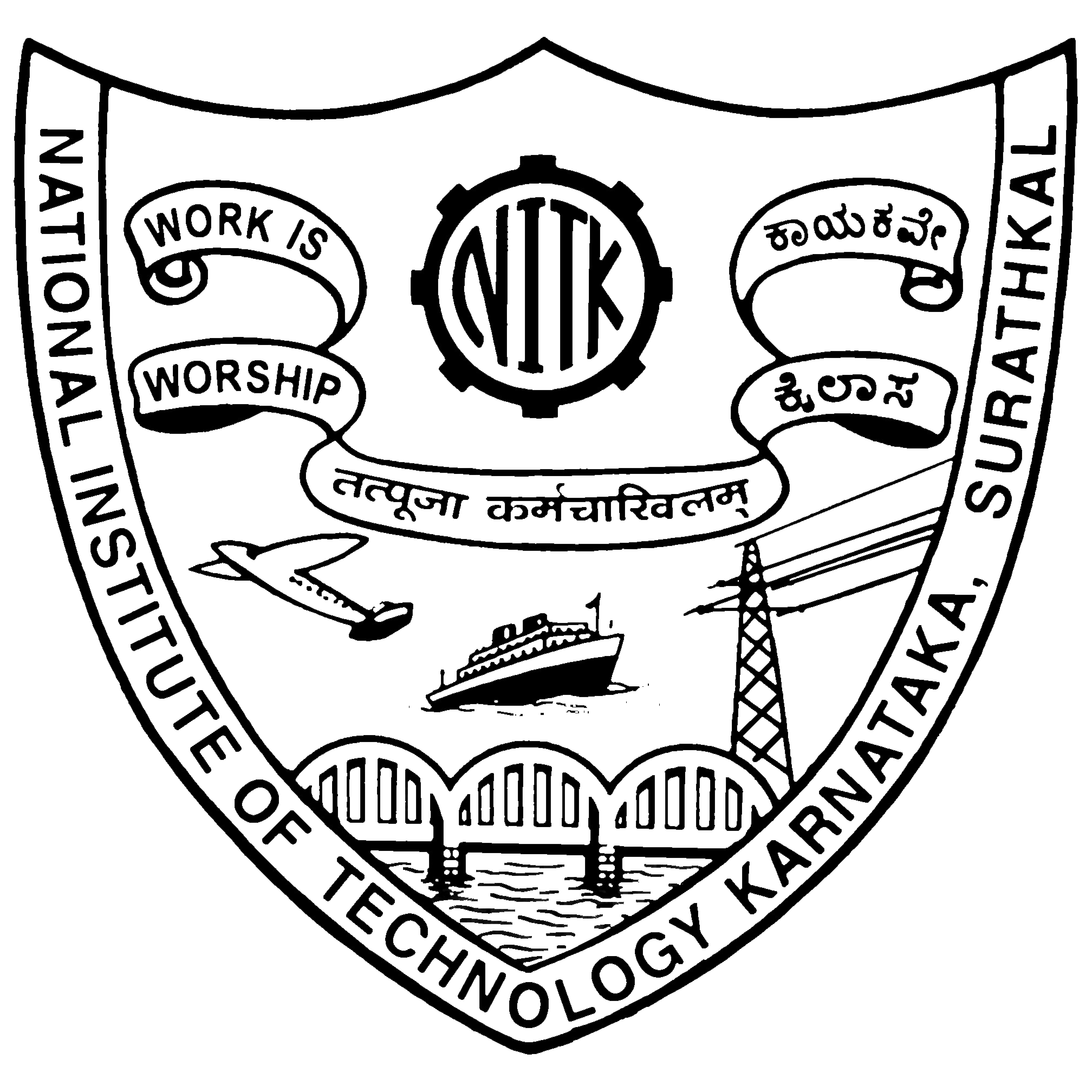
Lexical Analyzer for C



National Institute of Technology Karnataka

Surathkal

**Date:** 28th January 2016

**Submitted To:**

Ms Meghana Pujar

**Group Members:**

Shrukul Habib (13CO143)

Sathvik S (13CO138)

**Abstract:**

This report contains the details of the tasks finished as a part of the Phase One of Compliers Lab. We have developed a Lexer program which scans through a C program and produces different tokens present in the program. The lexer also generates a list of identifiers and constants present in the input program at the end.

The lexer code has functionality of taking input through a file or through standard input. This makes it more user friendly and efficient at the same time.

# **Contents**:

* Introduction
  + Lexical Analyzer
  + Flex Script
  + C Program
* Design of Programs
  + Code
  + Explanation
* Test Cases
  + Without Errors
  + With Errors
* Implementation
* Results / Future work
* References

**List of Figures and Tables:**

1. Table 1: Test Cases without errors
2. Table 2: Test cases with errors
3. Figure 1: Input for: For loop with valid and invalid strings
4. Figure 2: Output for: For loop with valid and invalid strings
5. Figure 3: Input for: Various forms of multi-line comment
6. Figure 4: Output for: Various forms of multi-line comment
7. Figure 5: Input for: Sample C program for binary search.
8. Figure 6: Output for: Sample C program for binary search.

**Introduction**

**Lexical Analysis**

In computer science, lexical analysis is the process of converting a sequence of characters (such as in a computer program or web page) into a sequence of tokens (strings with an identified "meaning"). A program that performs lexical analysis may be called a lexer, tokenizer, or scanner (though "scanner" is also used to refer to the first stage of a lexer). Such a lexer is generally combined with a parser, which together analyze the syntax of programming languages, web pages, and so forth.

**Flex Script**

The script written by us is a program that generates lexical analyzers ("scanners" or "lexers"). Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.

The structure of our flex script is intentionally similar to that of a yacc file; files are divided into three sections, separated by lines that contain only two percent signs, as follows:

*Definition section*

*%%*

*Rules section*

*%%*

*C code section*

The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.

The rules section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.

The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file linked in at compile time.

**C Program**

This section describes the input C program which is fed to the flex script in order to generate the lex file after taking all the rules mentioned in account. Finally, a file called lex.yy.c is generated, which when executed recognizes the tokens present in the C program which was given as an input.

The script also has an option to take standard input instead of taking input from a file.

**Design of Programs**

**Code:**

digit [0-9]

letter [a-zA-Z]

operator [=\*+/%^&|!?]|"-"|"++"|"--"

logical\_op "=="|"<"|">"|"<="|">="|"!"|"!="|"&&"|"||"

int\_literal {digit}+

float\_literal {digit}+.{digit}+

%{

enum

{

IDENTIFIER,

CONSTANT

};

int check\_present(int type, char \*word);

void add\_to\_table(int type, char \*word);

void print\_symbol\_table();

%}

%%

#.\* printf("PRE\_PROCESSOR\_INSTRUCTION: %s\n", yytext);

";" printf("SEMICOLON\n");

":" printf("COLON\n");

"," printf("COMMA\n");

"'" printf("QUOTES\n");

"//"(.\*) printf("SINGLE\_LINE COMMENT: %s\n", yytext);

"/\*"([^\*]\*"\*"+[^\*/])\*[^\*]\*"\*"+"/" printf("MULTI\_LINE COMMENT: %s\n", yytext);

"/\*"([^\*/]|("\*"+[^/]))\* printf("ERR\_UNMATCHED\_COMMENT: %s\n", yytext);

"(" printf("ROUND\_BRACKET\_OPEN\n");

")" printf("ROUND\_BRACKET\_CLOSE\n");

"[" printf("SQUARE\_BRACKET\_OPEN\n");

"]" printf("SQUARE\_BRACKET\_CLOSE\n");

"{" printf("CURLY\_BRACKET\_OPEN\n");

"}" printf("CURLY\_BRACKET\_CLOSE\n");

({operator}|{logical\_op}) printf("OPERATOR: %s\n", yytext);

{int\_literal}({letter}|"\_")({letter}|{digit}|"\_")\* printf("ERR\_UNCLEAN\_INTEGER: %s\n", yytext);

{int\_literal} printf("INTEGER: %s\n", yytext); add\_to\_table(CONSTANT, yytext);

{float\_literal}({letter}|"\_")({letter}|{digit}|"\_")\* printf("ERR\_UNCLEAN\_FLOAT: %s\n", yytext);

{float\_literal} printf("FLOAT: %s\n", yytext); add\_to\_table(CONSTANT, yytext);

\".\*\" printf("STRING: %s\n", yytext); add\_to\_table(CONSTANT, yytext);

