DESIGNING A SCANNER FOR C LANGUAGE

A MINI PROJECT REPORT

Submitted by

RAKSHIT AGARWAL(RA2011026010340)
TAPNANSHU ATHARVA(RA2011026010309)

Under the guidance of

Dr. Maheshwari. A
(Assistant Professor, Dept Of Computational Intelligence)

In partial satisfaction of the requirements for the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING

SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR - 603203

APRIL 2023

COLLEGE OF ENGINEERING & TECHNOLOGY
SRM INSTITUTE OF SCIENCE & TECHNOLOGY
S.R.M. NAGAR, KATTANKULATHUR – 603 203

BONAFIDE CERTIFICATE

Certified that this project report **" Designing a scanner for c language "** is the bonafide work of **" Rakshit Agarwal(RA2011026010340) , Tapnanshu Atharva(RA2011026010309) ",** of III Year/VI Sem B.tech(CSE) who carried out the mini project work under my supervision for the course 18CSC304J- Compiler Design in SRM Institute of Science and Technology during the academic year 2022-2023(Even semester).

SIGNATURE

Dr. Maheshwari. A

Assistant Professor

Department of computational intelligence

SIGNATURE

Dr. R. Annie Uthra

Professor and Head

Department of computational intelligence

INDEX

Introduction

Compiler

A compiler is a program that can read a program in one language - the source language - and translate it to an equivalent program in another language - the target language. An important role of the compiler is to detect any errors in the source program during the translation process.

Aim

The aim of designing a scanner for C language is to develop a program that can efficiently and accurately analyse C language source code and break it down into individual tokens or lexemes, which can then be further processed by the compiler. The scanner should be able to identify keywords, identifiers, constants, operators, and other language constructs, and generate a corresponding stream of tokens that can be fed into the compiler for syntax and semantic analysis.
The ultimate goal of the scanner is to facilitate the creation of robust and error-free C programs by providing a reliable means of parsing and interpreting the source code.

ABSTRACT

The process of developing software requires a deep understanding of the programming languages that are used to create it. One important aspect of this is the ability to analyze and parse the language's syntax. In the case of the C programming language, this requires designing a scanner that can break down source code into individual tokens, such as keywords, identifiers, constants, and operators. This abstract outlines the aim of designing a scanner for C language, which is to create a program that can accurately and efficiently parse C code and generate a stream of tokens that can be used by the compiler. The scanner should be able to identify various language constructs and facilitate the creation of robust and error-free C programs. The development of such a scanner is essential for creating high-quality software in C and is an important step towards improving the overall quality of code in the programming community..

Problem statement

The C programming language is widely used for system programming, application development, and embedded systems, among other areas. However, analyzing and parsing C code can be a challenging task due to its complex syntax and grammar. The problem is that existing C language scanners may not be efficient or accurate enough to handle the increasing complexity of modern software applications. As a result, there is a need to design a new scanner for C language that can more effectively analyze C code and generate an accurate stream of tokens. This scanner should be able to identify keywords, identifiers, constants, operators, and other language constructs in a way that is reliable, efficient, and consistent with the C language specification. The aim of this project is to design a scanner that can meet these requirements and facilitate the creation of high-quality C programs.

Requirements to run the script

To run a script for designing a scanner for C language, you need to have the following requirements:

1. **Operating System:** The script can be run on any operating system that supports the programming language and dependencies required by the scanner. Popular operating systems include Windows, macOS, and Linux.
2. **Programming Language:** The scanner script needs to be written in a suitable programming language, such as C, C++, Java, or Python, which is supported by the operating system.
3. **Compiler:** The script should be compatible with standard C compilers, including open-source and commercial compilers. The choice of compiler can impact the efficiency and accuracy of the scanner, as different compilers may handle certain C constructs differently.
4. **Dependencies:** The script may have dependencies on third-party libraries, which need to be installed on the system before running the script. Common dependencies for designing a scanner include lex and yacc, which are used for generating the scanner code.
5. **Input/Output:** The script requires an input file containing the C code to be scanned and an output file to which the scanner generates a stream of tokens. The input file can be in any format supported by the programming language used, such as a text file or a string variable in the code.
6. **Documentation:** The scanner script should be accompanied by comprehensive documentation, including user manuals,

reference materials, and code comments, to facilitate its use and maintenance.

Meeting these requirements will ensure that the scanner script can be run efficiently and accurately, and that it can generate an accurate stream of tokens for the given C code.

Software Requirement

Designing a scanner for C language requires several software requirements to ensure that the scanner can efficiently analyze C code and generate an accurate stream of tokens. Here are some of the key software requirements:

1. Programming Language: The scanner needs to be developed using a suitable programming language that is capable of efficiently implementing the scanning algorithm. Common choices for implementing scanners include C, C++, Java, and Python.
2. Development Environment: The scanner development requires a suitable development environment, such as an integrated development environment (IDE) or a text editor. Popular IDEs for developing C programs include Visual Studio Code, Eclipse, and Xcode.
3. Compiler: The scanner must be compatible with standard C compilers, including open-source and commercial compilers. The choice of compiler can impact the efficiency and accuracy of the scanner, as different compilers may handle certain C constructs differently.

