COMPILER DESIGN LAB (CO351)
REPORT
Project 1 : Scanner(Lexical Analyzer) for C-language

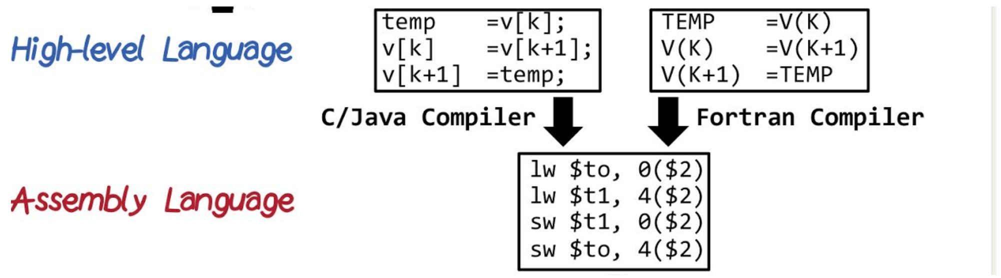
Submitted by :-

(1) Patel Harsh Rakeshkumar (15CO233)

(2) Shah Harsh Kamlesh (15CO243)

INTRODUCTION OF A COMPILER AND PHASES INVOLVED

- Compiler - A program that translates a program from source language into a target language is called a compiler.



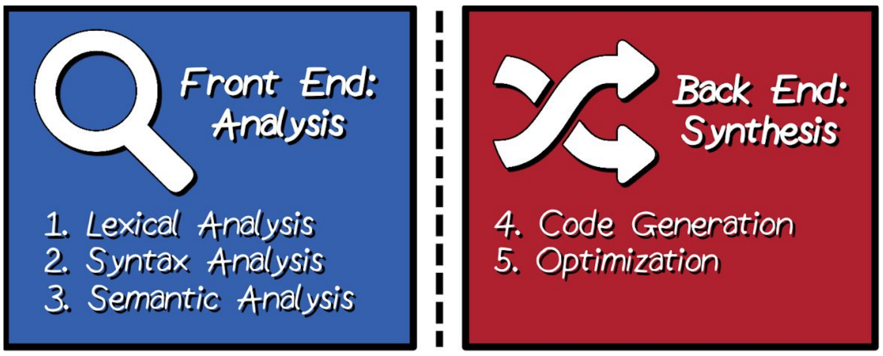
```
temp     =v[k];
v[k]     =v[k+1];
v[k+1]   =temp;
```

```
TEMP     =V(K)
V(K)     =V(K+1)
V(K+1)   =TEMP
```

```
lw $to, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $to, 4($2)
```

- The conversion from high-level language i.e. the source language to target language i.e. assembly language is obtained using the help of a compiler.
- As we can see in the figure on the right, in a language processing system, compiler converts the modified source program into target assembly program.

STRUCTURE OF A COMPILER



➢ A compiler is divides into 2 main parts:-
(1) Analysis (Front-end)
(2) Synthesis (Back-end)

➢ The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them.
It then uses this structure to create an intermediate representation of the

source program.
If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action.
The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