\"[^"\n]\* printf("ERR\_INCOMPLETE\_STRING: %s\n", yytext);

auto |

break |

case |

char |

const |

continue |

default |

do |

double |

else |

enum |

extern |

float |

for |

goto |

if |

int |

long |

register |

return |

short |

signed |

sizeof |

static |

struct |

switch |

typedef |

union |

unsigned |

void |

volatile |

while printf("KEYWORD: %s\n", yytext);

({letter}|\_)({letter}|{digit}|\_)\* printf("IDENTIFIER: %s\n", yytext); add\_to\_table(IDENTIFIER, yytext);

. ; //printf("ERR\_FOREIGN\_CHARACTER: %s\n", yytext);

%%

int main(int argc, char \*\*argv)

{

if(argc == 2)

yyin = fopen(argv[1], "r");

yylex();

print\_symbol\_table();

}

struct word

{

char \*word\_name;

struct word \*next;

};

struct word \*identifier\_list=NULL, \*constant\_list=NULL;

int check\_present(int type, char \*word)

{

struct word \*current\_list = (type == IDENTIFIER ? identifier\_list : constant\_list);

while(current\_list)

{

//printf("Comparing %s with %s\n", word, current\_list->word\_name);

if(strcmp(current\_list->word\_name, word)==0)

{

//printf("Word already present in table.\n");

return 1;

}

current\_list = current\_list->next;

}

//printf("Adding word %s to symbol table.\n", word);

return 0;

}

void add\_to\_table(int type, char \*word)

{

if(check\_present(type, word))

return;

struct word \*current\_list;

struct word \*new\_word = (struct word \*) malloc(sizeof(struct word));

new\_word->word\_name = (char \*) malloc(strlen(word)+1);

strcpy(new\_word->word\_name, word);

new\_word->next = NULL;

switch(type)

{

case IDENTIFIER:

if(!identifier\_list)

{

identifier\_list = new\_word;

return;

}

current\_list = identifier\_list;

break;

case CONSTANT:

if(!constant\_list)

{

constant\_list = new\_word;

return;

}

current\_list = constant\_list;

break;

}

while(current\_list->next)

current\_list = current\_list->next;

current\_list->next = new\_word;

}

void print\_symbol\_table()

{

printf("\n----- Begin Symbol Table -----\n");

printf("\n--- IDENTIFIERS ---\n");

struct word \*list = identifier\_list;

while(list)

{

printf("%s\n", list->word\_name);

list = list->next;

}

printf("\n--- CONSTANTS ---\n");

list = constant\_list;

while(list)

{

printf("%s\n", list->word\_name);

list = list->next;

}

printf("\n----- End Symbol Table -----\n\n");

}

**Explanation:**

The flex script recognises the following classes of tokens from the input:

* Pre-processor instructions
* Single-line comments
* Multi-line comments
* Errors for unmatched comments
* Errors for nested comments
* Parentheses (all types)
* Operators
* Literals (integer, float, string)
* Errors for unclean integers and floating point numbers
* Errors for incomplete strings
* Keywords
* Identifiers

Keywords accounted for:

auto

break

case

char

const

continue

default

do

double

else

enum

extern

float

for

goto

if

int

long

register

return

short

signed

sizeof

static

struct

switch

typedef

union

unsigned

void

volatile

while

**Test Cases:**

**Without Errors:**

|  |  |  |  |
| --- | --- | --- | --- |
| Serial No | Test Case | Expected Output | Status |
| 1 | int a=25; | KEYWORD: int  IDENTIFIER: a  OPERATOR: =  INTEGER: 25  SEMICOLON | PASS |
| 2 | \_iterator++; | IDENTIFIER: \_iterator  OPERATOR: ++  SEMICOLON | PASS |
| 3 | printf(“Apple pie”); | IDENTIFIER: printf  ROUND BRACKET OPEN  STRING: “Apple pie”  ROUND BRACKET CLOSE  SEMICOLON | PASS |
| 4 | /\* Hello  World\*/ | MULTILINE\_COMMENT:  /\* Hello  World\*/ | PASS |
| 5 | // Hallelujah | SINGLE\_LINE\_COMMENT:  // Hallelujah | PASS |
| 6 | for(i=1; i<=5; i++)  j = i; | KEYWORD: for  ROUND BRACKET OPEN  IDENTIFIER: i  OPERATOR: =  SEMICOLON  IDENTIFIER: i  OPERATOR: <=  INTEGER: 5  SEMICOLON  IDENTIFIER: i  OPERATOR: ++  ROUND BRACKET CLOSE  IDENTIFIER: j  OPERATOR: =  IDENTIFIER: i  SEMICOLON | PASS |
| 7 | char str[10] = “a<b”; | KEYWORD: char  IDENTIFIER: str  SQUARE BRACKET OPEN  INTEGER: 10  SQUARE BRACKET CLOSE  OPERATOR: =  STRING: “a<b”  SEMICOLON | PASS |
| 8 | switch(a)  {  case 1:  break;  case 2:  b = a;  break;  } | KEYWORD: switch  ROUND BRACKET OPEN  IDENTIFIER: a  ROUND BRACKET CLOSE  CURLY BRACKET OPEN  KEYWORD: case  INTEGER: 1  COLON  KEYWORD: break  SEMICOLON  KEYWORD: case  INTEGER: 2  COLON  IDENTIFIER: a  OPERATOR: =  IDENTIFIER: b  SEMICOLON  KEYWORD: break  SEMICOLON  CURLY BRACKET CLOSE | PASS |

**With Errors:**

|  |  |  |  |
| --- | --- | --- | --- |
| Serial No | Test Case | Expected Output | Status |
| 1 | atmeg = “dfsfds | IDENTIFIER: atmeg  OPERATOR: =  ERR\_INCOMPLETE\_STRING:  “dfsfds | PASS |
| 2 | /\* dead meat | ERR\_UNMATCHED\_COMMENT:  /\* dead meat | PASS |

**Implementation**

The Regular Expressions for most of the features of C are fairly straightforward. However, a few features require a significant amount of thought, such as:

* **The Regex for Identifiers:** The lexer must correctly recognize all valid identifiers in C, including the ones having one or more underscores.
* **Multiline comments should be supported:** This has been supported by using custom regular algorithm especially robust in cases where tricky characters like \* or / are used within the comments.
* **Literals:** Different regular expressions have been implemented in the code to support all kinds of literals, i.e integers, floats, strings, etc.
* **Error Handling for Incomplete String:** Open and close quote missing, both kind of errors have been handled in the rules written in the script.
* **Error Handling for Nested Comments:** This use-case has been handled by the custom defined regular expressions which help throw errors when comment opening or closing is missing.

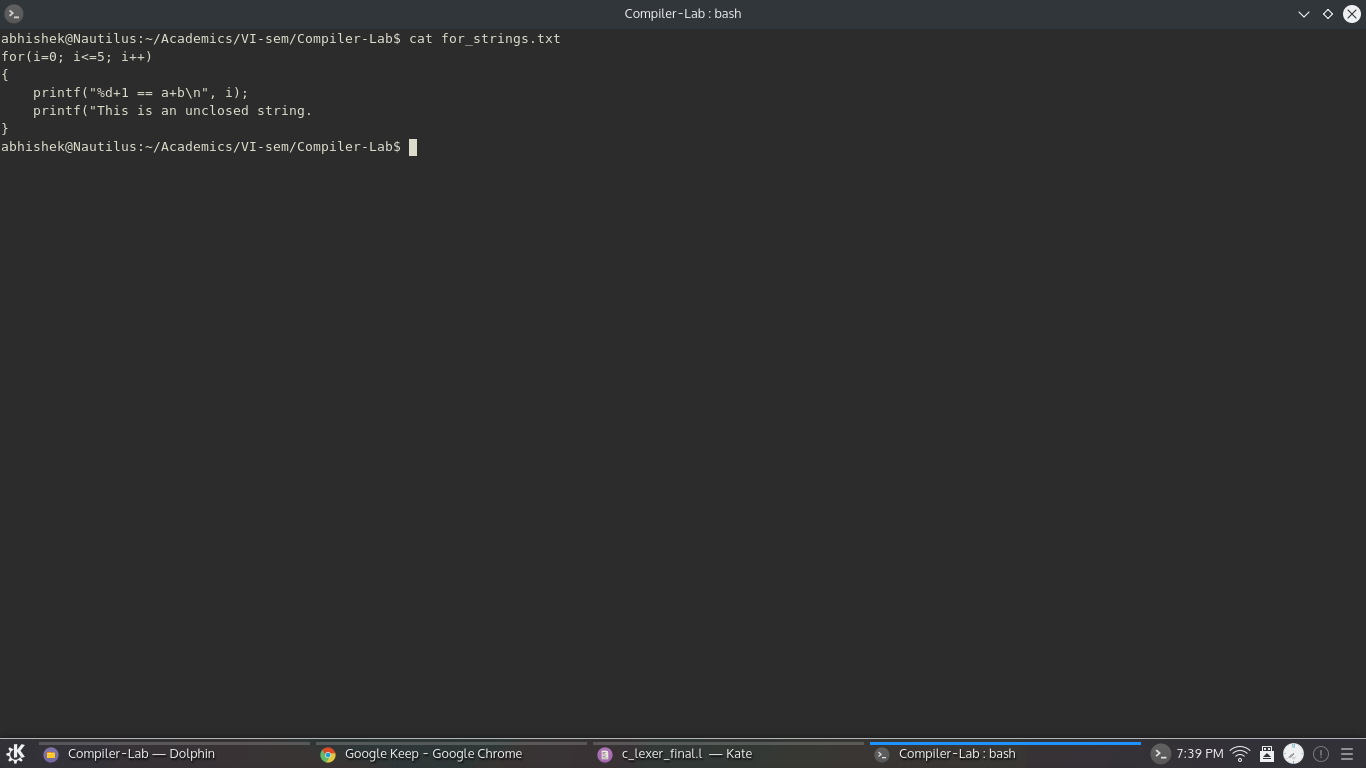
At the end of the token recognition, the lexer prints a list of all the identifiers and constants present in the program. We use the following technique to implement this:

