Compiler Design

Project Report - 1

Lex analyser

By:

15CO248 - Jeshventh Raja T K

15CO229 - Zoeb Mithaiwala

Introduction

What is a compiler?

A compiler is a special program that processes statements written in a high level programming language and turns them into machine language.

The input given to the compiler is a source code written in a high level language. The compiler outputs an equivalent and correct target code in machine level language. The compiler checks and outputs errors, if present. The compiler can check for syntax and semantic errors only.

There are two phases in compiling:

1) Analysis phase (Front end)
2) Synthesis phase (Back end)

Analysis phase

This phase consists of four steps:

1) Lexical analysis - Read and convert characters into words. Done by lexical/linear analyser or scanner
2) Syntax analysis - Convert the words into sentences. Done by syntax analyser or parser
3) Semantic analysis - Understand the meaning of sentences.
4) Intermediate code generation - Convert the understanding into intermediate code

Synthesis phase

This phase consists of two steps:

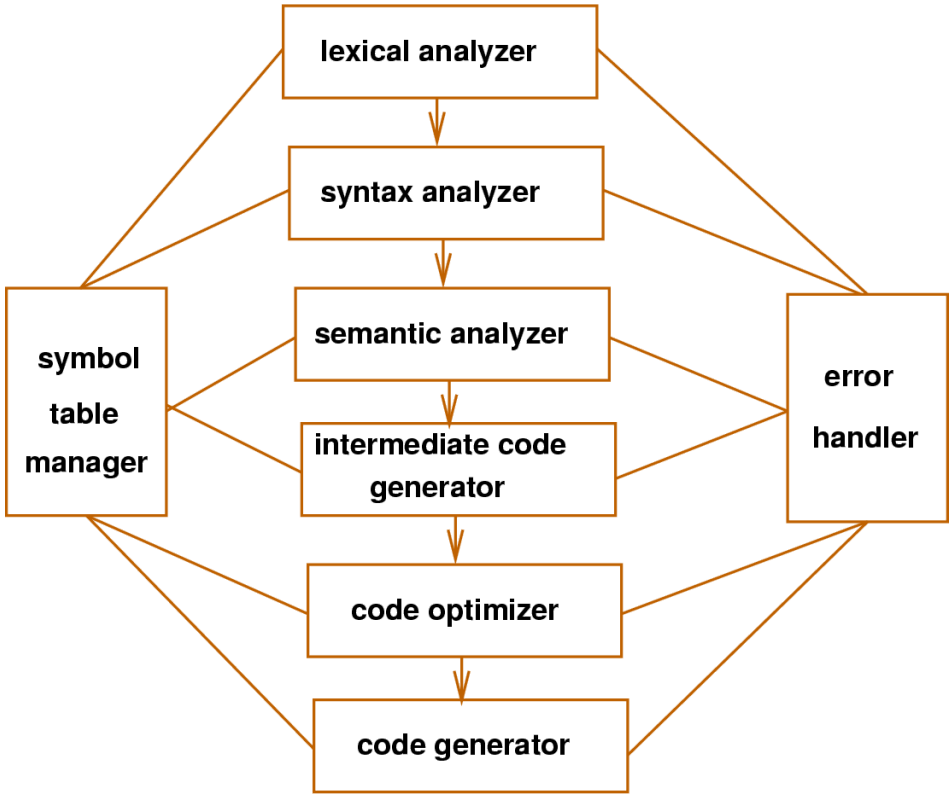
1) Code optimisation - Remove any unwanted or redundant code
2) Code generation - Convert to machine level language

There are various types of compilers:

1) Retargetable compiler - Converts H/L language to M/L language that is understood by a different machine
2) Single pass compiler - It scans an instruction, converts it to machine level code and then goes to the next instruction
3) Multipass compiler - It processes the source code or abstract syntax tree of a program several times, generating an output file after each pass

Additional functionalities of the compiler can be:

1) Error handling
2) Fast compilation
3) Efficient machine level code
4) Fast executing machine level output



Lexical analyser

Lexical analysis is the first phase in a compiler. It converts a sequence of characters into a sequence of tokens.