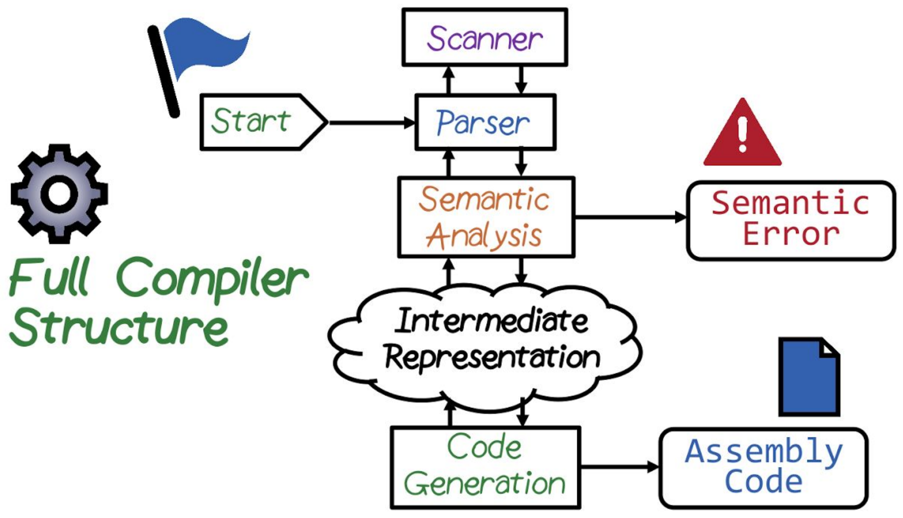
- ➢ The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table.
- ➢ The analysis part is further divided into 4 stages :-
  (1) Lexical Analyzer
  (2) Syntax Analyzer
  (3) Semantic Analyzer
  (4) Intermediate Code Generator
- ➢ At the end of the analysis stage the input source file is converted into ‘intermediate representation’.
- ➢ The synthesis part is further divided into 2 stages :-
  (1) Code-Optimizer
    (a) Machine-Independent Code Optimizer
    (b) Machine-Dependent Code Optimizer
  (2) Code Generator

Following is brief introduction about all the parts of the compiler spread along the analysis and the synthesis phase :-

- ❏ **Lexical Analysis -** The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called *lexemes.*For each lexeme, the lexical analyzer produces as output a *token* of form <u><token name, attribute-value></u> that it passes on to the subsequent phase, syntax analysis.
- ❏ **Syntax Analysis -** Syntax analysis is also known as parsing. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a *syntax tree* in which each interior node represents an operation and the children of the node represent the arguments of the operation.
- ❏ **Semantic Analysis -** The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.
- ❏ **Intermediate Code Generation -** After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or

machine-like intermediate representation, which we can think of as a program for an abstract machine. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine.

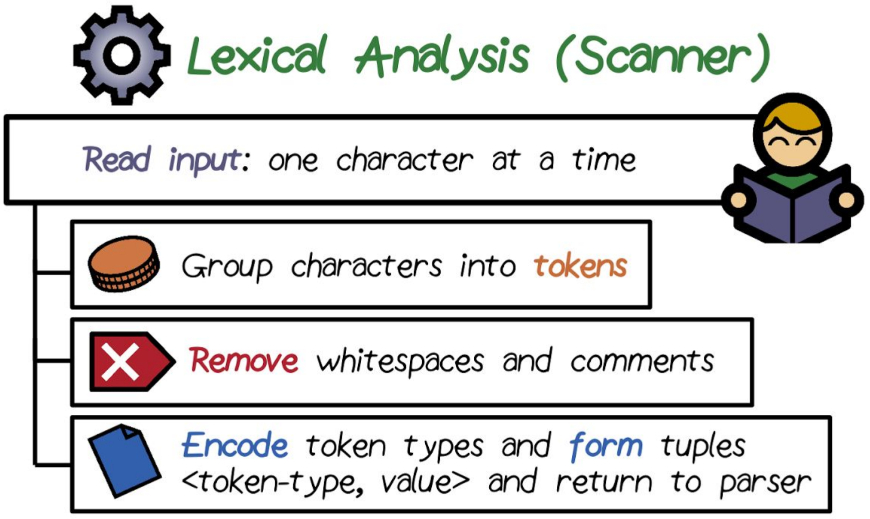
- **Code Optimization -** The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.
- **Code Generation -** The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task. A crucial aspect of code generation is the judicious assignment of registers to hold variables.



- Also, the compiler performs certain other crucial operation which are required to convert the source language into the target language. They are as follows :-
  - **Symbol-Table Management -** The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.
  - **The Grouping of Phases into Passes -** The discussion of phases deals with the logical organization of a compiler. In an implementation, activities from several phases may be grouped together into a pass that reads an input file and writes an output file.
- Following are the applications of compiler technology :-
  - Implementation of High-Level Programming Languages
  - Optimizations for Computer Architectures

        - Parallelism
        - Memory Hierarchies
    - Design of New Computer Architectures
        - RISC (Reduced Instruction-Set Computer)
        - Specialized Architectures
    - Program Translations
        - Binary Translation
        - Hardware Synthesis
        - Database Query Interpreters
        - Compiled Simulation
    - Software Productivity Tools
        - Type Checking
        - Bounds Checking
        - Memory-Management Tools

LEXICAL ANALYSIS


Lexical Analysis (Scanner)
Read input: one character at a time
Group characters into tokens
Remove whitespaces and comments
Encode token types and form tuples <token-type, value> and return to parser

The role of a Lexical Analyzer

➢ The main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program.