4. Operating System: The scanner must be compatible with the operating system on which it is running, including Windows, macOS, Linux, and other Unix-like operating systems.
5. Third-Party Libraries: The scanner may rely on third-party libraries to implement certain features, such as regular expressions, file input/output, or memory management. The choice of libraries can impact the efficiency and portability of the scanner.
6. Documentation: The scanner must be accompanied by comprehensive documentation, including user manuals, reference materials, and code comments, to facilitate its use and maintenance.
Meeting these software requirements will ensure that the scanner is reliable, efficient, and easy to use, and that it can be integrated effectively into the C programming workflow

Hardware requirement

Designing a scanner for C language does not have any specific hardware requirements, as it can run on a wide range of hardware platforms, including personal computers, servers, and embedded systems. However, the efficiency and performance of the scanner can be influenced by the hardware on which it is running. Here are some hardware considerations that can impact the performance of the scanner:
1. CPU: The scanner requires a CPU that is capable of executing the scanning algorithm efficiently. Faster CPUs with more cores will generally provide better performance, especially for large codebases.

2. Memory: The scanner needs sufficient memory to store the program's source code and the token stream generated by the scanner. The amount of memory required depends on the size of the program being analyzed.
3. Disk Space: The scanner requires disk space to store the program's source code, intermediate files, and output files generated by the compiler.
4. Input/Output Devices: The scanner requires input/output devices, such as keyboards, mice, and monitors, to allow users to input the source code and view the scanner's output.

Overall, the hardware requirements for designing a scanner for C language are relatively modest, and the scanner can run on a wide range of hardware platforms, from low-powered embedded systems to high-performance server cluster

Lexical Analysis

As the first phase of a compiler, 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 lexical analyzer maintains a data structure called as the symbol table. When the lexical analyzer discovers a lexeme constituting an identifier, it enters that lexeme into the symbol table.

The lexical analyzer performs certain other tasks besides identification of lexemes. One such task is stripping out comments and whitespace. Another task is correlating error messages generated by the compiler with the source program

Designing a scanner for C language

We have used Flex to perform lexical analysis on a subset of the C programming language. Flex is a lexical analyzer generator that

takes in a set of descriptions of possible tokens and produces a C file that performs lexical analysis and identifies the tokens. Here we describe the functionality and construction of our scanner.

This document is divided into the following sections:

- **Functionality:** Contains a description of our Flex program and the variety of tokens that it can identify and the error handling strategies.
- **Symbol table and Constants table:** Contains an overview of the architecture of the symbol and constants table which contain descriptions of the lexemes identified during lexical analysis.
- **Code Organisation:** Contains a description of the files used for lexical analysis
- **Source Code:** Contains the source code used for lexical analysis

Functionality

Below is a list containing the different tokens that are identified by our Flex program. It also gives a detailed description of how the different tokens are identified and how errors are detected if any.

Keywords

The keywords identified are: **`int, long, short, long long, signed, unsigned, for, break, continue, if, else, return.`**

Identifiers

- Identifiers are identified and added to the symbol table. The rule followed is represented by the regular expression **`(_|{letter})({letter}|{digit}|_){0,31}`**.

- The rule only identifies those lexemes as identifiers which either begin with a letter or an underscore and is followed by either a letter, digit or an underscore with a maximum length of 32.
- The first part of the regular expression **(_|{letter})** ensures that the identifiers begin with an underscore or a letter and the second part**({letter}|{digit}|_){0,31}** matches a combination of letters, digits and underscore and ensures that the maximum length does not exceed 32. The definitions of **{letter}** and **{digit}** can be seen in the code at the end.
- Any identifier that begins with a digit is marked as a lexical error and the same is displayed on the output. The regex used for this is **{digit}+({letter}|_)+**

Comments

Single and multi line comments are identified. Single line comments are identified by **//.*** regular expression. The multiline regex is identified as follows:

- We make use of an exclusive state called **<CMNT>**. When a **/*** pattern is found, we note down the line number and enter the **<CMNT>** exclusive state. When the ***/** is found, we go back to the **INITIAL** state, the default state in Flex, signifying the end of the comment.
- Then we define patterns that are to be matched only when the lexer is in the **<CMNT>** state. Since, it is an exclusive state, only the patterns that are defined for this state (the ones prepended with **<CMNT>** in the lex file are matched, rest of the patterns are inactive.

- We also identify nested comments. If we find another **/*** while still in the **<CMNT>** state, we print an error message saying that nested comments are invalid.
- If the comment does not terminate until **EOF**, and error message is displayed along with the line number where the comment begins. This is implemented by checking if the lexer matches **<<EOF>>** pattern while still in the **<CMNT>** state, which means that a ***/** has not been found until the end of file and therefore the comment has not been terminated.

Strings

The lexer can identify strings in any C program. It can also handle double quotes that are escaped using a **** inside a string. Further, error messages are displayed for unterminated strings. We use the following strategy.

- We first match patterns that are within double quotes.
- But if the string is something like **"This is \" a string",** it will only match **"This is \"**. So as soon as a match is found we first check if the last double quote is escaped using a backslash.
- If the last quote is not escaped with a backslash we have found the string we are looking for and we add it to the constants table.
- But in case the last double quote is escaped with a backslash we push the last double quote back for scanning. This can be achieved in lex using the command **yyless(yyleng - 1).**
- **yyless(n)** [1] tells lex to "push back" all but the first n characters of the matched token. **yyleng** [1] holds the length of the matched token.

- And hence **`yyless(yyleng -1)`** will push back the last character i.e the double quote back for scanning and lex will continue scanning from **" is a string"**.
- We use another built-in lex function called **`yymore()`** [1] which tells lex to append the next matched token to the currently matched one.
- Now the lexer continues and matches **"is a string"** and since we had called **`yymore()`** earlier it appends it to the earlier token **"This is ** giving us the entire string **"This is \" a string"**. Notice that since we had called **`yyless(yyleng - 1)`** the last double quote is left out from the first matched token giving us the entire string as required. The following lines of code accomplish the above described process.

```
\"[^\"\n]*\" {

  if(yytext[yyleng-2]=='\\') /* check if it was an escaped quote */
  {
        yyless(yyleng-1);/* push the quote back if it was escaped */ yymore();
         /* Append next token to this one */
  } else{ insert( constant_table, yytext,
 STRING); }
```

We use the regular expression **`\"[^\"\n]*$`** to check for strings that don't terminate. This regular expression checks for a sequence of a double quote followed by zero or more occurrences of characters excluding double quotes and new line and this sequence should not have a close quote. This is specified by the **`$`** character which tests for the end of line. Thus, the regular expression checks for strings that do not terminate till end of line and it prints an error message on the screen.

