.io/lex.html>

-YouTube: Flex and Bison Installation & Configuration (English Subtitles):
<https://www.youtube.com/watch?v=sabcvideo>

Lab: 2

Lab Report: Lex Programs for Keyword, Identifier, Character, Word, Space, NewLine, Operators, and Comments.

Objective:

The objectives of this lab are:

1. To develop a Lex program to identify keywords and identifiers in source code.
2. To create a Lex program that recognizes characters, words, spaces, and newlines in a given input.
3. To write a Lex program that identifies operators, single-line comments, and multi-line comments.

Theory:

Lex is a lexical analyzer generator used to define tokens and patterns in source code. It reads the input stream character by character and matches it against user-defined patterns. These patterns are translated into tokens, which are passed to the parser for syntactic analysis. The tokens can represent keywords, identifiers, operators, comments, or any other elements in programming languages.

Key Concepts:

1. **Keywords:** Reserved words in a programming language that have a special meaning.
2. **Identifiers:** Names used to identify variables, functions, or user-defined elements in code.
3. **Characters and Words:** Individual characters and groupings of letters and digits.
4. **Operators:** Symbols such as `+`, `-`, `*`, `/` used for arithmetic or logical operations.
5. **Comments:** Annotations in the source code that are ignored by the compiler. These include both single-line and multi-line comments.

Program 1: Identifying Keywords and Identifiers

Code:

```
%{  
  
#include <stdio.h>%}
```

```

%%

"int"|"float"|"if"|"else"|"while"|"return" { printf("KEYWORD: %s\n", yytext); }

[a-zA-Z_][a-zA-Z0-9_]* { printf("IDENTIFIER: %s\n", yytext); }

[ \t]+ ; // Ignore spaces and tabs

\n ; // Ignore newlines

. { printf("UNKNOWN: %s\n", yytext); }

%%

int main() {

    yylex();

    return 0;

}

int yywrap() {

    return 1;

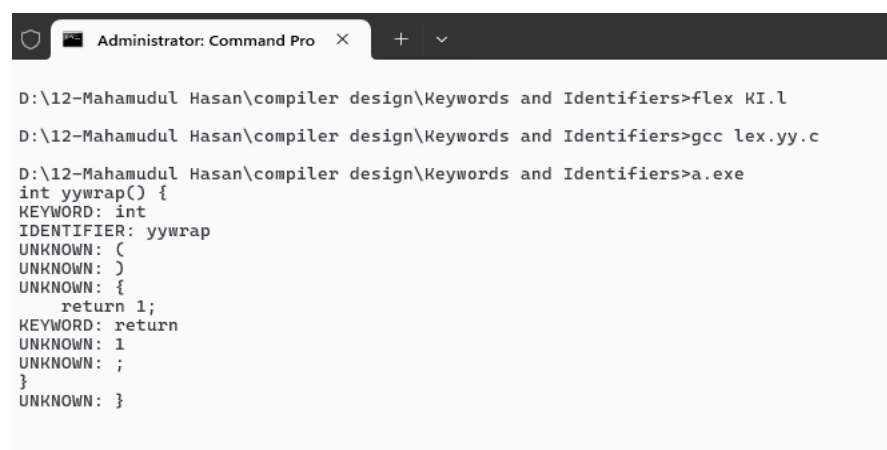
}

```

Explanation:

- **Pattern 1:** Matches reserved keywords like `if`, `else`, `while`, etc., and prints them as keywords.
- **Pattern 2:** Matches identifiers (any combination of letters, digits, and underscores starting with a letter or underscore).
- **Pattern 3:** Skips whitespaces (` ` , `\t`, `\n`).
- **Pattern 4:** Prints unknown characters as tokens not matching any pattern.