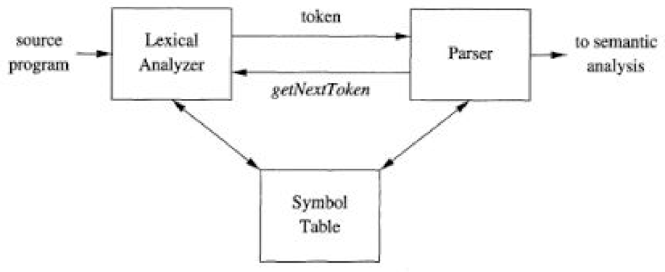
➢ The stream of tokens is sent to the parser for syntax analysis.

➢ It is common for the lexical analyzer to interact with the symbol table as well.

➢ When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.

➢ In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.

➢ Following are the interactions between the lexical analyzer :-



➢ Sometimes, lexical analyzers are divided into a cascade of two processes:
  - *Scanning* consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.
  - *Lexical analysis* proper is the more complex portion, where the scanner produces the sequence of tokens as output.

➢ A *token* is a pair consisting of a token name and an optional attribute value.

➢ A *pattern* is a description of the form that the lexemes of a token may take.

➢ A *lexeme* is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

➢ In many programming languages, the following classes cover most or all of the tokens:
  - One token for each keyword. The pattern for a keyword is the same as the keyword itself.
  - Tokens for the operators, either individually or in classes.
  - One token representing all identifiers.
  - One or more tokens representing constants, such as numbers and literal strings.
  - Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

Lexical Errors

➢ Following are the different types of lexical errors that our program handles :-
  - Unterminated comment
  - Invalid characters
  - Ill-formed identifiers

Input Buffering

➢ <u>Buffer pairs</u> - Two pointers to the input are maintained :-
  - Pointer *lexemeBegin*, marks the beginning of the current lexeme, whose extent we are attempting to determine.
  - Pointer *forward* scans ahead until a pattern match is found.

- Sentinels - The sentinel is a special character that cannot be part of the source program, and a natural choice is the character ‘eof’(end of file).

Specification of Tokens

- Strings and Languages
    - An *alphabet* is any finite set of symbols.
    - A *string* over an alphabet is a finite sequence of symbols drawn from that alphabet.
    - A *language* is any countable set of strings over some fixed alphabet.
- Operations on Languages
    - Union
    - Concatenation
    - Kleene closure
    - Positive closure
- Regular Expressions - A sequence of symbols and characters expressing a string or pattern to be searches for within a longer piece of text.
- Extensions of Regular Expressions
    - One or more instances - The unary, postfix operator + represents the positive closure of a regular expression and its language.
    - Zero or one instance - The unary postfix operator ? means "zero or one occurrence."
    - Character classes - A regular expression $a_1 \mid a_2 \mid \ldots \mid a_n$, where the a_1's are each symbols of the alphabet, can be replaced by the shorthand $[a_1a_2\ldots a_n]$.

Recognition of Tokens

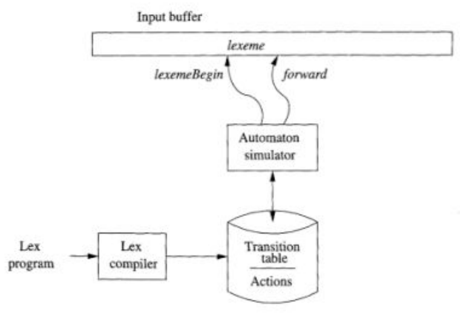
- Transition Diagrams - Transition diagrams have a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns. Edges are directed from one state of the transition diagram to another.
  Some important conventions about transition diagrams :-
    - Certain states are said to be *accepting*, or *final*.
    - In addition, if it is necessary to retract the *forward pointer* one position then we shall additionally place a * near that accepting state.
    - One state is designated the *start state*, or *initial state*.
- Recognition of Reserved Words and Identifiers - There are two ways that we can handle reserved words that look like identifiers :-
    - Install the reserved words in the symbol table initially.
    - Create separate transition diagrams for each keyword.