Integer Constants

- The Flex program can identify two types of numeric constants: decimal and hexadecimal. The regular expressions for these are **`[+-]?{digit}+[lLuU]?`** and **`[+-]?0[xX]{hex}+[lLuU]?`** respectively.
- The sign is considered as optional and a constant without a sign is by default positive.
  All hexadecimal constants should begin with **`0x`** or **`0X`**.
- The definition of **`{digit}`** is all the decimal digits **`0-9`**. The definition of **`{hex}`** consists of the hexadecimal digits **`0-9`** and characters **`a-f`**.
- Some constants which are assigned to long or unsigned variables may suffix **`l`** or **`L`** and **`u`** or **`U`** or a combination of these characters with the constant. All of these conditions are taken care of by the regular expression.

Preprocessor Directives

The filenames that come after the **`#include`** are selectively identified through the exclusive state **`<PREPROC>`** since the regular expressions for treating the filenames must be kept different from other regexes.

Upon encountering a **`#include`** at the beginning of a line, the lexer switches to the state **`<PREPROC>`** where it can tokenize filenames of the form **`“stdio.h”`** or **`<stdio.h>`**. Filenames of any other format are considered as illegal and an error message regarding the same is printed.

Symbol Table & Constants table

We implement a generic hash table with chaining than can be used to declare both a symbol table and a constant table. Every entry in the hash table is a struct of the following form.

```
/* struct to hold each entry */
struct entry_s
{ char* lexeme; int token_name;
    struct entry_s* successor;
}; typedef struct entry_s

entry_t;
```

The struct consists of a character pointer to the lexeme that is matched by the lexer, an integer token that is associated with each type of token as defined in "**tokens.h**" and a pointer to the next node in the case of chaining in the hash table.

A symbol table or a constant table can be created using the **`create_table()`** function. The function returns a pointer to a new created hash table which is basically an array of pointers of the type **`entry_t*`**. This is achieved by the following lines:

```
/* declare pointers and assign hash tables */
entry_t**       symbol_table;       entry_t**
constant_table; symbol_table = create_table();
constant_table = create_table();
```

Every time the lexer matches a pattern, the text that matches the pattern (lexeme) is entered into the associated hash table using an **`insert()`** function. There are two hash tables maintained: the symbol table and the constants table. Depending on whether the lexeme is a constant or a symbol, an appropriate parameter is passed to the insert function. For example, **`insert(symbol_table, yytext, INT)`** inserts the keyword **`INT`** into the symbol table and

`insert(constant_table, yytext, HEX_CONSTANT)` inserts a hexadecimal constant into the constants table. The values associated with **`INT, HEX_CONSTANT`** and other tokens are defined in the tokens.h file.

A hash is generated using the matched pattern string as input. We use the Jenkins hash function[2]. The hash table has a fixed size as defined by the user using **`HASH_TABLE_SIZE`**. The generated hash value is mapped to a value in the range **`[0, HASH_TABLE_SIZE)`** through the operation **`hash_value % HASH_TABLE_SIZE`**. This is the index in the hash table for this particular entry. In case the indices clash, a linked list is created and the multiple clashing entries are chained together at that index.