Token: A sequence of characters that can be treated as a logical unit. Typical tokens are,

1) Identifiers
2) Keywords
3) Operators
4) Special symbols
5) Constants

Lexeme: It is an instance of a token

The input given to the lexical analyser is a source code written in a high level language. It outputs a stream of tokens. It also checks and outputs lexical errors, if present.

The lexical analyser is also called as linear analyser or a scanner.

The functions of a lexical analyser are:

1) Strips out all the unwanted information from the source program(blank space, comments, tabs, newline character, etc)
2) Reads the source program character by character(from left to right, top to bottom)
3) Group the characters based on regular expressions. It decides tokens based on one character look ahead
4) Insert all the symbols in the source program to a symbol table.
5) Give lexical errors

Lexical errors

The various kinds of lexical errors are:

1) **Ill formed tokens** - This happens when no token rule matches.

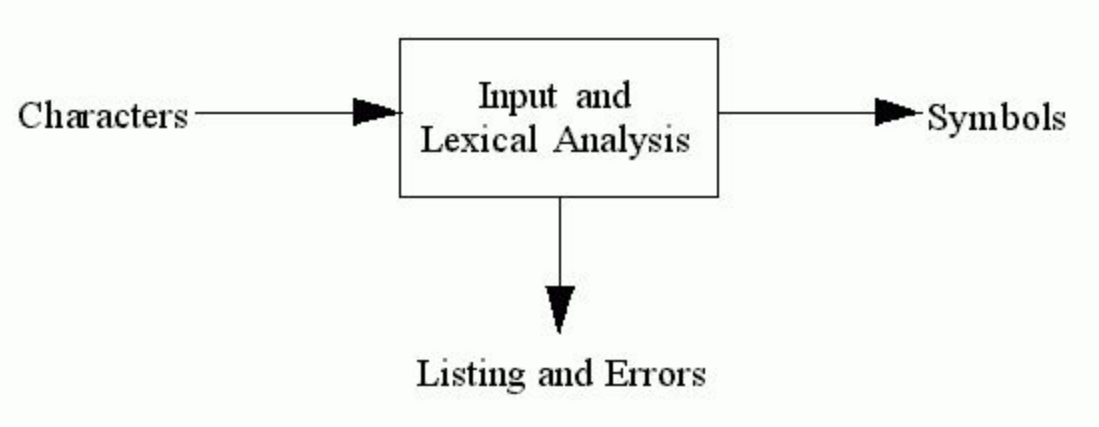
Example: int 9abc; // Here, although 9 is a valid character in identifier, 9abc is an invalid identifier because it starts with a number

2) **Illegal characters** - This happens when illegal characters are user in the source program.

Example: int abc$ // Here, $ is an illegal character

3) **Unclosed multiline comments or strings** - This causes the whole program after the start of the comment or the string to be considered as a comment or string respectively.
4) **Errors due to rule priority** - This happens as per the rule order in the lexical analyser.

Example: 12.345 can be read as (i) a floating point constant or (ii) an int (12) and a floating point constant(.345)



Symbol table after lexical analysis

The symbol table consists of three columns after the lexical analysis phase.

1) Index
2) Lexeme
3) Token type

It is made using a hash table. A hash is calculated on the lexeme and chaining is used to deal with collisions.

Challenges faced

1) **Multiline comments 1:** A common and simple regular expression to match /* and */ for a multiline comment did not work because of the longest matching characteristic of regular expression matching. Given two multiline comments, the code in between the comments as well was commented.

Solution: We wrote a complex regular expression which stops reading at the first match of */

2) **Multiline comment 2:** Writing a regular expression to stop at the first */ was not easy. This was because the not operator(^) did not consider the two characters(*/) together.