Finite Automata

- ➢ Nondeterministic Finite Automata - A nondeterministic finite automaton (NFA) consists of :-
  - ○ A finite set of states S.
  - ○ A set of input symbols C, the input alphabet. We assume that ϵ stands for the empty string, is never a member of **Σ**.
  - ○ A transition function that gives, for each state, and for each symbol in **Σ**U{ϵ} a set of next states.
  - ○ A state so from S that is distinguished as the start state (or initial state).
  - ○ A set of states F, a subset of S, that is distinguished as the accepting states (or final states).
- ➢ Transition Tables - A transition table is a table whose rows correspond to states, and whose columns correspond to the input symbols and **ϵ**.
- ➢ Acceptance of Input Strings by Automata - An NFA accepts input string x if and only if there is some path in the transition graph from the start state to one of the accepting states, such that the symbols along the path spell out x.
- ➢ Deterministic Finite Automata - A deterministic finite automaton (DFA) is a special case of an NFA where :-
  - ○ There are no moves on input ϵ
  - ○ For each state s and input symbol a, there is exactly one edge out of s labeled a.

Design of a Lexical- Analyzer Generator

- ➢ The Structure of the Generated Analyzer - The components are :-
  - ○ A transition table for the automaton.
  - ○ Those functions that are passed directly through Lex to the output.
  - ○ The actions from the input program, which appear as fragments of code to be invoked at the appropriate time by the automaton simulator.



IMPLEMENTATION

The following points describe the functionalities implemented in the lexical analyzer:-

- The lexical analyser takes source program as input file and converts it into 3 parts :- **Stream of Tokens**, **Symbol Table** and **Constant Table**.
- The lexical analyzer uses **Hash Organization** to store entries into Symbol table and constant table.
- It includes the list of all the recognized **Identifiers** in the Symbol Table.
- All the **literals** - constants and strings are stored in the Constant Table.
- It recognizes **lexical errors** such as unterminated comments and invalid characters.
- Also, the **line number** is printed along with the lexical error message.
- For **functions**, it considers the function name as identifier and the braces as special characters which will be used to identify function during parsing phase. Also, the **looping constructs** like for and while are keywords and are similarly further processed during parsing. The **array** name is identifier and the braces as special characters as mentioned above.