Source Code

lexer.l

```
%{

#include <stdlib.h>
#include <stdio.h>
#include "symboltable.h"
#include "tokens.h"

entry_t** symbol_table;
entry_t** constant_table;
int cmnt_strt = 0;

%}

letter  [a-zA-Z]  digit  [0-9]  ws  [  \t\r\f\v]+
identifier  (_|{letter})({letter}|{digit}|_){0,31}
hex [0-9a-f]

 /* Exclusive states */
%x CMNT
%x PREPROC

%%
  /* Keywords*/
"int"                                {printf("\t%-30s : %3d\n",yytext,INT);}
"long"                               {printf("\t%-30s : %3d\n",yytext,LONG);}
"long long"                            {printf("\t%-30s : %3d\n",yytext,LONG_LONG);}
"short"                              {printf("\t%-30s : %3d\n",yytext,SHORT);}
"signed"                             {printf("\t%-30s : %3d\n",yytext,SIGNED);}
"unsigned"                             {printf("\t%-30s : %3d\n",yytext,UNSIGNED);}
"for"                                {printf("\t%-30s : %3d\n",yytext,FOR);}
"break"                              {printf("\t%-30s : %3d\n",yytext,BREAK);}
"continue"                             {printf("\t%-30s : %3d\n",yytext,CONTINUE);}
"if"                                 {printf("\t%-30s : %3d\n",yytext,IF);}
"else"                               {printf("\t%-30s : %3d\n",yytext,ELSE);}
"return"                             {printf("\t%-30s : %3d\n",yytext,RETURN);}

{identifier}                          {printf("\t%-30s : %3d\n", yytext,IDENTIFIER);
                                     insert( symbol_table,yytext,IDENTIFIER );}
{ws}                                 ;
```

```
[+\-]?[0][x|X]{hex}+[lLuU]?   {printf("\t%-30s   :   %3d\n",   yytext,HEX_CONSTANT);   insert(
                                    constant_table,yytext,HEX_CONSTANT);}
[+\-]?{digit}+[lLuU]?               {printf("\t%-30s : %3d\n", yytext,DEC_CONSTANT); insert(
                                    constant_table,yytext,DEC_CONSTANT);}
"/*"                                 {cmnt_strt = yylineno; BEGIN CMNT;}
<CMNT>.|{ws}                        ;
<CMNT>\n                            {yylineno++;}
<CMNT>"*/"                          {BEGIN INITIAL;}
<CMNT>"/*"      {printf("Line %3d: Nested comments are not valid!\n",yylineno);}
<CMNT><<EOF>>   {printf("Line %3d: Unterminated comment\n", cmnt_strt); yyterminate();}
^"#include"                         {BEGIN PREPROC;}
<PREPROC>"<"[^>\n]+">"               {printf("\t%-30s : %3d\n",yytext,HEADER_FILE);}
<PREPROC>{ws}                       ;
<PREPROC>\"[^"\n]+\"                 {printf("\t%-30s : %3d\n",yytext,HEADER_FILE);}
<PREPROC>\n                         {yylineno++; BEGIN INITIAL;}
<PREPROC>.                          {printf("Line %3d: Illegal header file format
\n",yylineno); yyterminate();}
"//".*                              ;

\"[^\"\n]*\" {

  if(yytext[yyleng-2]=='\\') /* check if it was an escaped quote */
  {
       yyless(yyleng-1);/* push the quote back if it was escaped */ yymore();
  }
  else
  insert( constant_table,yytext,STRING); }

\"[^\"\n]*$                     {printf("Line %3d: Unterminated string %s\n",yylineno,yytext);}
{digit}+({letter}|_)+           {printf("Line %3d: Illegal identifier name
%s\n",yylineno,yytext);}
\n                              {yylineno++;}
"--"                            {printf("\t%-30s : %3d\n",yytext,DECREMENT);}
"++"                            {printf("\t%-30s : %3d\n",yytext,INCREMENT);}
"->"                            {printf("\t%-30s : %3d\n",yytext,PTR_SELECT);}
"&&"                            {printf("\t%-30s : %3d\n",yytext,LOGICAL_AND);}
"||"                            {printf("\t%-30s : %3d\n",yytext,LOGICAL_OR);}
"<="                            {printf("\t%-30s : %3d\n",yytext,LS_THAN_EQ);}
">="                            {printf("\t%-30s : %3d\n",yytext,GR_THAN_EQ);}
"=="                            {printf("\t%-30s : %3d\n",yytext,EQ);}
"!="                            {printf("\t%-30s : %3d\n",yytext,NOT_EQ);}
";"                            {printf("\t%-30s : %3d\n",yytext,DELIMITER);}
"{"                             {printf("\t%-30s : %3d\n",yytext,OPEN_BRACES);}
"}"                             {printf("\t%-30s : %3d\n",yytext,CLOSE_BRACES);}
","                            {printf("\t%-30s : %3d\n",yytext,COMMA);}
```

```
"="                             {printf("\t%-30s : %3d\n",yytext,ASSIGN);}
"("                             {printf("\t%-30s : %3d\n",yytext,OPEN_PAR);}
")"                             {printf("\t%-30s : %3d\n",yytext,CLOSE_PAR);}
"["                              {printf("\t%-30s : %3d\n",yytext,OPEN_SQ_BRKT);}
"]"                              {printf("\t%-30s : %3d\n",yytext,CLOSE_SQ_BRKT);}
"-"                             {printf("\t%-30s : %3d\n",yytext,MINUS);}
"+"                             {printf("\t%-30s : %3d\n",yytext,PLUS);}
"*"                             {printf("\t%-30s : %3d\n",yytext,STAR);}
"/"                              {printf("\t%-30s : %3d\n",yytext,FW_SLASH);}
"%"                             {printf("\t%-30s : %3d\n",yytext,MODULO);}
"<"                             {printf("\t%-30s : %3d\n",yytext,LS_THAN);}
">"                             {printf("\t%-30s : %3d\n",yytext,GR_THAN);}
.                                  {printf("Line %3d: Illegal character %s\n",yylineno,yytext);}

%%

int main()
{  yyin=fopen("testcases/test-case-4.c","r");
  Symbol_table     =     create_table();
  Constant_table    =    create_table();
  yylex();            printf("\n\tSymbol
  table\n");     display(symbol_table);
  printf("\n\n\tConstants    Table\n");
  display(constant_table);
}
```