Input & Output



```

Administrator: Command Prom...
D:\12-Mahamudul Hasan\compiler design\Keywords and Identifiers>flex KI.l
D:\12-Mahamudul Hasan\compiler design\Keywords and Identifiers>gcc lex.yy.c
D:\12-Mahamudul Hasan\compiler design\Keywords and Identifiers>a.exe
int yywrap() {
KEYWORD: int
IDENTIFIER: yywrap
UNKNOWN: (
UNKNOWN: )
UNKNOWN: {
    return 1;
KEYWORD: return
UNKNOWN: 1
UNKNOWN: ;
}
UNKNOWN: }

```

Program 2: Identifying Characters, Words, Spaces, and Newlines

Code:

```
%{
#include <stdio.h>

int characters = 0, words = 0, spaces = 0, newlines = 0;

%}

%%

[a-zA-Z0-9]+ { words++; characters += yyleng; }

" "      { spaces++; characters++; }

\n      { newlines++; characters++; }

.        { characters++; }

%%

int main() {
    yylex();

    printf("Words: %d\nSpaces: %d\nCharacters: %d\nNewlines: %d\n", words, spaces,
characters, newlines);

    return 0;
}
```

Explanation:

- **Pattern 1:** Matches words (combination of letters or digits) and counts them as words and characters.
- **Pattern 2:** Matches spaces and increments the space count and character count.
- **Pattern 3:** Matches newlines and increments the newline count and character count.
- **Pattern 4:** Matches all other characters and counts them as characters.

Sample Input:

Hello World

This is a test.

Sample Output:

Words: 5

Spaces: 4

Characters: 24

Newlines: 2

Program 3: Identifying Operators, Single-line Comments, and Multi-line Comments

Code:

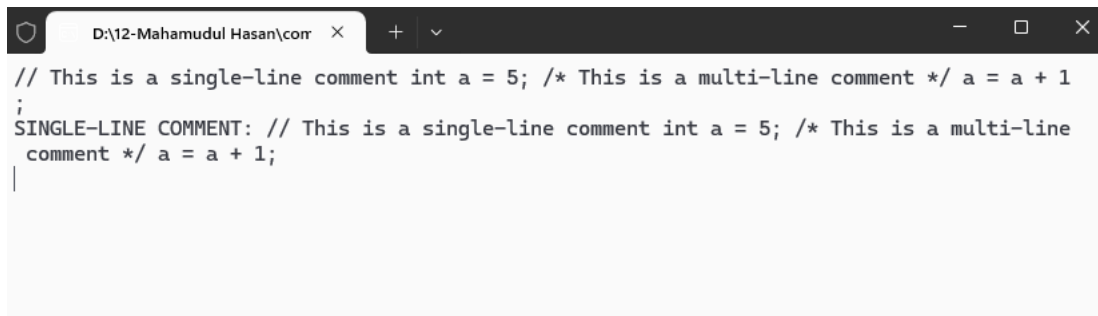
```
%{  
#include <stdio.h>  
%}  
%%  
"+"|"-"|"*"|"\/|"="          { printf("Operator: %s\n", yytext); }  
"//".*                        { printf("Single-line Comment: %s\n", yytext); }  
"/*"([^\n]|[\r\n]|(\n+([^\n]|[\r\n])))**+/" { printf("Multi-line Comment: %s\n", yytext); }  
[\t\n]+                        ; // Ignore whitespaces  
.  
                                { printf("Unknown token: %s\n", yytext); }  
  
%%  
  
int main() {  
    yylex();  
    return 0;  
}
```

Explanation:

- **Pattern 1:** Matches arithmetic and assignment operators like `+`, `-`, `*`, `/`, and `=`.
- **Pattern 2:** Matches single-line comments starting with `//` and followed by any characters.
- **Pattern 3:** Matches multi-line comments that begin with `/*` and end with `*/`.

- **Pattern 4:** Skips whitespaces.
- **Pattern 5:** Prints unknown tokens that do not match any pattern.

Sample Input & Output:



```
D:\12-Mahamudul Hasan\corr X + - □ X
// This is a single-line comment int a = 5; /* This is a multi-line comment */ a = a + 1
;
SINGLE-LINE COMMENT: // This is a single-line comment int a = 5; /* This is a multi-line
comment */ a = a + 1;
|
```

Conclusion:

In this lab, we successfully wrote and executed three Lex programs that identified different token types in a given input. The first program identified **keywords** and **identifiers**, the second identified **characters, words, spaces, and newlines**, and the third identified **operators, single-line comments**, and **multi-line comments**. These Lex programs form the building blocks for lexical analyzers used in compilers to tokenize source code for further analysis.