CODE - Lexical Analyzer

```
%{
#include<string.h>
int comment_nesting = 0;
int lineno=1;
struct stable{
        char name[100];
        char type[50];
}symbol_table[1000];
struct ctable{
        char name[100];
        char type[50];
}constant_table[1000];
int hash_cal(char * str)
{
        int i;
        int sum=0;
        for(i=0;i<100;i++)
        {
                if(str[i]=='\0')
                {
                        break;
                }
                else{
                        sum+=(int)str[i];
                        sum=sum%1000;
                }
        }
        return sum;
}
%}

keyword
“auto"|"double"|"int"|"struct"|"break"|"else"|"long"|"switch"|"case"|"enum"|"register"|"typedef"|"
char"|"extern"|"return"|"union"|"const"|"float"|"short"|"unsigned"|"continue"|"for"|"signed"|"void
"|"default"|"goto"|"sizeof"|"volatile"|"do"|"if"|"static"|"while"
identifier [A-Za-z_]([A-Za-z_]|[0-9])*
notid [0-9]+([A-Za-z_]|[0-9])*
number (([0-9])*)(\.(([0-9])+))?(([eE])([\+\-]?)(([0-9])+))?
ws " "|" "|"\n"

spclsymb ";"|","|"#"|"("|")"|"["|"]"|"{"|"}"
op
"&"|"+"|"-"|"*"|"/"|"%"|"="|"=="|"!="|"<"|">"|"<="|">="|"!"|"<>"|"?"|":"|"++"|"--"|"+="|"-="|"*="|"/="|"."

%x SC_COMMENT

%%

"/*"            { comment_nesting++; BEGIN(SC_COMMENT); }
<SC_COMMENT>{
  "/*"          { comment_nesting++; }
  "*"+"/"         { comment_nesting=0;
                  BEGIN(INITIAL);  }
  "*"+          ;
  [^/*\n]+        ;
  [/]           ;
  \n            {lineno++;}
}

\"[^\n]*\" {printf("%s - string\n",yytext);

                                int k=hash_cal(yytext);
                                        struct ctable temp;
                                        strcpy(temp.name,yytext);
                                        strcpy(temp.type,"string constant");
                                        int i=0;
                                        int flag=0;
                                        for(i=0;i<1000;i++)
                                        {
                                                if(constant_table[k].name[0]=='\0')
                                                {
                                                        break;
                                                }
                                                else
                                                {

if(strcmp(yytext,constant_table[k].name)==0)
                                                        {
                                                                flag=1;
                                                                break;
                                                        }
                                                        k=(k+1)%1000;
                                                }
                                        }
                                        if(flag==0)
                                        {
                                                constant_table[k]=temp;
                                        }
                        }

\/\/[^\n]*\n {lineno++;}

{keyword} {printf("%s - keyword\n",yytext);}
{number} {printf("%s - constant\n",yytext);
                                int k=hash_cal(yytext);
                                        struct ctable temp;
                                        strcpy(temp.name,yytext);
                                        strcpy(temp.type,"numeric constant");
                                        int i=0;
                                        int flag=0;
                                        for(i=0;i<1000;i++)
                                        {
                                                if(constant_table[k].name[0]=='\0')
                                                {
                                                        break;
                                                }
                                                else
                                                {

if(strcmp(yytext,constant_table[k].name)==0)
                                                        {
                                                                flag=1;
                                                                break;
                                                        }
                                                        k=(k+1)%1000;
                                                }
                                        }
                                        if(flag==0)
                                        {
                                                constant_table[k]=temp;
                                        }
                        }
{notid} {printf("ERROR - %s : ill-formed identifier at line [%d]\n",yytext,lineno);}
{identifier} {printf("%s - identifier\n",yytext);
                                        int k=hash_cal(yytext);
                                        struct stable temp;
                                        strcpy(temp.name,yytext);
                                        strcpy(temp.type,"identifier");
                                        int i=0;
                                        int flag=0;
                                        for(i=0;i<1000;i++)
                                        {
                                                if(symbol_table[k].name[0]=='\0')
                                                {
                                                        break;
                                                }
                                                else
                                                {

if(strcmp(yytext,symbol_table[k].name)==0)
                                                        {
                                                                flag=1;
                                                                break;
                                                        }
                                                        k=(k+1)%1000;
                                                }
                                        }
                                        if(flag==0)
                                        {
                                                symbol_table[k]=temp;
                                        }

                                }
{spclsymb} {printf("%s - special character\n",yytext);}
{op} {printf("%s - operator\n",yytext);}
{ws} { if(yytext[0]=='\n') {lineno++;} }
. {printf("ERROR - %s : invalid character at line [%d]\n",yytext,lineno);}

%%
```

```
int main()
{
	int i;
	for(i=0;i<1000;i++)
	{
		symbol_table[i].name[0]='\0';
		symbol_table[i].type[0]='\0';
	}

	for(i=0;i<1000;i++)
	{
		constant_table[i].name[0]='\0';
		constant_table[i].type[0]='\0';
	}

	yyin = fopen("test2.txt","r");
	printf("******Tokens*****\n\n");
	yylex();
	if(comment_nesting!=0)
	{
		printf("ERROR : unterminated comment\n");
	}


	printf("\n*****Symbol Table******\n\n");
	for(i=0;i<1000;i++)
	{
		if(symbol_table[i].name[0]!='\0')
		{
			printf("%s	%s\n",symbol_table[i].name,symbol_table[i].type);
		}
	}


	printf("\n*****Constant Table******\n\n");
	for(i=0;i<1000;i++)
	{
		if(constant_table[i].name[0]!='\0')
		{
			printf("%s	%s\n",constant_table[i].name,constant_table[i].type);
		}
	}
	return 0;
}
int yywrap()
{
        return 1;
}
```