Solution: We had to write a complex regular expression which allows a "/" in between but when a "*" occurs, the immediate character must not be a "*" or "/"

3) **Function identification in lexical analysis phase:** We have written a regular expression that can identify a function with no or fixed number of arguments. But the drawback being that the arguments would be identified along with the function name and would not be stored separated in the symbol table. **NOTE:** It is possible to identify the function as an identifier and the arguments as independent tokens by commenting the function's regular expression.

Code

```
%{
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    int size=20;
    int globalIndex=0;
    struct symbolTable{
       int index;
       char *symbol;
       char *attribute;
       struct symbolTable *next;
    };
    struct symbolTable *hash[2][20];
    int i=0;
    void init()
    {
       for(;i<size;i++)
       {
             hash[0][i] = NULL;
             hash[1][i] = NULL;
       }
    }

    int hashLocation(char *sym)
    {
       int sum=0;
       size_t length = strlen(sym);
       int k=0;
       for(;k<length;k++)
       {
             sum = sum + (int)sym[k];
       }
       return sum%size;
    }

    int searchHash(int type, char *sym, int x)
    {
       struct symbolTable *temp = hash[type][x];
       while(temp!=NULL)
```

```
        {
                if (strcmp(temp->symbol, sym)==0)
                {
                        return 0;
                }
                temp=temp->next;
        }
        return 1;
    }

    void addToTable(int type, char *sym, char *attr)
    {
        int x = hashLocation(sym);
        if (searchHash(type, sym, x)==0)
                return;
        struct symbolTable *newSymbol = (struct symbolTable
*)malloc(sizeof(struct symbolTable));
        char *te = (char *)malloc(strlen(sym)+1);
        strcpy(te, sym);
        newSymbol->symbol = te;
        newSymbol->attribute = attr;
        newSymbol->next = NULL;
        newSymbol->index = globalIndex + 1;
        globalIndex++;
        struct symbolTable *temp = hash[type][x];
        hash[type][x] = newSymbol;
        hash[type][x]->next = temp;
    }

    void display()
    {
        int k=0;
        printf("\n\nSYMBOL TABLE:\n");

printf("--------------------------------------------------------------------
\n");
        printf("%*s\t|\t%*s\t|\t%*s\n", 10, "INDEX", 10, "SYMBOL", 10,
"ATTRIBUTE");

printf("--------------------------------------------------------------------
\n");
        for(;k<size;k++)
```

```
	{
		struct symbolTable *temp = hash[0][k];
		while(temp!=NULL)
		{
			printf("%*d\t|\t%*s\t|\t%*s\n",10, temp->index, 10,
temp->symbol, 10, temp->attribute);
			temp = temp->next;
		}
	}

printf("------------------------------------------------------------------
\n");
	k=0;
	printf("\n\nCONSTANT TABLE:\n");

printf("------------------------------------------------------------------
\n");
	printf("%*s\t|\t%*s\t|\t%*s\n", 10, "INDEX", 10, "SYMBOL", 10,
"ATTRIBUTE");

printf("------------------------------------------------------------------
\n");
	for(;k<size;k++)
	{
		struct symbolTable *temp = hash[1][k];
		while(temp!=NULL)
		{
			printf("%*d\t|\t%*s\t|\t%*s\n",10, temp->index, 10,
temp->symbol, 10, temp->attribute);
			temp = temp->next;
		}
	}

printf("------------------------------------------------------------------
\n");
    }
%}

keyword
auto|break|case|char|const|continue|default|do|double|else|enum|extern|floa
t|for|goto|if|int|long|register|return|short|signed|sizeof|static|struct|sw
itch|typedef|union|unsigned|void|volatile|while
```

```
singleLineComment \/\/.*
multilineComment "/*"([^*]|\*+[^*/])*\*+"/"
multilineCommentError "/*"([^*]|\*+[^*/])*
str \"[^\"]*\"
strError \"[^\"]*
letter [a-zA-Z]
digit [0-9]
nonIdentifier ({digit})+(_|{letter})(_|{letter}|{digit})*
identifier (_|{letter})(_|{letter}|{digit})*
dataType int|float|double|short\ int|long\ int
argument {dataType}[\ ]*{identifier}
function {identifier}"("(([\ ]*{argument}[\ ]*\,[\ ]*)*[\ ]*{argument}[\
]*)?")"
integer ({digit})+
float ({digit})*\.({digit})+
assignmentOperator "="
arithmeticOperator "+"|"-"|"*"|"/"|"%"