* We maintain two linked lists of words, one corresponding to identifiers and other to constants.
* Two functions have been implemented, namely add\_to\_table() and check\_present() which is used for adding a new identifier/constant to the linked list and for checking if the identifier/constant is already present in the linked list, respectively.
* Whenever we encounter an identifier/constant, we call the add\_to\_table() function which in turns call check\_present() and adds it to the corresponding liked list.
* In the end, in main() function, after yylex returns, we call print\_symbol\_table(), which in turn prints the list of identifier and constants in a proper format.

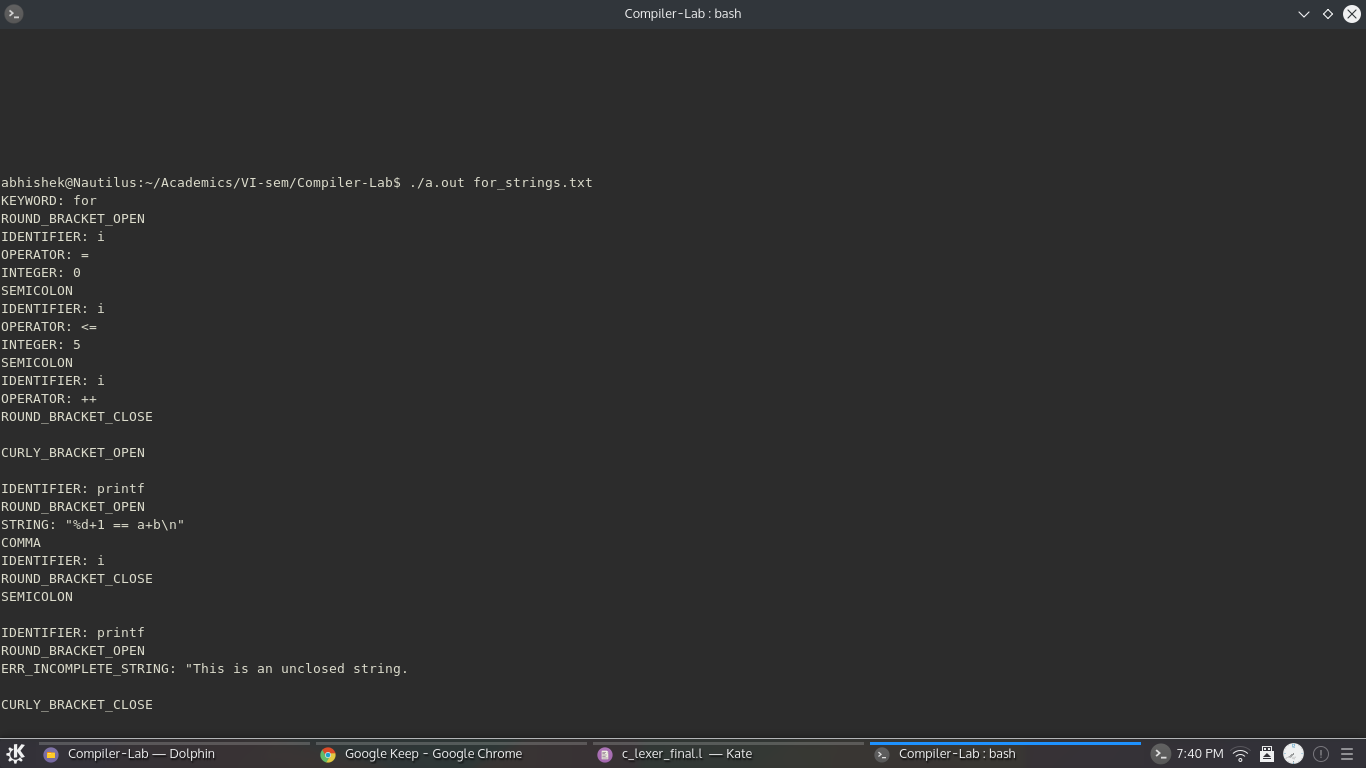
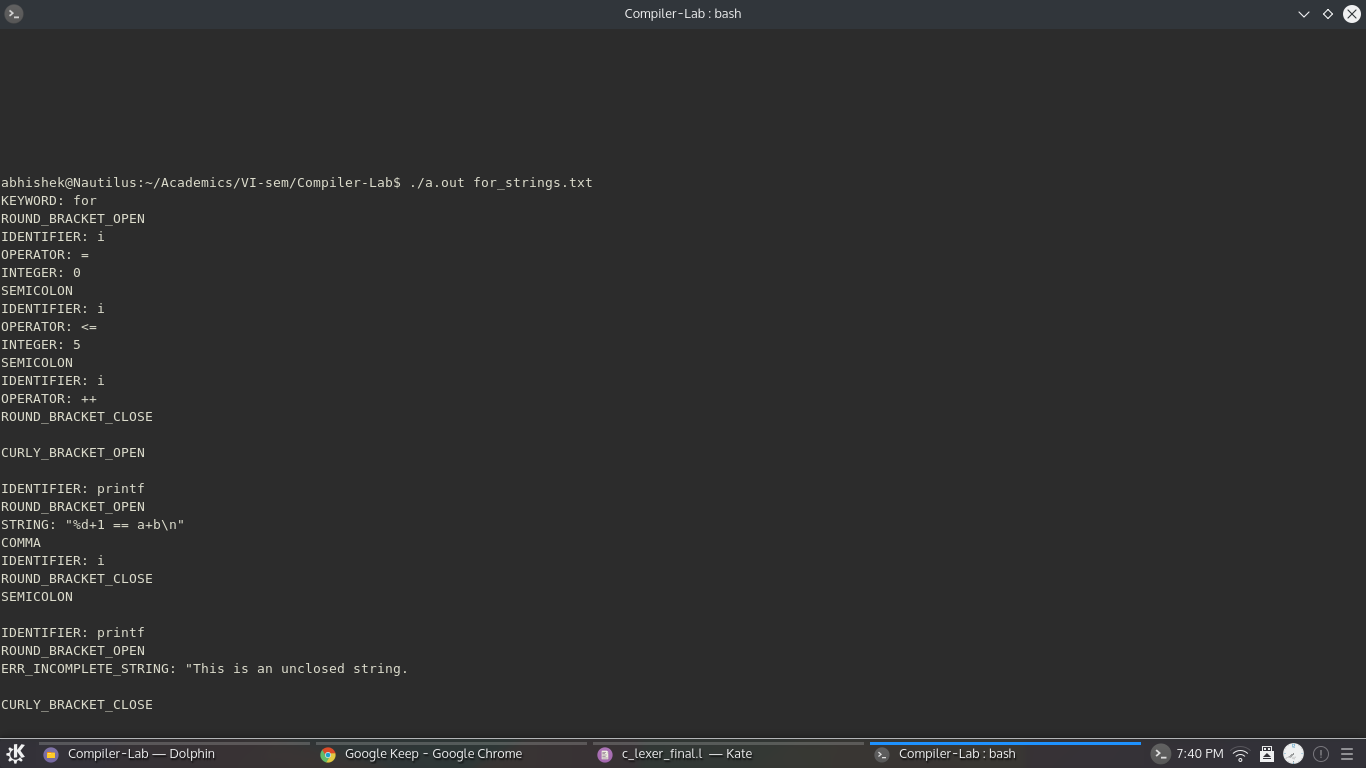
**Results and Future Work**

We were able to successfully recognize tokens present in the input program successfully using the flex script for C. The output also displays the set of identifiers and constants present in the program. We have taken care of all corner cases including error handling related to incomplete string, multiple line comments, nested comments and lexically wrong code.