symboltable.h

```
/*
*               Compiler Design Project 1 : Lexical Analyser *
*               File      : symboltable.h
*               Description : This file contains functions related to a hash organised symbol
                table.
*               The functions implemented are:
*               create_table(), insert(), search, display()
*
*/
#include <stdint.h>
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include <string.h>

#define HASH_TABLE_SIZE 100

/* struct to hold each entry */
struct entry_s
{ char* lexeme; int token_name;
    struct entry_s* successor;
}; typedef struct entry_s

entry_t;

/* Create a new hash_table. */ entry_t**
create_table()
{ entry_t** hash_table_ptr = NULL; // declare a pointer

    /* Allocate memory for a hashtable array of size HASH_TABLE_SIZE */
    if( ( hash_table_ptr = malloc( sizeof( entry_t* ) * HASH_TABLE_SIZE ) ) == NULL )
    return NULL; int i;

    // Initialise all entries as NULL for( i =
       0; i < HASH_TABLE_SIZE; i++ )
    { hash_table_ptr[i] = NULL;
    }

    return hash_table_ptr;
}

/* Generate hash from a string. Then generate an index in [0, HASH_TABLE_SIZE) */
uint32_t hash( char *lexeme )
{     size_t     i;
    uint32_t hash;

    /* Apply Jenkins hash function
     *   https://en.wikipedia.org/wiki/Jenkins_hash_function#one-at-a-time
     */
    for ( hash = i = 0; i < strlen(lexeme); ++i )
       {hash += lexeme[i];
       hash += ( hash << 10 );
```

```
        hash ^= ( hash >> 6 );
        }
    hash += ( hash << 3 );
    hash ^= ( hash >> 11 );
    hash += ( hash << 15 );

    return hash % HASH_TABLE_SIZE; // return an index in [0, HASH_TABLE_SIZE)
}

/* Create an entry for a lexeme, token pair. This will be called from the insert function */
entry_t *create_entry( char *lexeme, int token_name )
{ entry_t *newentry;

    /* Allocate space for newentry */ if( ( newentry =
    malloc( sizeof( entry_t ) ) ) == NULL ) {return NULL;
    }
    /* Copy lexeme to newentry location using strdup (string-duplicate). Return NULL if it
fails */ if( ( newentry->lexeme = strdup( lexeme ) ) == NULL ) {return NULL;
    }

    newentry->token_name = token_name; newentry->successor
    = NULL;

    return newentry;
}

/* Search for an entry given a lexeme. Return a pointer to the entry of the lexeme exists,
else return NULL */ entry_t* search( entry_t** hash_table_ptr, char* lexeme )
{  uint32_t  idx  =  0;
    entry_t* myentry;

        // get the index of this lexeme as per the hash function idx
    = hash( lexeme );

    /* Traverse the linked list at this idx and see if lexeme exists */ myentry
    = hash_table_ptr[idx];

    while( myentry != NULL && strcmp( lexeme, myentry->lexeme ) != 0 )
    { myentry = myentry->successor;
    } if(myentry == NULL) // lexeme is not
    found
```

```
        return NULL;

    else // lexeme found return
        myentry;

}

/* Insert an entry into a hash table. */ void insert( entry_t**
hash_table_ptr, char* lexeme, int token_name )
{ if( search( hash_table_ptr, lexeme ) != NULL) // If lexeme already exists, don't insert,
return return;

    uint32_t idx; entry_t*
    newentry = NULL; entry_t*
    head = NULL;

    idx = hash( lexeme ); // Get the index for this lexeme based on the hash function newentry
    = create_entry( lexeme, token_name ); // Create an entry using the <lexeme,
token> pair

    if(newentry == NULL) // In case there was some error while executing create_entry()
    { printf("Insert failed. New entry could not be created.");
        exit(1);
    } head = hash_table_ptr[idx]; // get the head entry at this

    index

    if(head == NULL) // This is the first lexeme that matches this hash index
    { hash_table_ptr[idx] = newentry;
    }
    else // if not, add this entry to the head
    { newentry->successor = hash_table_ptr[idx];
    hash_table_ptr[idx] = newentry; }
}

// Traverse the hash table and print all the entries void
display(entry_t** hash_table_ptr)
{ int i; entry_t* traverser;
    printf("\n=========================================\n");
    printf("\t < lexeme , token >\n");
    printf("=========================================\n");
```

```
    for( i=0; i < HASH_TABLE_SIZE; i++)
    { traverser = hash_table_ptr[i];

        while( traverser != NULL)
        { printf("< %-30s, %3d >\n", traverser->lexeme, traverser->token_name); traverser =
        traverser->successor; }
    }  printf("=========================================\n");  printf("NOTE:
Please refer tokens.h for token meanings\n"); }
```

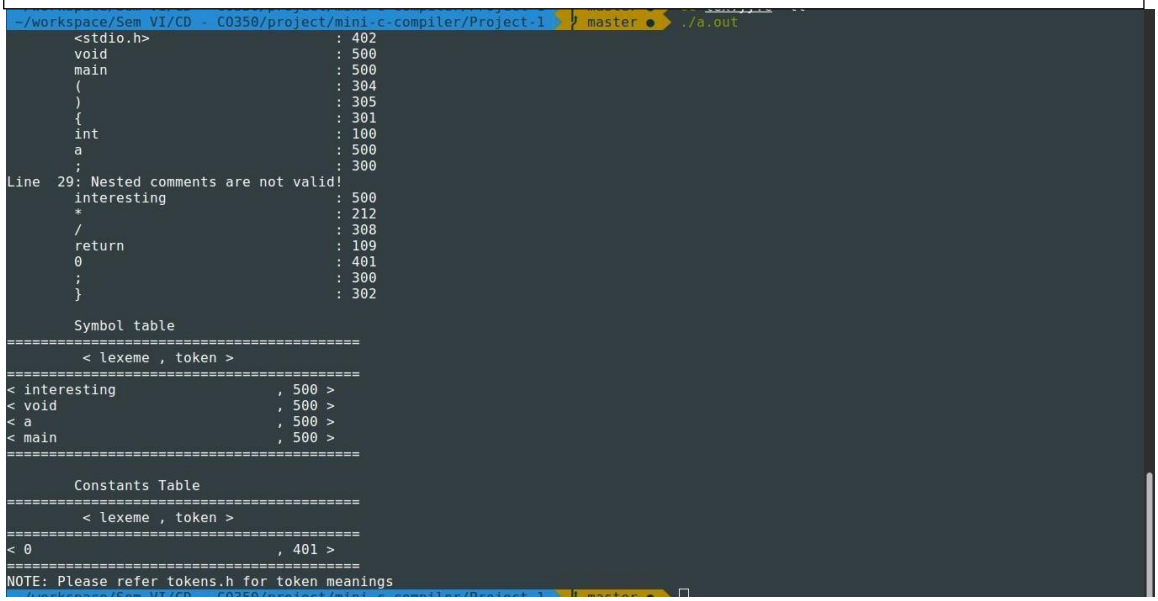
tokens.h

```
/*
* Compiler Design Project 1 : Lexical Analyser *
* File  : tokens.h
* Description : This file defines tokens and the values associated to them. *
*/

enum keywords
{
  INT=100,
  LONG,
  LONG_LONG,
  SHORT,
  SIGNED,
  UNSIGNED,
  FOR,
  BREAK,
  CONTINUE,
  RETURN,
  CHAR,
  IF,
  ELSE
};
```

```
enum operators
{
  DECREMENT=200,
  INCREMENT,
  PTR_SELECT,
  LOGICAL_AND,
  LOGICAL_OR,
  LS_THAN_EQ,
  GR_THAN_EQ, EQ,
  NOT_EQ,
  ASSIGN,
  MINUS,
  PLUS,
  STAR,
  MODULO,
  LS_THAN,
  GR_THAN
};

enum special_symbols
{
  DELIMITER=300,
  OPEN_BRACES,
  CLOSE_BRACES,
  COMMA,
  OPEN_PAR,
  CLOSE_PAR,
  OPEN_SQ_BRKT,
  CLOSE_SQ_BRKT,
  FW_SLASH
};

enum constants
{
  HEX_CONSTANT=400,
  DEC_CONSTANT,
  HEADER_FILE,
  STRING
};

enum IDENTIFIER
{
  IDENTIFIER=500
};
```