References:

- Lex and Yacc (O'Reilly): <https://www.oreilly.com/library/view/lex-yacc/9780596805418/>
- Lex Manual: <https://westes.github.io/flex/manual>

Lab: 3

Lab Report: Lex Programs for Identifying Number Data Types, Negative Fractions, Functions, Pre-processor Directives, and Delimiters

Objective:

The objectives of this lab are:

1. To write a Lex program that identifies number data types in the source code.
2. To create a Lex program that identifies negative fractions and functions in a given input.
3. To develop a Lex program to identify pre-processor directives and delimiters.

Theory:

Lexical analysis is the first phase of the compiler where the input source code is divided into tokens. Tokens represent basic language constructs such as keywords, operators, and identifiers. In this lab, we will focus on identifying specific patterns related to number data types, negative fractions, functions, pre-processor directives, and delimiters using Lex programs.

Key Concepts:

1. **Number Data Types:** Represent different forms of numeric values, such as integers and floating-point numbers.
2. **Negative Fractions:** Fractions or floating-point numbers that are negative.
3. **Functions:** Identifiable by their naming convention followed by parentheses.
4. **Pre-processor Directives:** Instructions provided to the compiler to process before actual compilation begins, such as `#include` or `#define`.
5. **Delimiters:** Symbols used to separate or define blocks of code, such as semicolons (`;`), commas (`,`), or braces (`{}`, `[]`).

Program 1: Identifying Number Data Types

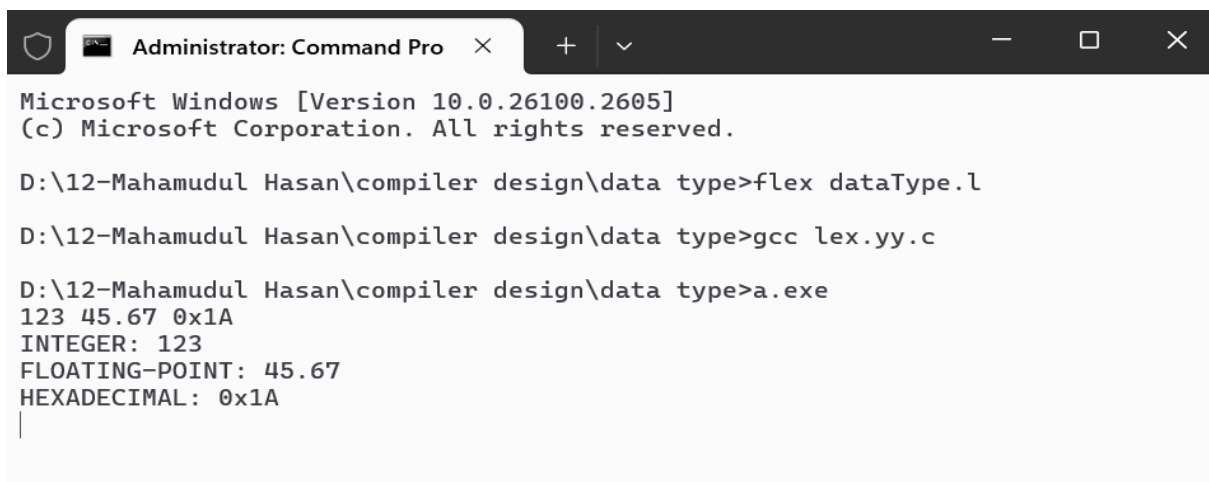
Code:

```
%{  
#include <stdio.h>  
  
%}  
%%  
  
[0-9]+      { printf("Integer: %s\n", yytext); }  
[0-9]+ "." [0-9]+  { printf("Floating Point: %s\n", yytext); }  
"0x"[0-9a-fA-F]+  { printf("Hexadecimal: %s\n", yytext); }  
[ \t\n]      ; // ignore whitespaces  
.  
      { printf("Unknown token: %s\n", yytext); }  
%%  
  
int main() {  
    yylex();  
    return 0;  
}
```

Explanation:

- **Pattern 1:** Matches integers (e.g., 123, 456).
- **Pattern 2:** Matches floating-point numbers with a decimal point (e.g., 123.45, 0.001).
- **Pattern 3:** Matches hexadecimal numbers starting with `0x` (e.g., 0x1A, 0xFF).
- **Pattern 4:** Skips whitespaces.