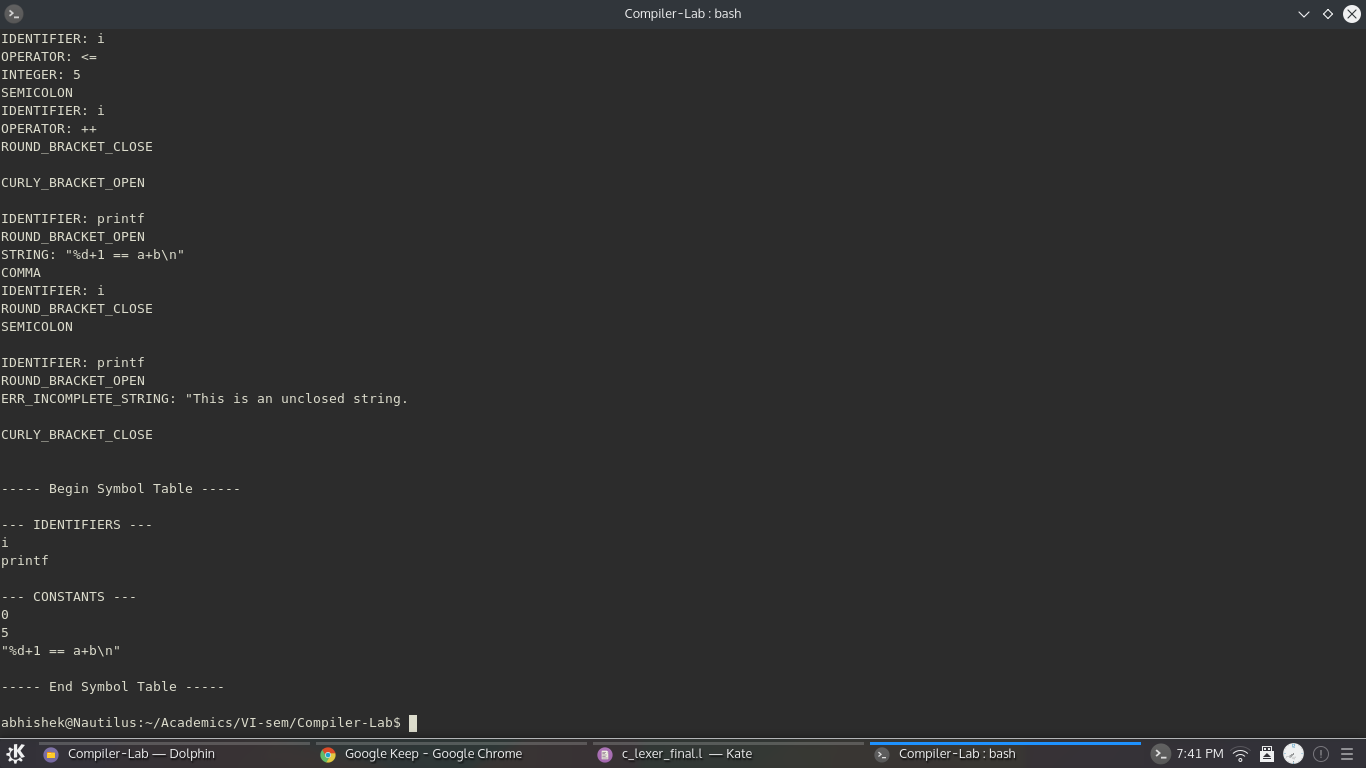
**Input: For loop with valid and invalid strings**