operator
"+="|"-="|"*="|"/="|"%="|">>="|"<<="|"&="|"^="|"|="|"++"|"--"|"=="|"!="|">"
|"<"|">="|"<="|"||"|"&&"|"!"|"&"|"|"|"^"|"~"|"<<"|">>"|"?"
headerFile "#include"[\ ]*"<"[\ ]*[^>]*[\ ]*">"
specialCharacters \;|\{|\}|\(|\)|\[|\]|,
whitespaces [\ ]|[\n]|[\t]
%%
{multilineCommentError} {printf("ERROR: Multiline comment is not
closed\n");}
{strError} {printf("ERROR: String is not closed\n");}
{singleLineComment} {}
{multilineComment} {}
{headerFile} {printf("Header file found\n");}
{str} {
       char *te = (char *)malloc(strlen(yytext)-1);
       int l = strlen(yytext);
       strncpy(te, yytext + 1, l-2);
       printf("%s %d- String constant\n", te, l);
addToTable(1,te,"String");}
auto {printf("Keyword auto found\n"); addToTable(0,"auto","keyword");}
break {printf("Keyword break found\n"); addToTable(0,"break", "keyword");}
case {printf("Keyword case found\n"); addToTable(0,"case", "keyword");}
char {printf("Keyword char found\n"); addToTable(0,"char", "keyword");}
const {printf("Keyword const found\n"); addToTable(0,"const", "keyword");}
continue {printf("Keyword continue found\n"); addToTable(0,"continue",
```

```
"keyword");}
default {printf("Keyword default found\n"); addToTable(0,"default",
"keyword");}
do {printf("Keyword do found\n"); addToTable(0,"do","keyword");}
double {printf("Keyword double found\n");
addToTable(0,"double","keyword");}
else {printf("Keyword else found\n"); addToTable(0,"else","keyword");}
enum {printf("Keyword enum found\n"); addToTable(0,"enum","keyword");}
extern {printf("Keyword extern found\n");
addToTable(0,"extern","keyword");}
float {printf("Keyword float found\n"); addToTable(0,"float","keyword");}
for {printf("Keyword for found\n"); addToTable(0,"for","keyword");}
goto {printf("Keyword goto found\n"); addToTable(0,"goto","keyword");}
if {printf("Keyword if found\n"); addToTable(0,"if","keyword");}
int {printf("Keyword int found\n"); addToTable(0,"int","keyword");}
long {printf("Keyword long found\n"); addToTable(0,"long","keyword");}
register {printf("Keyword register found\n");
addToTable(0,"register","keyword");}
return {printf("Keyword return found\n");
addToTable(0,"return","keyword");}
short {printf("Keyword short found\n"); addToTable(0,"short","keyword");}
signed {printf("Keyword signed found\n");
addToTable(0,"signed","keyword");}
sizeof {printf("Keyword sizeof found\n");
addToTable(0,"sizeof","keyword");}
static {printf("Keyword static found\n");
addToTable(0,"static","keyword");}
struct {printf("Keyword struct found\n");
addToTable(0,"struct","keyword");}
switch {printf("Keyword switch found\n");
addToTable(0,"switch","keyword");}
typedef {printf("Keyword typedef found\n");
addToTable(0,"typedef","keyword");}
union {printf("Keyword union found\n"); addToTable(0,"union","keyword");}
unsigned {printf("Keyword unsigned found\n");
addToTable(0,"unsigned","keyword");}
void {printf("Keyword void found\n"); addToTable(0,"void","keyword");}
volatile {printf("Keyword volatile found\n");
addToTable(0,"volatile","keyword");}
while {printf("Keyword while found\n"); addToTable(0,"while","keyword");}
{keyword} {printf("%s is a Keyword\n",yytext);}
{function} {printf("%s is a Function\n",yytext);
```

```
addToTable(0,yytext,"function");}
{identifier} {printf("%s is a Identifier\n", yytext);
addToTable(0,yytext,"identifier");}
{integer} {printf("%s is an Integer constant\n", yytext);
addToTable(1,yytext,"integer");}
{float} {printf("%s is a floating point constant\n", yytext);
addToTable(1,yytext,"float");}
{nonIdentifier} {printf("ERROR: %s is an ill formed token\n", yytext);}
{operator} {printf("%s is an operator\n", yytext);
addToTable(0,yytext,"operator");}
{assignmentOperator} {printf("%s is an assignment operator\n", yytext);
addToTable(0,yytext,"assignment operator");}
{arithmeticOperator} {printf("%s is an arithmetic operator\n", yytext);
addToTable(0,yytext,"arithmetic operator");}
{specialCharacters} {printf("%s is a special character\n",yytext);}
{whitespaces} {}
. {printf("Error: %s is an Illegal characters\n",yytext);}
%%
int main()
{
    init();
    yyin=fopen("testcases/test4.c","r");
    yylex();
    display();
}
int yywrap()
{
    return(1);
}
```