Test-cases & Screenshots

```
 /*
Compiler Design Project 1

Test Case 1
 - Test for single line comments
 - Test for multi-line comments
 - Test for single line nested comments
 - Test for multiline nested comments

The output in lex should remove all the comments including this one */

#include<stdio.h>

void main(){
        // Single line comment

        /* Multi-line comment
        Like this */

        /* here */ int a; /* "int a" should be untouched */

    // This nested comment // This comment should be removed should be removed



    /* To make things /* nested multi-line comment */ interesting */

    return 0;
}
```



test-case-1

```c
/*
Compiler Design Project 1

Test Case 2
 - Test for multi-line comment that doesn't end till EOF

The output in lex should print as error message when the comment does not terminate
It should remove the comments that terminate */

#include<stdio.h>

void main(){

    // This is fine
    /* This as well
    like we know */

    /* This is not fine since this
    comment has to end somewhere

    return 0;
}
```

```
~/workspace/Sem VI/CD - CO350/project/mini-c-compiler/Project-1  master ●  ./a.out
        <stdio.h>                       : 402
        void                            : 500
        main                            : 500
        (                               : 304
        )                               : 305
        {                               : 301
Line  21: Unterminated comment

        Symbol table
========================================
         < lexeme , token >
========================================
< void                          , 500 >
< main                          , 500 >
========================================

        Constants Table
========================================
         < lexeme , token >
========================================
========================================
NOTE: Please refer tokens.h for token meanings
~/workspace/Sem VI/CD - CO350/project/mini-c-compiler/Project-1  master ●
```

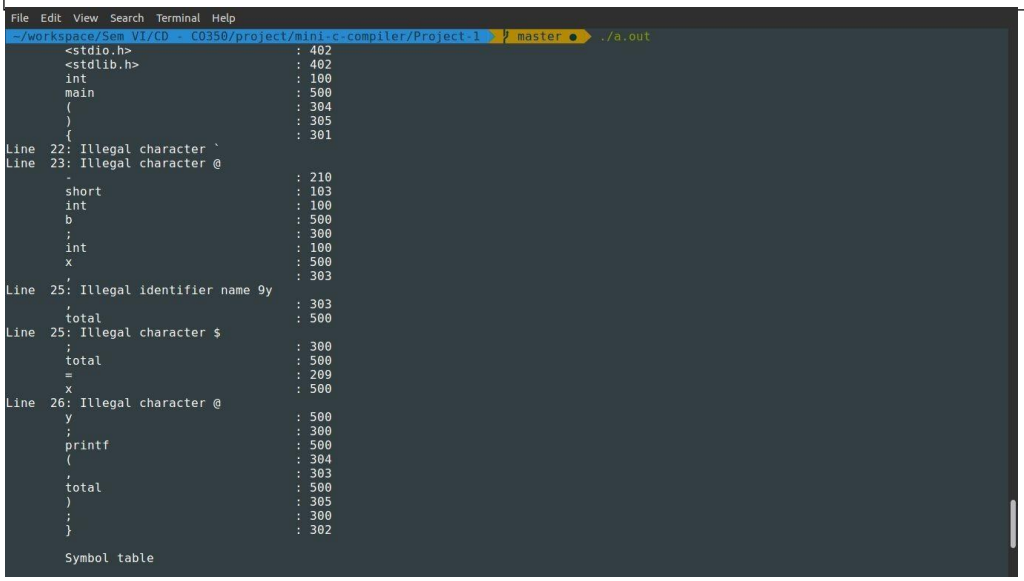
test-case-2

```c
/*
Compiler Design Project 1

Test Case 3
 - Test for string
 - Test for string that doesn't end till EOF
 - Test for invalid header name

The output in lex should identify the first string correct and display error message that
the second one does not terminate */

#include<stdio.h>
#include <<stdlib.h>
#include "custom.h"
#include ""wrong.h"
void main(){

    printf("This is a string");
    printf("This is a string that never terminates); }
```