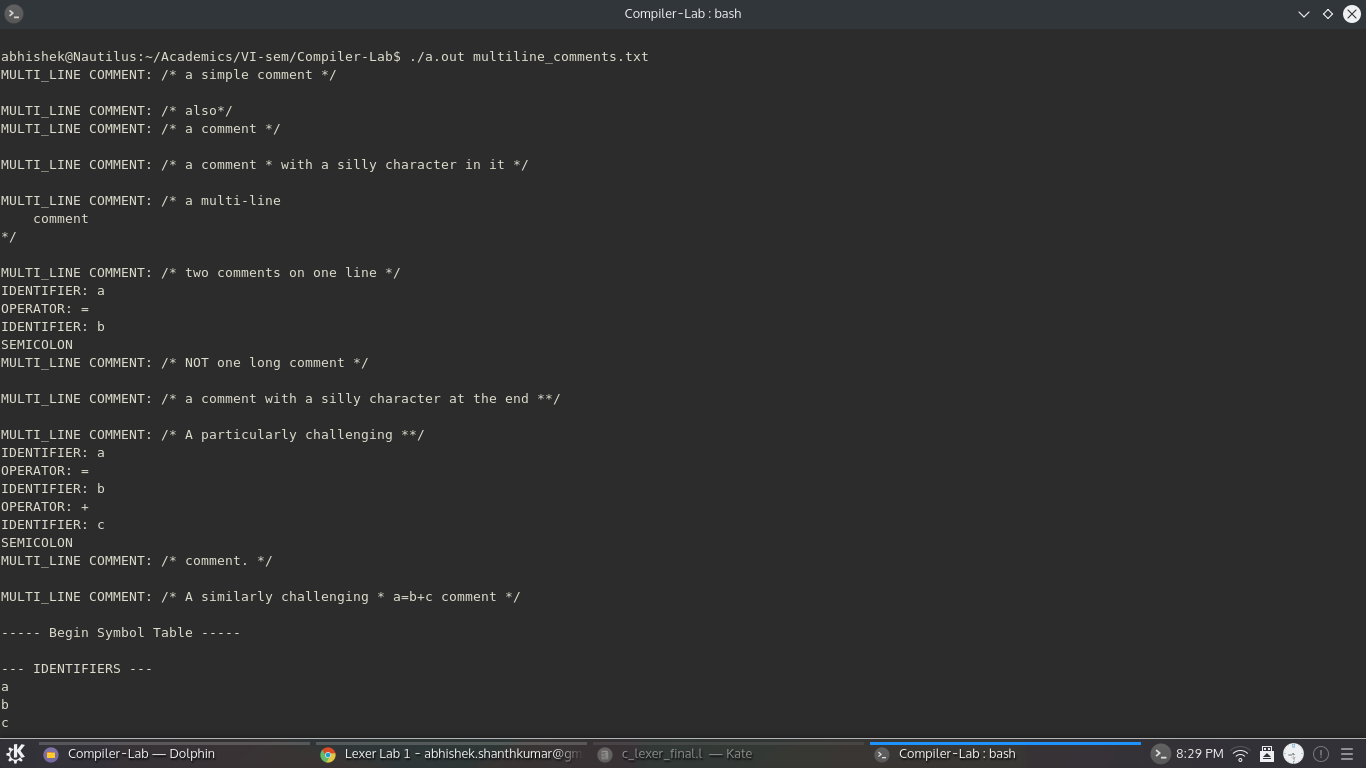
**Output: Tokens**



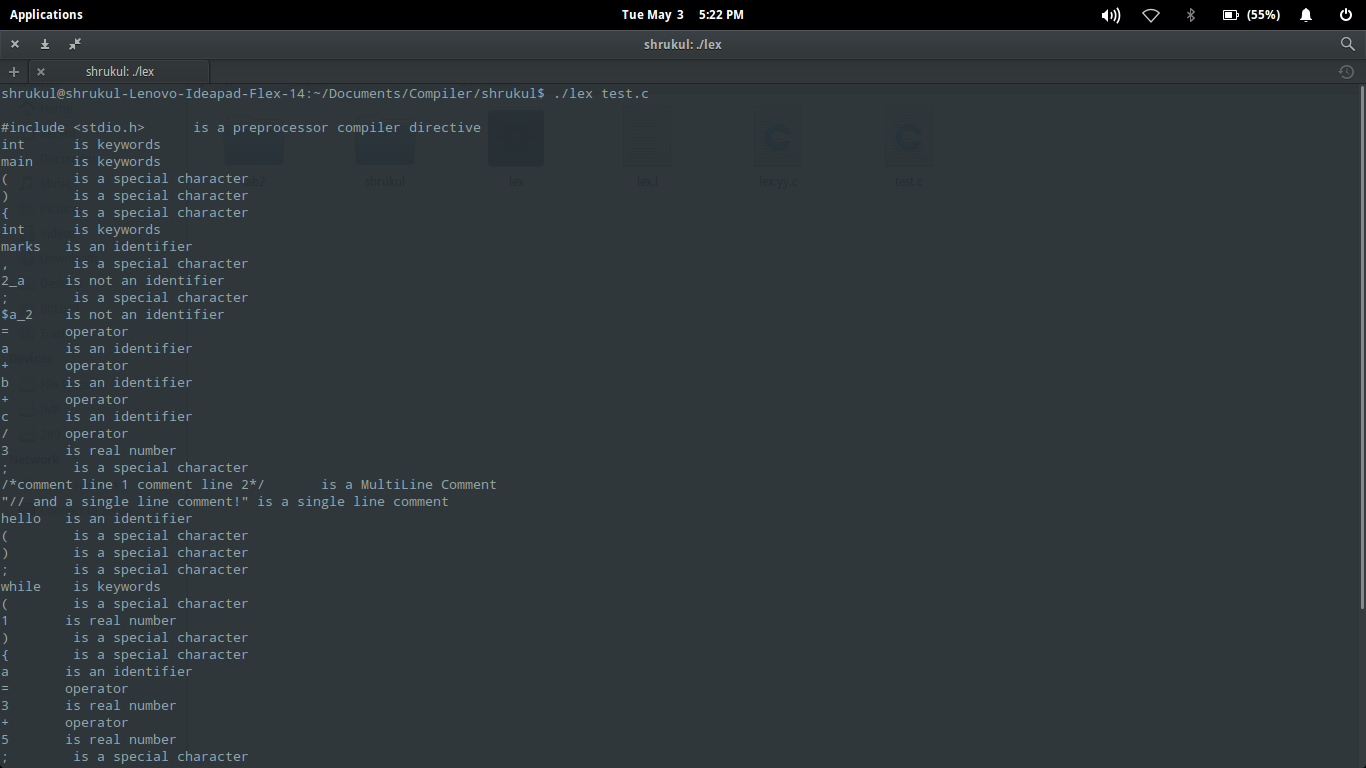
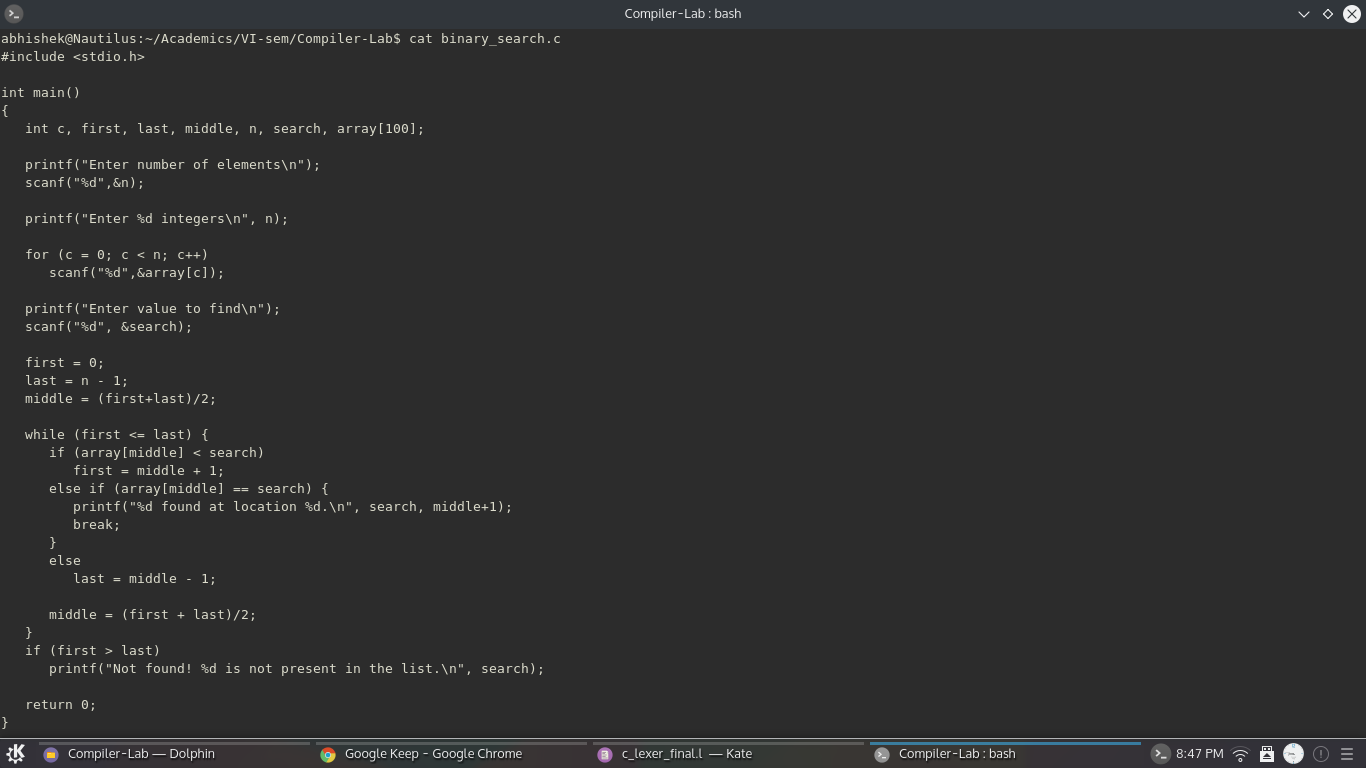
**Output: Symbol Table**



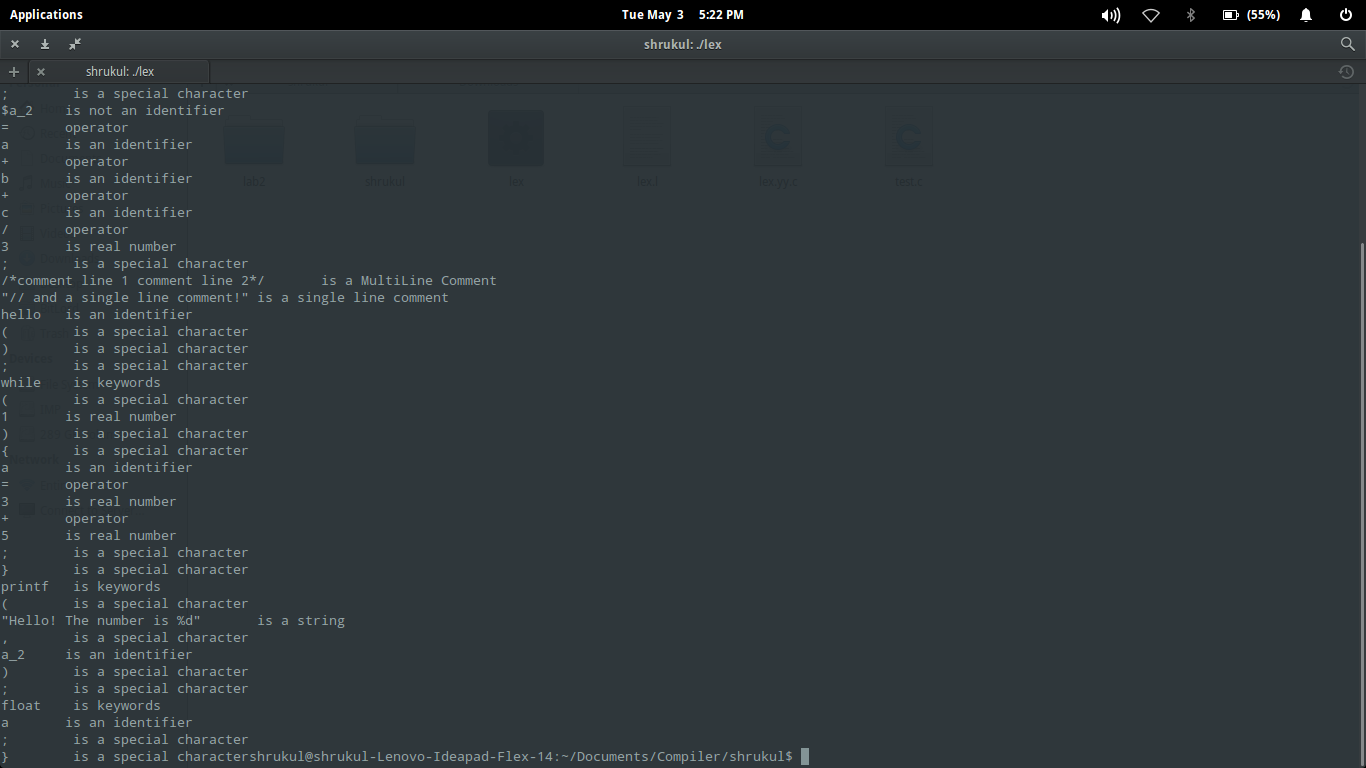
**Input: Various forms of multi-line comment**