Sample Input:



```
Administrator: Command Pro  
Microsoft Windows [Version 10.0.26100.2605]  
(c) Microsoft Corporation. All rights reserved.  
  
D:\12-Mahamudul Hasan\compiler design\data type>flex dataType.l  
  
D:\12-Mahamudul Hasan\compiler design\data type>gcc lex.yy.c  
  
D:\12-Mahamudul Hasan\compiler design\data type>a.exe  
123 45.67 0x1A  
INTEGER: 123  
FLOATING-POINT: 45.67  
HEXADECIMAL: 0x1A  
|
```

Program 2: Identifying Negative Fractions and Functions

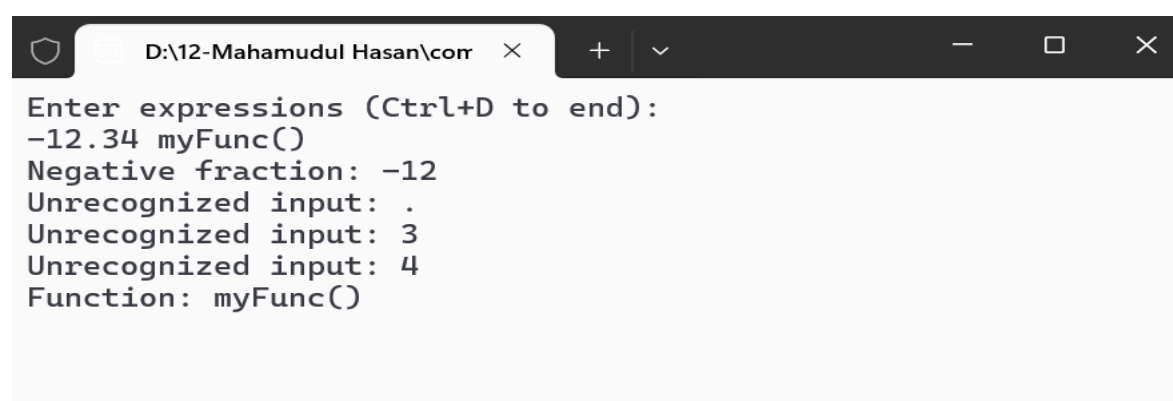
Code:

```
%{  
#include <stdio.h>  
%}  
%%  
  
"-"[0-9]+"."[0-9]+      { printf("Negative Fraction: %s\n", yytext); }  
[a-zA-Z_][a-zA-Z0-9_]*()" { printf("Function: %s\n", yytext); }  
[ \t\n]                ; // Ignore whitespaces  
.  
                        { printf("Unknown token: %s\n", yytext); }  
%%  
  
int main() {  
    yylex();  
    return 0;  
}
```

Explanation:

- **Pattern 1:** Matches negative fractions (e.g., -0.5, -123.456).
- **Pattern 2:** Matches function names followed by parentheses (e.g., `main()`, `compute()`).
- **Pattern 3:** Skips whitespaces.
- **Pattern 4:** Prints unknown characters as tokens that do not match any pattern.

Sample Input:



```
Enter expressions (Ctrl+D to end):  
-12.34 myFunc()  
Negative fraction: -12  
Unrecognized input: .  
Unrecognized input: 3  
Unrecognized input: 4  
Function: myFunc()
```