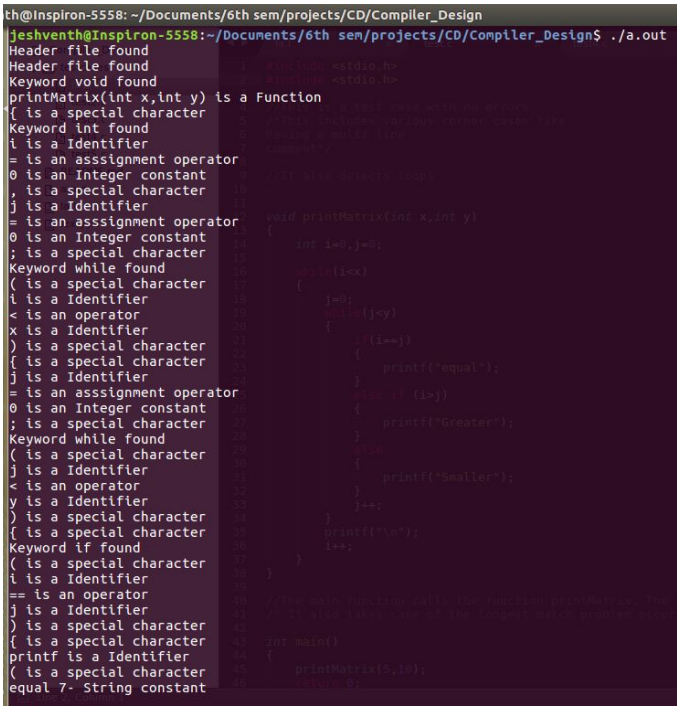
Output and screenshots

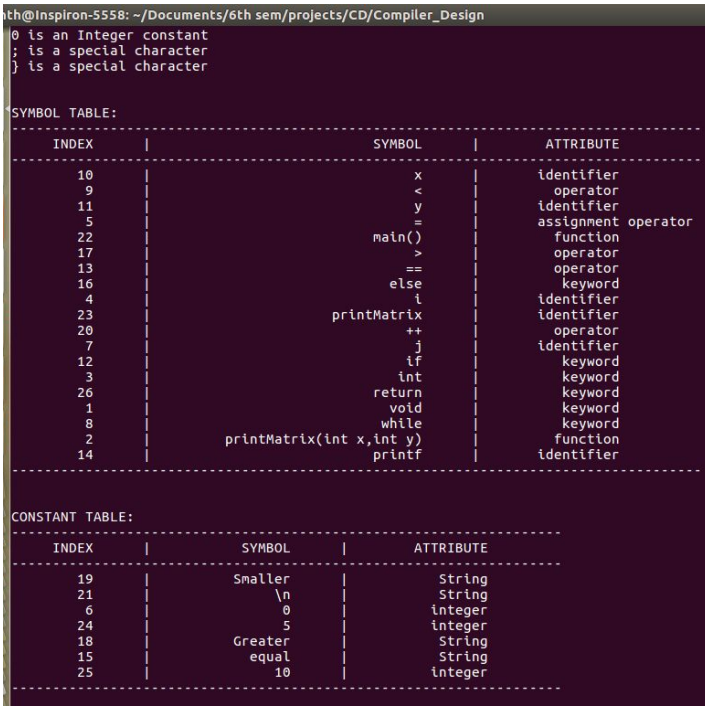
Our program identifies various types of tokens and displays them. Initially, it prints all the tokens as it encounters them. In the end, every token is displayed in symbol and constants table. We have made various test cases depicting various functionalities of our compiler.