TEST CASES

Test Case 1:

```
int main()
{
  // single line comment

  /* block
  comment */

  int a,b,_z;
  float c;

  a=2;
  b=10;
  c=5.3e-3;

  if(a!=2)
  {
        a++;
  }
  else if(b!=10)
  {
        b-=3;
  }
  /* this is
  a /* multi-line
  comment */
}
```

Output:

```
******Tokens*****

int - keyword
main - identifier
( - special character
) - special character
{ - special character
int - keyword
a - identifier
, - special character
```

```
b - identifier
, - special character
_z - identifier
; - special character
float - keyword
c - identifier
; - special character
a - identifier
= - operator
2 - constant
; - special character
b - identifier
= - operator
10 - constant
; - special character
c - identifier
= - operator
5.3e-3 - constant
; - special character
if - keyword
( - special character
a - identifier
!= - operator
2 - constant
) - special character
{ - special character
a - identifier
++ - operator
; - special character
} - special character
else - keyword
if - keyword
( - special character
b - identifier
!= - operator
10 - constant
) - special character
{ - special character
b - identifier
-= - operator
3 - constant
; - special character
} - special character
} - special character

*****Symbol Table******

a    identifier
b    identifier
c    identifier
_z    identifier
main    identifier

*****Constant Table******

2    numeric constant
3    numeric constant
10    numeric constant
5.3e-3    numeric constant
```

Test Case 2:

```
void reduce()
{
   int a=4;
   while(a)
   {
         a--;
   }
}
```

Output:

```
******Tokens*****

void - keyword
reduce - identifier
( - special character
) - special character
{ - special character
int - keyword
a - identifier
= - operator
4 - constant
; - special character
while - keyword
( - special character
a - identifier
) - special character
{ - special character
a - identifier
-- - operator
; - special character
} - special character
} - special character

*****Symbol Table******

a    identifier
reduce    identifier

*****Constant Table******

4    numeric constant
```

Test Case 3

```
float a=$;
```

Output:

```
******Tokens*****

float - keyword
a - identifier
= - operator
ERROR - $ : invalid character at line [1]
; - special character

*****Symbol Table******

a    identifier

*****Constant Table******
```

Test Case 4

```
int main()
{
        int 1a,b,ans;
        1a=2;
        b=3;
        if(a<b)
        {
                ans=b;
        }
        else
        {
                ans=1a;
        }
        return 0;
}
```

Output:

```
******Tokens*****

int - keyword
main - identifier
( - special character
) - special character
{ - special character
int - keyword
ERROR - 1a : ill-formed identifier at line [3]
, - special character
b - identifier
, - special character
ans - identifier
; - special character
ERROR - 1a : ill-formed identifier at line [4]
= - operator
2 - constant
; - special character
b - identifier
= - operator
3 - constant
; - special character
if - keyword
( - special character
a - identifier
< - operator
b - identifier
) - special character
{ - special character
ans - identifier
= - operator
b - identifier
; - special character
} - special character
else - keyword
{ - special character
ans - identifier
= - operator
ERROR - 1a : ill-formed identifier at line [12]
; - special character
} - special character
return - keyword
0 - constant
; - special character
} - special character

*****Symbol Table******

a    identifier
b    identifier
ans    identifier
main    identifier

*****Constant Table******

0    numeric constant
2    numeric constant
3    numeric constant
```

Test Case 5

```
int main()
{
   // this is a comment
   int a=3;
   /* this is also commented
   a+=2;
   return 0;
}
```

Output:

```
******Tokens*****

int - keyword
main - identifier
( - special character
) - special character
```

```
{ - special character
int - keyword
a - identifier
= - operator
3 - constant
; - special character
ERROR : unterminated comment

*****Symbol Table******

a    identifier
main    identifier

*****Constant Table******

3    numeric constant
```