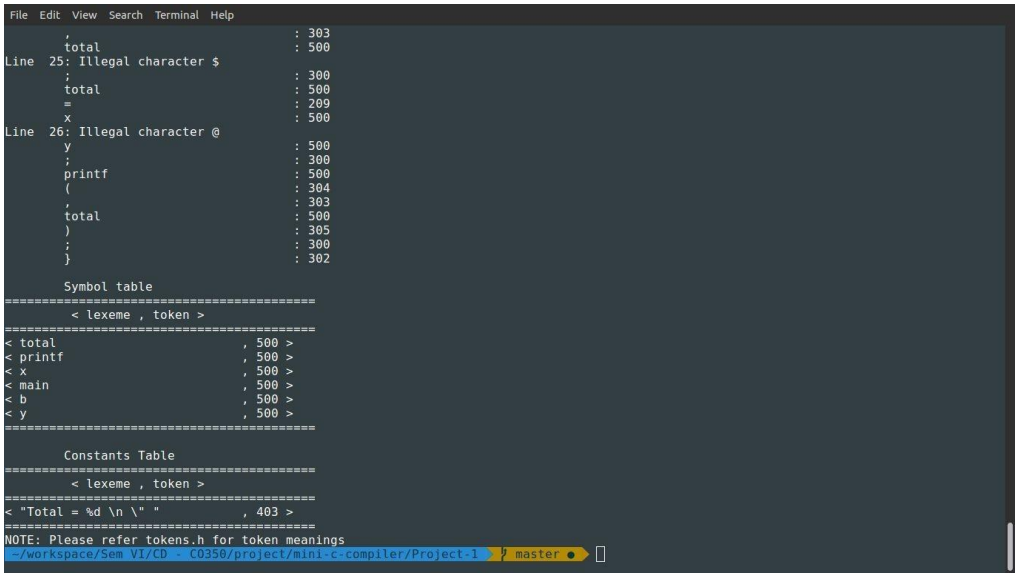
```
File  Edit  View  Search  Terminal  Help
~/workspace/Sem VI/CD - CO350/project/mini-c-compiler/Project-1  master ●  cc lex.yy.c -ll
~/workspace/Sem VI/CD - CO350/project/mini-c-compiler/Project-1  master ●  ./a.out
        <stdio.h>                       : 402
Line  15: Illegal header file format
        <stdlib.h>                      : 402
        "custom.h"                      : 402
Line  17: Illegal header file format
        "wrong.h"                       : 402
        void                            : 500
        main                            : 500
        (                               : 304
        )                               : 305
        {                               : 301
        printf                          : 500
        (                               : 304
        )                               : 305
        ;                               : 300
        printf                          : 500
        (                               : 304
Line  22: Unterminated string "This is a string that never terminates);

        Symbol table
=======================================
         < lexeme , token >
=======================================
< printf                        , 500 >
< void                          , 500 >
< main                          , 500 >
=======================================

        Constants Table
=======================================
         < lexeme , token >
=======================================
< "This is a string"            , 403 >
=======================================
NOTE: Please refer tokens.h for token meanings
~/workspace/Sem VI/CD - CO350/project/mini-c-compiler/Project-1  master ●
```

test-case-3

```c
/*
Compiler Design Project 1

Test Case 4

Following errors must be detected
 -       Invalid identifiers: 9y, total$
 -       Invalid operator: @
 -       Escaped quoted should be part of the string that is identified - Stray characters: `,
 @, -

The output should display appropriate errors */

#include<stdio.h>
#include<stdlib.h>

int main()
{
  `
  @ -
  short int b;
  int x, 9y, total$;
  total = x @ y;
  printf ("Total = %d \n \" ", total);
}
```



Test-case-4a



test-case-4b

```
/*
Test Case 5

Identifying tokens and displaying symbol and constants table

Following tokens must be detected
                    - Keywords      (int, long int, long long int, main include)
```

```
 - Identifiers (main,total,x,y,printf),
 - Constants (-10, 20, 0x0f, 123456l)
 - Strings ("Total = %d \n")
 - Special symbols and Brackets ( (), {}, ;, ,)
 - Operators (+,-,=,*,/,%,--,++)

The output should display appropriate tokens with their type and also the symbol and constants
table
*/
#include<stdio.h>
#include<stdlib.h>

int main()
{
  int x, y;
  long long int total, diff;
  int *ptr; unsigned int a =
  0x0f; long int mylong =
  123456l;
  long int i, j; for(i=0; i <
  10; i++){ for(j=10; j > 0;
  j--){
        printf("%d",i);
        }
  }
  x = -10, y = 20;
  x=x*3/2;
  total = x + y;
  diff = x - y;
  int rem = x % y;
  printf ("Total = %d \n", total);

}
```

```
File  Edit  View  Search  Terminal  Help
~/workspace/Sem VI/CD - CO350/project/mini-c-compiler/Project-1  master ●  ./a.out
        <stdio.h>                       : 402
        <stdlib.h>                      : 402
        int                             : 100
        main                            : 500
        (                               : 304
        )                               : 305
        {                               : 301
        int                             : 100
        x                               : 500
        ,                               : 303
        y                               : 500
        ;                               : 300
        long long                       : 102
        int                             : 100
        total                           : 500
        ,                               : 303
        diff                            : 500
        ;                               : 300
        int                             : 100
        *                               : 212
        ptr                             : 500
        ;                               : 300
        unsigned                        : 105
        int                             : 100
        a                               : 500
        =                               : 209
        0x0f                            : 400
        ;                               : 300
        long                            : 101
        int                             : 100
        mylong                          : 500
        =                               : 209
        123456l                         : 401
        ;                               : 300
        long                            : 101
        int                             : 100
        i                               : 500
```