Program 3: Identifying Pre-processor Directives and Delimiters

Code:

```
%{
#include <stdio.h>

%}

%%

"#include"|"#define"|"#ifdef"|"#ifndef" { printf("Pre-processor Directive: %s\n", yytext); }

"{"|"}"|"("|")"|"["|"]"|"."|","|";" { printf("Delimiter: %s\n", yytext); }

[ \t\n]          ; // Ignore whitespaces

.                { printf("Unknown token: %s\n", yytext); }

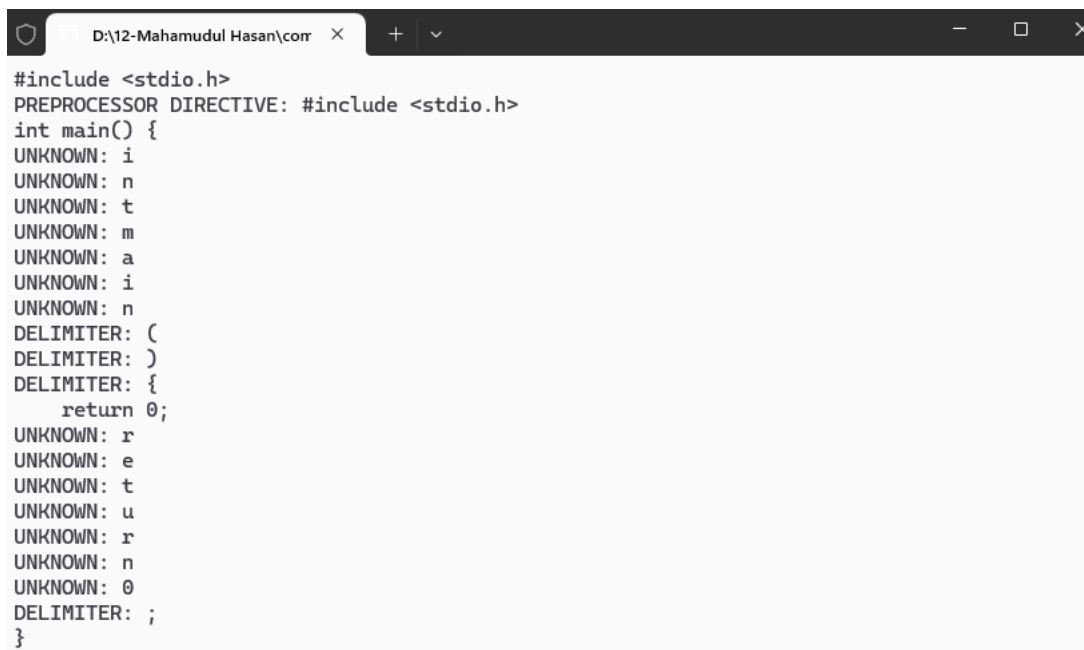
%%

int main() {
    yylex();
    return 0;
}
```

Explanation:

- **Pattern 1:** Matches common pre-processor directives such as ``#include``, ``#define``, ``#ifdef``, and ``#ifndef``.
- **Pattern 2:** Matches common delimiters such as curly braces ``{}``, parentheses ``()``, brackets ``[]``, semicolons ``;``, and commas ```,``.
- **Pattern 3:** Skips whitespaces.
- **Pattern 4:** Prints unknown characters as tokens that do not match any pattern.

Sample Input & Output:



```
#include <stdio.h>
PREPROCESSOR DIRECTIVE: #include <stdio.h>
int main() {
UNKNOWN: i
UNKNOWN: n
UNKNOWN: t
UNKNOWN: m
UNKNOWN: a
UNKNOWN: i
UNKNOWN: n
DELIMITER: (
DELIMITER: )
DELIMITER: {
    return 0;
UNKNOWN: r
UNKNOWN: e
UNKNOWN: t
UNKNOWN: u
UNKNOWN: r
UNKNOWN: n
UNKNOWN: 0
DELIMITER: ;
}
```

Conclusion:

In this lab, we successfully wrote and executed three Lex programs that identified specific tokens in the input. The first program identified **number data types** such as integers, floating-point numbers, and hexadecimal numbers. The second program identified **negative fractions** and **function definitions**, and the third program identified **pre-processor directives** and **delimiters**. These Lex programs are useful for building lexical analyzers that can tokenize various components of a programming language.

References:

- Lex and Yacc (O'Reilly): <https://www.oreilly.com/library/view/lex-yacc/9780596805418/>
- Flex Manual: <https://westes.github.io/flex/manual>