Assignment: 1

Write a program using LEX specifications to implement lexical analysis phase of compiler to generate tokens of subset of 'C' program.

file.l

```
%{


%}


letter [A-Za-z]

digit [0-9]

identifier {letter}({letter}|{digit})*

number {digit}+(\.{digit}+)?

punctuation [;,."#(){}]

operators [+=*<>]

keywords if|else|for|int|while|do|void

headerfile "#",*

literal ["].*["]

comment "//".*

multicommnt "/*"(.|\n)*"*/"


%%



{headerfile} {printf("\n headerfile %s is found",yytext);}

{keywords} {printf("\n keywords %s if found",yytext);}
```

```
{identifier} {printf("\n identifier %s if found",yytext);}
{number} {printf("\n number %s if found",yytext);}
{punctuation} {printf("\n punctuation %s if found",yytext);}
{operators} {printf("\n operators %s if found",yytext);}
{literal} {printf("\n literal %s if found",yytext);}
{comment} {printf("\n comment %s if found",yytext);}
{multicommnt} {printf("\n multicommnt %s if found",yytext);}

%%

int main(void)
{
   yylex();
   return 0;

}

int yywrap()
{
   return 1;

}
```

Input.c

```c
#include<stdio.h>

void main()
{
 int a,b,c;
  a=4;
  b=5;

 c=a+b;  // addition
 printf("Accept number a");
/* number is
accepted */

 getch();
}
```

Output: lex file.l

cc lex.yy.c

./a.out < one.c

Certainly! Let's break down this program line by line, along with the relevant theory.

Introduction

This is a Flex (Fast Lexical Analyzer Generator) program. Flex is a tool for generating scanners: programs which recognize lexical patterns in text. The

program provided is a simple lexical analyzer that can recognize different tokens in a given input.

Definitions Section

```c
%{
%}
```

The `%{` and `%}` denote the C code section that is copied verbatim into the generated C file. Here it is empty, so no additional C code is added at the beginning of the generated file.

Patterns Section

```c
letter [A-Za-z]
digit [0-9]
identifier {letter}({letter}|{digit})*
number {digit}+(\.{digit}+)?
punctuation [;,."#(){}]
operators [+=*<>]
keywords if|else|for|int|while|do|void
headerfile "#".*
```

literal ["].*["]

comment "//".*

multicommnt "/*"(.|\n)*"*/"

```

This section defines regular expressions for different types of tokens:

- `letter`: Matches any uppercase or lowercase alphabetic character.
- `digit`: Matches any digit from 0 to 9.
- `identifier`: Matches a sequence starting with a letter followed by any number of letters or digits.
- `number`: Matches a sequence of digits, possibly containing a decimal point.
- `punctuation`: Matches any single punctuation character from the specified set.
- `operators`: Matches any single operator from the specified set.
- `keywords`: Matches any of the specified keywords (e.g., `if`, `else`, etc.).
- `headerfile`: Matches a `#` followed by any number of characters (used for preprocessing directives).
- `literal`: Matches a string enclosed in double quotes.
- `comment`: Matches a single-line comment starting with `//`.
- `multicommnt`: Matches a multi-line comment enclosed in `/* ... */`.

### Rules Section

```c
%%

```

The `%%` marks the beginning of the rules section where each pattern is associated with an action.

```c
{headerfile} {printf("\n headerfile %s is found",yytext);}
{keywords} {printf("\n keywords %s if found",yytext);}
{identifier} {printf("\n identifier %s if found",yytext);}
{number} {printf("\n number %s if found",yytext);}
{punctuation} {printf("\n punctuation %s if found",yytext);}
{operators} {printf("\n operators %s if found",yytext);}
{literal} {printf("\n literal %s if found",yytext);}
{comment} {printf("\n comment %s if found",yytext);}
{multicommnt} {printf("\n multicommnt %s if found",yytext);}
```

For each pattern, the action is to print a message indicating the type of token found along with the matched text (`yytext` is a global variable containing the matched text).

User Code Section

```c
%%
```
```

```

This `%%` marks the end of the rules section. The following code is the user code section, which typically includes the main function and other C code.

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

- `int main(void)`: Defines the main function of the program.
- `yylex()`: This function is generated by Flex and starts the scanning process. It reads input, matches it against the patterns, and executes the corresponding actions.
- `return 0;`: Indicates that the program terminates successfully.

```c
int yywrap()
{
    return 1;
}
```
```

- `int yywrap()`: This function is called by `yylex()` when the end of the input is reached. Returning 1 indicates that there is no more input to process and `yylex()` should stop.

Summary

- **Definitions section**: Defines the patterns for tokens.
- **Rules section**: Associates patterns with actions.
- **User code section**: Contains the main function to start the lexical analysis and the `yywrap` function to handle end-of-input.

When you compile and run this program, it will read from standard input, match the input against the defined patterns, and print messages indicating which tokens it has recognized.