Test case 1

This test case doesn't have any lexical errors. This test case covers:

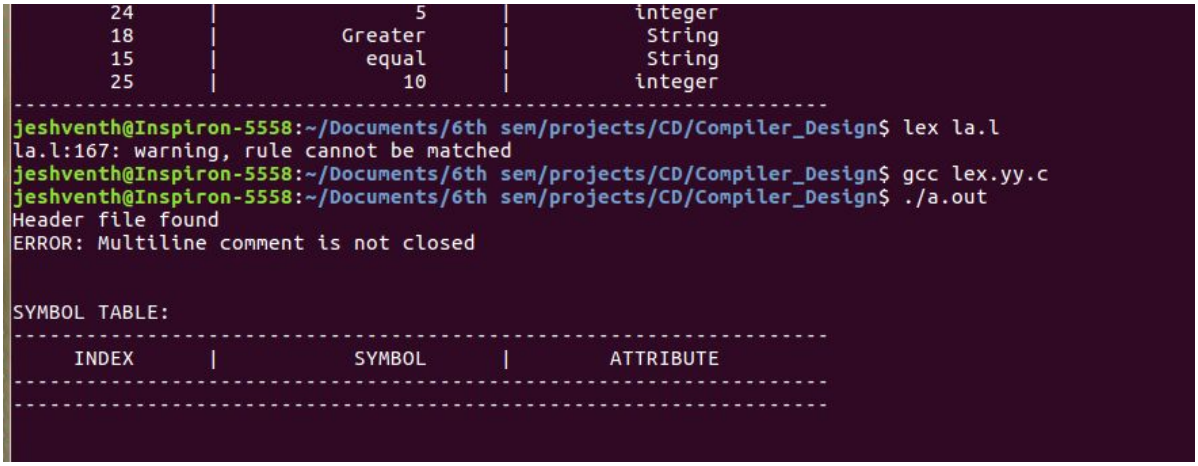
1) Header files
2) Multiline and single line comments(Takes care of longest match problem as well)
3) Functions
4) Loops
5) Identifiers
6) Constants
7) Special characters