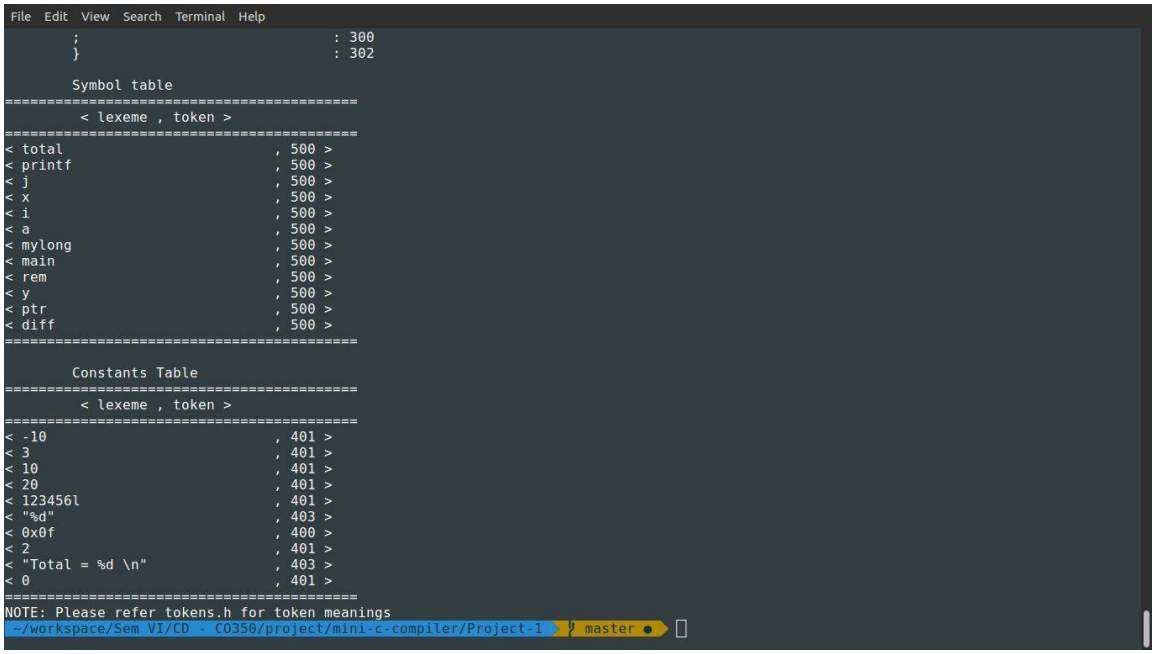
test-case-5a

```
File  Edit  View  Search  Terminal  Help
        int                             : 100
        i                               : 500
        ,                               : 303
        j                               : 500
        ;                               : 300
        for                             : 106
        (                               : 304
        i                               : 500
        =                               : 209
        0                               : 401
        ;                               : 300
        i                               : 500
        <                               : 214
        10                              : 401
        ;                               : 300
        i                               : 500
        ++                              : 201
        )                               : 305
        {                               : 301
        for                             : 106
        (                               : 304
        j                               : 500
        =                               : 209
        10                              : 401
        ;                               : 300
        j                               : 500
        >                               : 215
        0                               : 401
        ;                               : 300
        j                               : 500
        --                              : 200
        )                               : 305
        {                               : 301
        printf                          : 500
        (                               : 304
        ,                               : 303
        i                               : 500
        )                               : 305
```

test-case-5b

```
File  Edit  View  Search  Terminal  Help
        i                               : 500
        )                               : 305
        ;                               : 300
        }                               : 302
        }                               : 302
        x                               : 500
        =                               : 209
        -10                             : 401
        ,                               : 303
        y                               : 500
        =                               : 209
        20                              : 401
        ;                               : 300
        x                               : 500
        =                               : 209
        x                               : 500
        *                               : 212
        3                               : 401
        /                               : 308
        2                               : 401
        ;                               : 300
        total                           : 500
        =                               : 209
        x                               : 500
        +                               : 211
        y                               : 500
        ;                               : 300
        diff                            : 500
        =                               : 209
        x                               : 500
        -                               : 210
        y                               : 500
        ;                               : 300
        int                             : 100
        rem                             : 500
        =                               : 209
        x                               : 500
        %                               : 213
```

test-case-5c


File Edit View Search Terminal Help
; : 300
} : 302
Symbol table
< lexeme , token >
< total , 500 >
< printf , 500 >
< j , 500 >
< x , 500 >
< i , 500 >
< a , 500 >
< mylong , 500 >
< main , 500 >
< rem , 500 >
< y , 500 >
< ptr , 500 >
< diff , 500 >
Constants Table
< lexeme , token >
< -10 , 401 >
< 3 , 401 >
< 10 , 401 >
< 20 , 401 >
< 123456l , 401 >
< "%d" , 403 >
< 0x0f , 400 >
< 2 , 401 >
< "Total = %d \n" , 403 >
< 0 , 401 >
NOTE: Please refer tokens.h for token meanings
~/workspace/Sem VI/CD - CO350/project/mini-c-compiler/Project-1 master

test-case-5d

Result

The lexical analyzer that was created in this project helps us to break down a C source file into tokens as per the C language specifications. Each token (such as identifiers, keywords, special symbols, operators, etc.) has an integer value associated with it, as specified in the **tokens.h** file.

When we design the parser in the next phase, the parser will call upon the Flex program to give it tokens and the lexical analyzer will return to the parser the integer value associated with the tokens as and when required by the parser.

Together with the symbol, the parser will prepare a syntax tree with the help of a grammar that we provide it with. The parser can then logically group the tokens to form meaningful statements and can detect C programming constructs such as arrays, loops, and functions. The parser will also help us identify errors that could not be detected in the lexical analysis phase such as unbalanced parentheses, unterminated statements, missing operators, two operators in a row, etc.