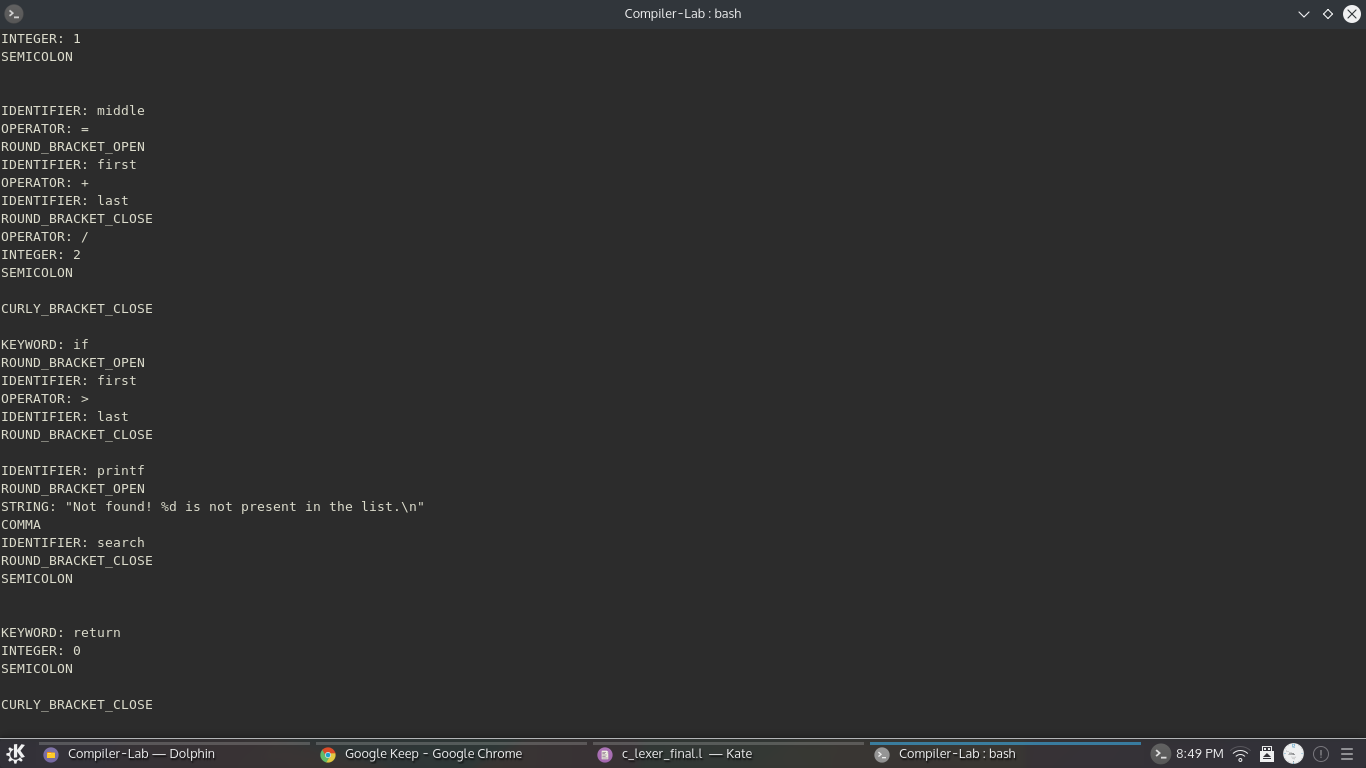
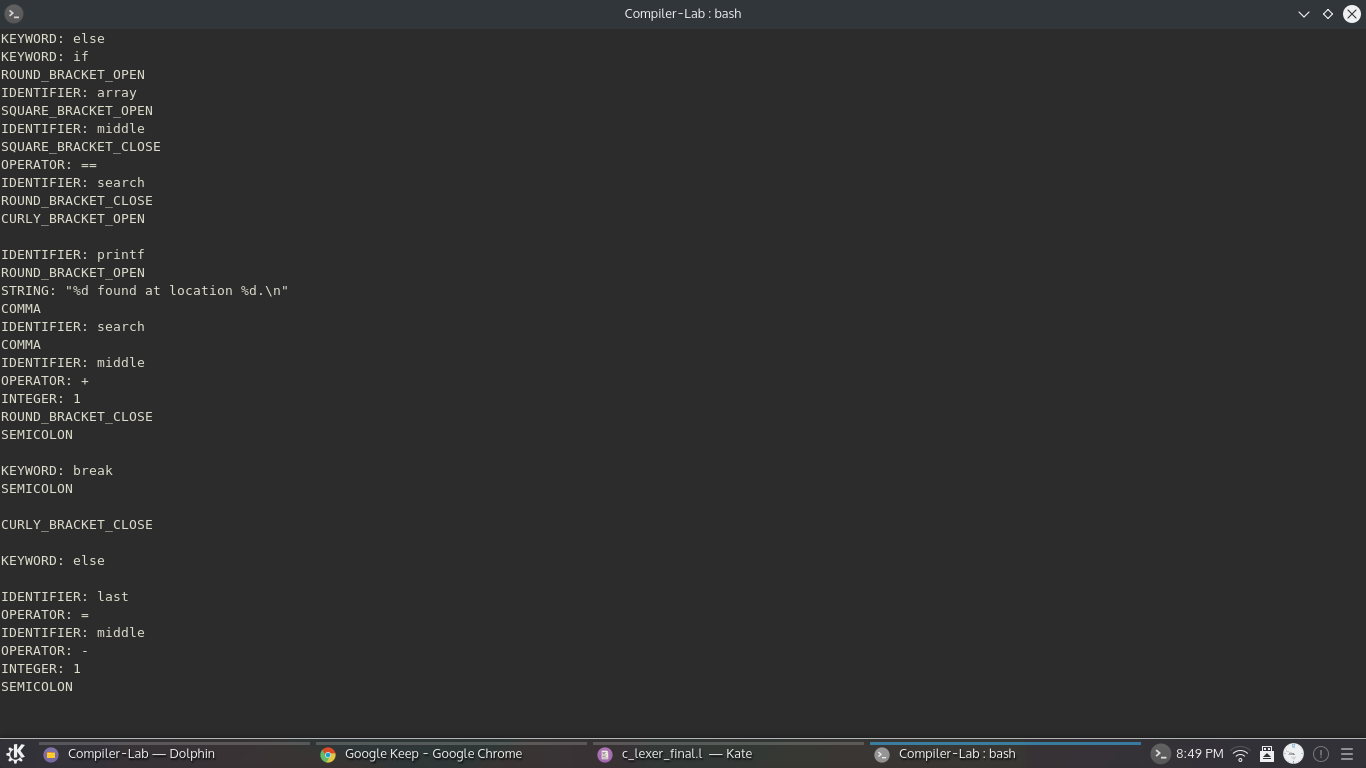
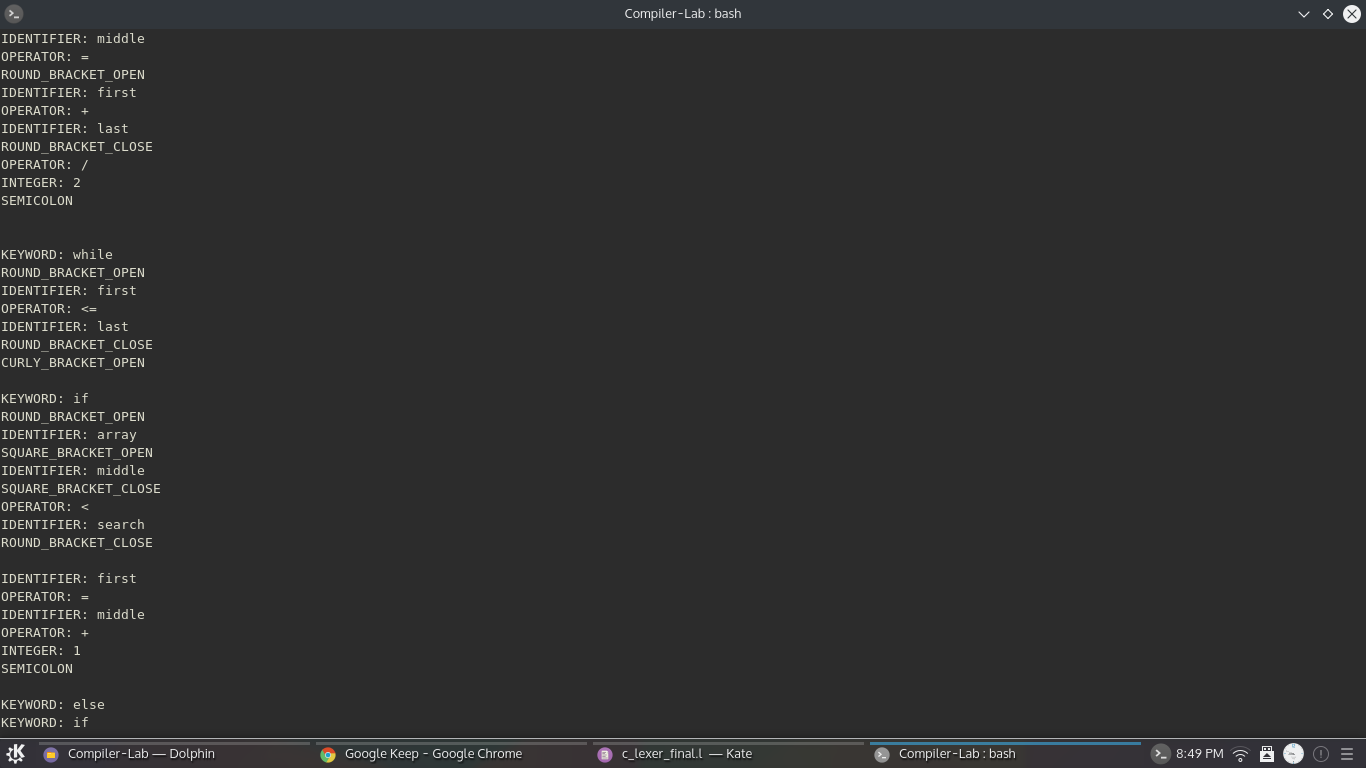