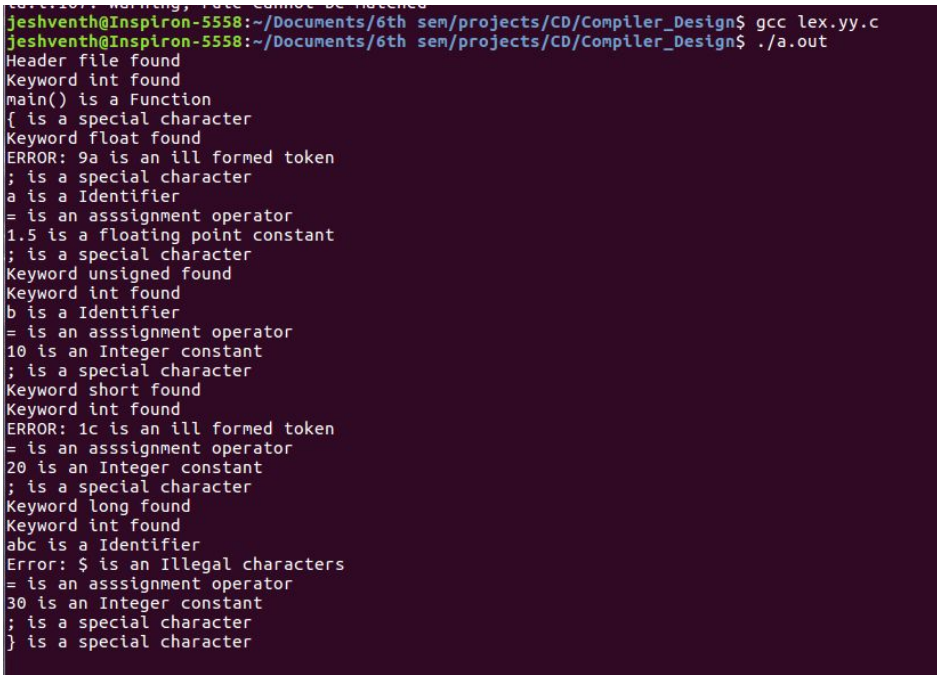
Test case 2

This test case contains a lex error. The multiline comment is not closed. Hence it prints a error message telling what the problem is.



Test case 3

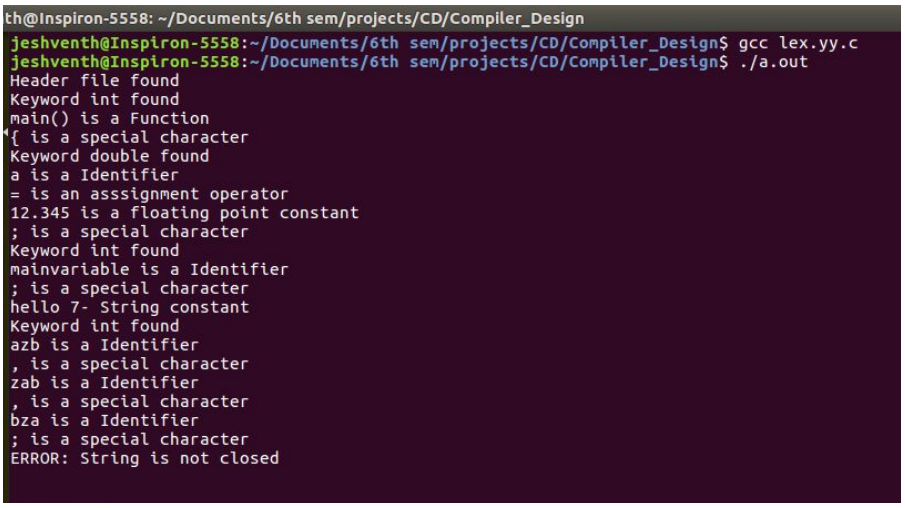
This test case contains two lexical errors. Ill formed tokens and illegal character error.

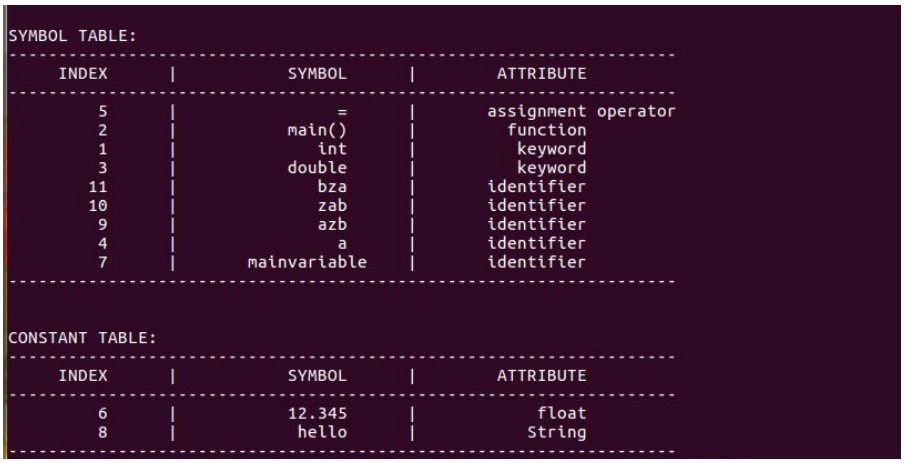


Test case 4

This test case shows how rule priority avoids errors.