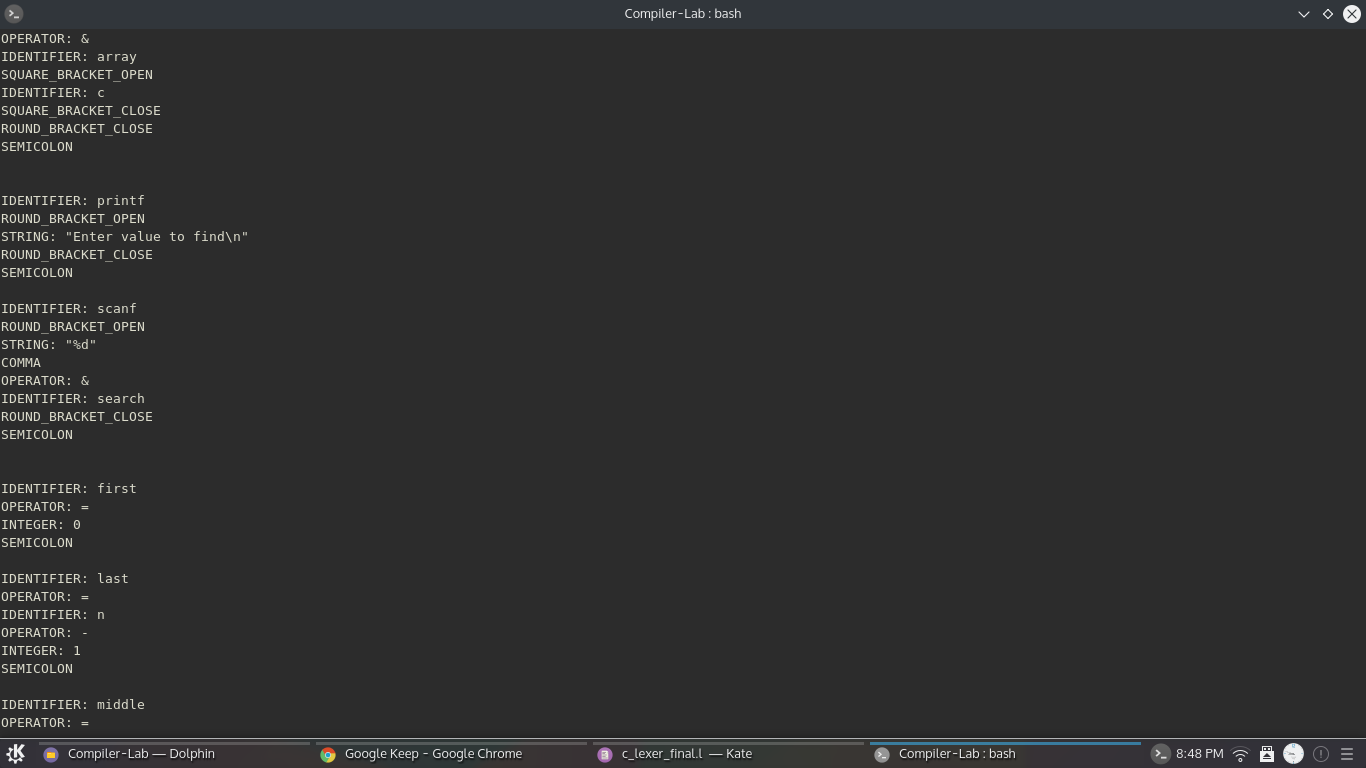
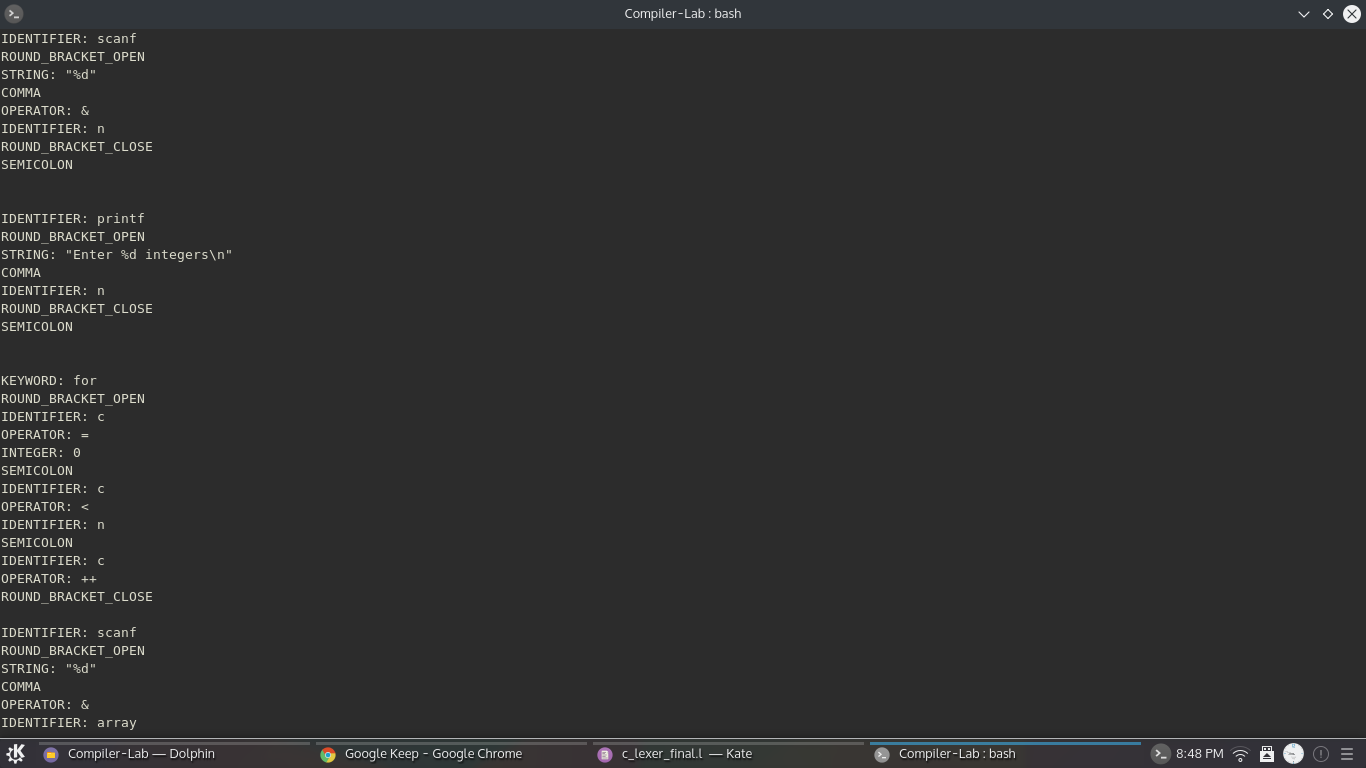
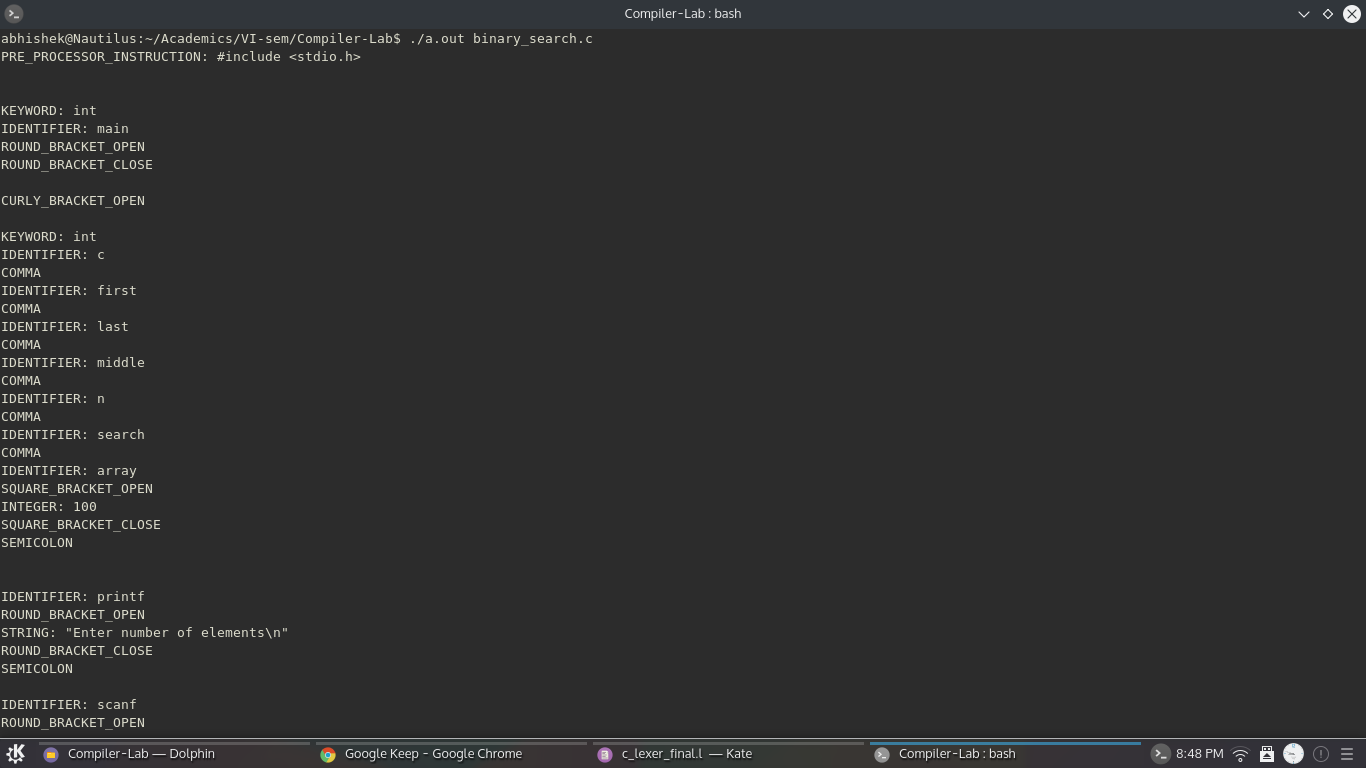


**Input: A sample C program for binary search. Has most features of C.**

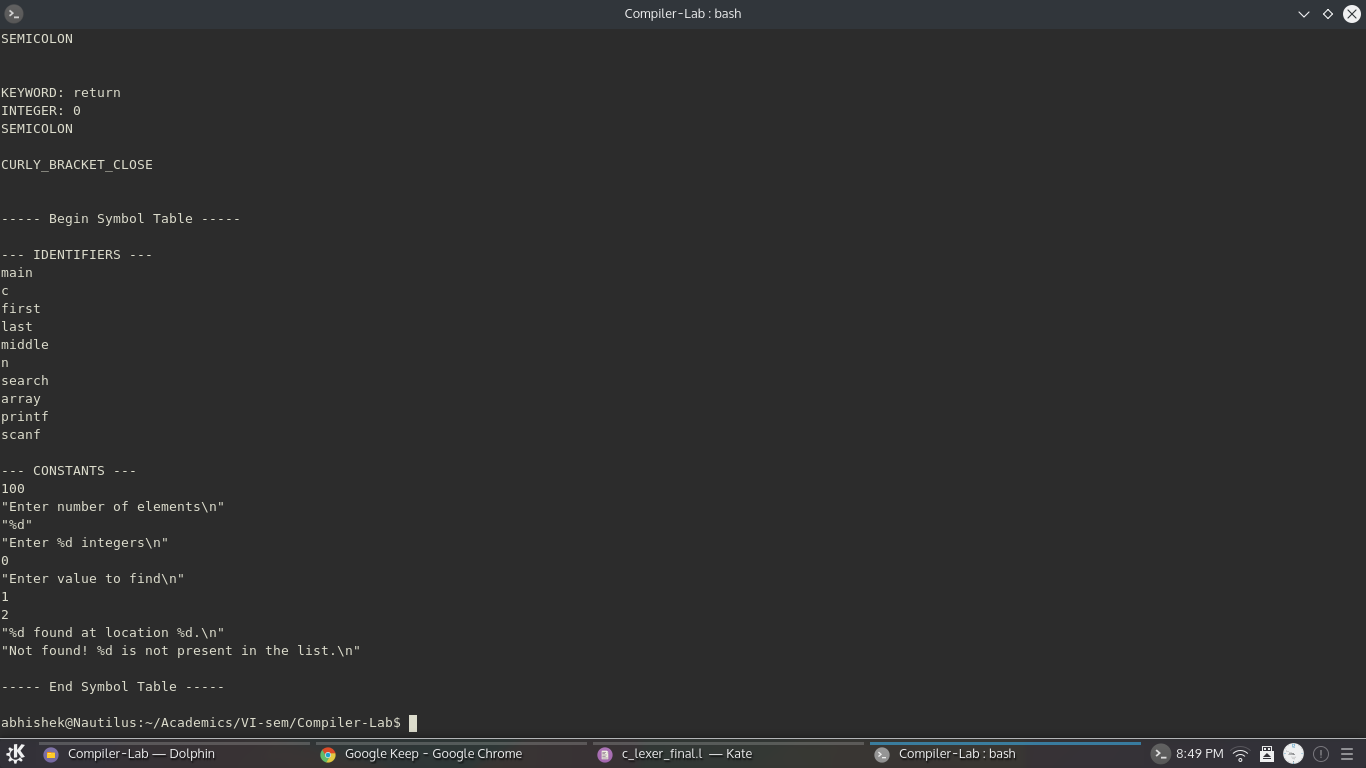


**Output: Tokens**





**Output: Symbol Table**



**Future work:**

The flex script presented in this report takes care of all the rules of C language, but is not fully exhaustive in nature. Our future work would include making the script even more robust in order to handle all aspects of C language and making it more efficient.

**References**

[**https://en.wikipedia.org/wiki/Lex\_(software)**](https://en.wikipedia.org/wiki/Lex_(software))

[**http://dinosaur.compilertools.net/**](http://dinosaur.compilertools.net/)

[**https://en.wikipedia.org/wiki/Lexical\_analysis**](https://en.wikipedia.org/wiki/Lexical_analysis)