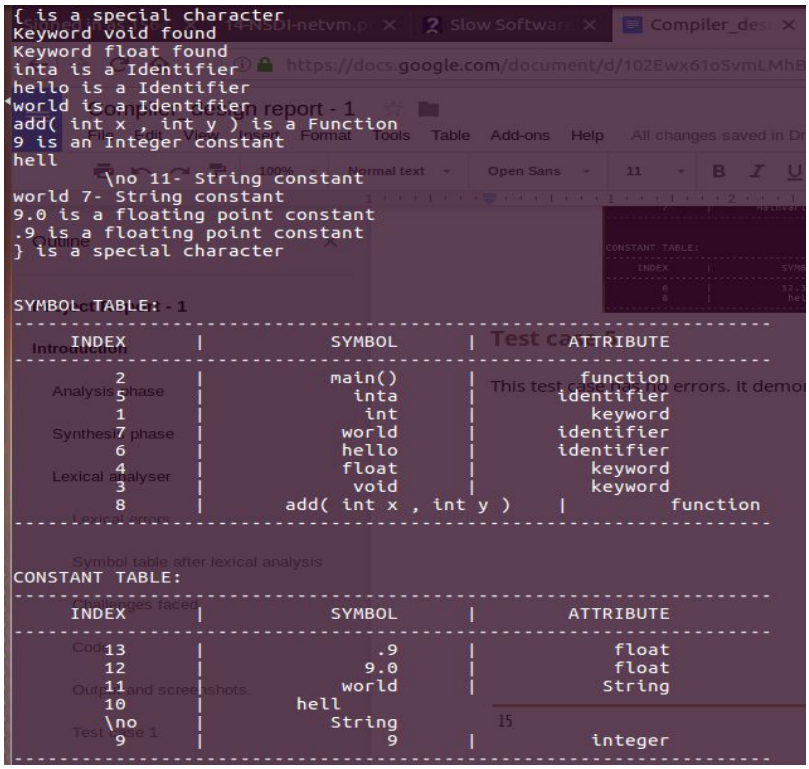
1) mainvariable is read a single identifier and not as two tokens main and variable
2) Similarly, 12.345 is read as a floating point constant and not as an integer(12) and a floating point constant(.345)





Test case 5

This test case has no errors. It demonstrates the correct identification of constants.



```
{ is a special character
Keyword void found
Keyword float found
inta is a Identifier
hello is a Identifier
world is a Identifier
add( int x , int y ) is a Function
9 is an Integer constant
hell
        \no 11- String constant
world 7- String constant
9.0 is a floating point constant
.9 is a floating point constant
} is a special character


SYMBOL TABLE:
--------------------------------------------------------------------------
      INDEX       |            SYMBOL       |          ATTRIBUTE
--------------------------------------------------------------------------
          2       |            main()       |            function
          5       |              inta       |          identifier
          1       |               int       |             keyword
          7       |             world       |          identifier
          6       |             hello       |          identifier
          4       |             float       |             keyword
          3       |              void       |             keyword
          8       |       add( int x , int y )    |          function
--------------------------------------------------------------------------


CONSTANT TABLE:
--------------------------------------------------------------------------
     INDEX        |            SYMBOL       |          ATTRIBUTE
--------------------------------------------------------------------------
         13       |                .9       |              float
         12       |               9.0       |              float
         11       |             world       |             String
         10       |          hell           |
          \no     |            String       |
          9       |                 9       |            integer
--------------------------------------------------------------------------
```