**Project Report**

Lexical Analysis for C Compiler

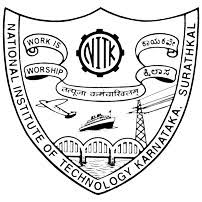
**Submitted by**

Agnitha Mohanram-15CO201

Vasudha Boddukuri-15CO211

**Under Guidance of**   
Sushmita

Umapriya



**National Institute of Technology-Karnataka**

**Contents**

1 Acknowledgement

2 Introduction 2.1 Overview

2.2 Motivation

2.3 Objectives

2.4 Concepts and tools

3 Compiler Design

4 Lexical Phase

5 Source Code

6 Regular Expressions

7 Output

8 Conclusion and Future Work

9 References

**Acknowledgement**

We would like to express our gratitude to Ms.Santhi Thilagam, Ms. Sushmita and Ms.Umapriya for their constant support, guidance and help. A special thanks to NITK for providing us with the resources and facilities required to complete our project successfully. This learning experience was extremely helpful and valuable. Last but not the least, we’re very thankful to our parents for their unwavering support and encouragement.

**Introduction**

**Overview**

The report elaborates on the lexical analysis phase of building a C-compiler. It describes the procedure used to built the lexical analyzer and contains the output obtained from several test cases. It also enlists the various tools and concepts used in the implementation.

**Motivation**

A compiler is a software that aims to transform code submitted to it which is written in one programming language(the source language) into another programming language(the target language).

Compilers are extensively used in the programming world and it’s essential for a computer science engineer to understand the working of a compiler. By doing so, he/she gains thorough knowledge of the system software and can manipulate it with ease if need arises.

The project aims to build a primitive compiler from scratch through the various phases of compiler design.   
 This report is on the lexical analysis phase where the input program is converted into sequence of tokens which are stored in a symbol table.

**Objective**

The objectives of the project are as follows:

* to study the grammar of the source language and identify the lexical components of the language specifications
* to study the input format of Flex and clearly understand how tokens are described
* to identify the tokens to be returned by the scanner
* to store the identified tokens in a symbol table
* to generate the scanner
* to understand parallelisation of programs to achieve optimised performance

**Concepts and Tools**

**Compiler**

Compilers are system softwares that convert code written in one language (usually high level language) into an equivalent machine understandable language to generate an executable.

**Lexical Analysis**

Lexical Analysis is the process of converting the input program into a sequence of tokens. The lexical analyzer scans the source program from left to right in a single pass. It removes unwanted information from the source program.

**Hashing**

Hashing is the process of converting given string into a key value by using some mathematical functions. These key values are used to optimize time complexity for searching, insertion and deletion from memory.

**Hash Table**

A hash table is a data structure which is used to map keys generated by hashing to the values they were generated from.

**Flex**

Lex is a computer 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.

**Git**

Git is a version control system(VCS) which helps keep track of changes made in projects. It is used to maintain records of changes made to any set of files.

**Compiler Design**

**Compiler**

A compiler is computer software that transforms computer code written in one programming language(the source language) into another programming language (the target language) without changing the meaning of the program.

The compiler is expected to make the target code efficient and optimized I terms of time and space complexities.

Compiler design deals with the translation and optimization processes. It covers basic translation mechanism, error detection and errors recovery.

Compiler design has two phases:

**FRONT END**

* 1. Lexical Analysis
  2. Syntax Analysis
  3. Semantic Analysis
  4. Intermediate Code Generation

**BACK END**

* 1. Code Optimization
  2. Code Generation

The front end builds an intermediate representation(IR) of the source code. It also manages the symbol table and mapping of each token in the program to its associated information.  
  
The back end on the other hand deals with the CPU architecture specific optimizations and code generation.

**Lexical Analysis**

Lexical analysis is the first phase of compiler design. It takes the source code as input an translates it into a sequence of tokens. It removes the unnecessary elements of the source code such as white spaces and comments and generates appropriate error messages if any invalid token is encountered.

**Syntax Analysis**

A syntax analyzer checks the syntactical correctness of the source program. It’s takes the token of streams generated by the lexical phase as input and analyzes

the source code against the production rules to detect errors. The output is in the form of a parse tree.

**Semantic Analysis**

Semantics of the language add meaning to its tokens and syntax structure. The semantic analyzer analysis the parse tree and determines if it derives any meaning. The output of this phase is a more verified parse tree.

**Intermediate Code Generation**

Intermediate code is generated from the verified parse tree. It is then converted to machine language through the last two phases. All the phases till the intermediate code generation are platform independent, whereas the next two phases are not. In order to build a compiler for a platform, one need not start from scratch. He/she could take the intermediate code from an already existing compiler and build the last two phases.

**Code optimization**

The code optimizer transforms the code into an optimized version which is designed to consume fewer resources and work faster. There are two types of optimizations, machine dependent and machine independent.

**Target Code Generator**

This can be looked at as the final stage of compilation. Its function is to write a code that the machine can understand. The output is assembler dependent.

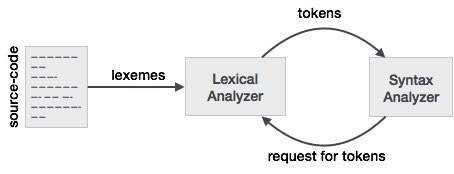
**Lexical Analysis**

The first phase of a compiler is the lexical analysis. It takes a source program as the input and generates a sequence of tokens. A token is a basic component of source code.

These tokens are inserted into a symbol table which maps the tokens to their types. The symbol often holds other information like memory location which may prove to be useful in the later phases.

Common token names are:

* identifiers: names the programmer chooses
* keywords: preexisting names in the programming language
* separators/punctuation: punctuation characters and paired-delimiters
* operators: symbols that operate on arguments and produce results
* literals: numeric, logical, textual, reference literals
* comments: line, block



**Source Code**

Source code:

%{

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

//int searchInHash(int key, char \*name) ;

int top=-1;

char a[100], err\_id\_str[20][10];

int co\_paranthesis=0,co\_curly=0 , co\_square=0 , co\_comment=0 , co\_quotes=0 , mul\_line\_comment=0 ;

int cc\_paranthesis=0, cc\_curly=0 , cc\_square=0 , cc\_comment=0 , err\_comment=0 ;

int mismatch\_paranthesis = 0 , mismatch\_curly = 0 , mismatch\_square = 0 , err\_id=0 , no\_of\_errors=0;

%}

alpha [a-zA-Z]

digit [0-9]

keyword "printf"|"scanf"|"int"|"float"|"if"|"break"|"case"|"char"|"const"|"main"|"default"|"do"|"double"|"else"|"enum"|"float"|"for"|"goto"|"if"|"int"|"long"|"register"|"return"|"short"|"signed"|"sizeof"|"static"|"struct"|"switch"|"typedef"|"union"|"unsigned"|"void"|"volatile"|"while"

operator "+"|"-"|"\*"|"/"|"%"|"++"|"--"|"="|"<"|">"|"&&"|"||"|"|"|"&"|"!"

spl\_symbols ";"

quotes ["]

%%

\n

#include[ ]?<{alpha}\*.h> {

printf("\n\n\t%-30s is a Pre-processor directive\n",yytext);

}

,

" "

\t

"void main()"|"int main()" {

printf("\n\t%-30s is the main function\n",yytext);

}

{alpha}({alpha}|{digit}|[\_])\*\(({alpha}|{digit}|[\_]|[ ])\*\) {

printf("\n\t%-30s is a user defined function\n",yytext);

}

"//".\* {

printf("\n\t%-30s is single line comment\n",yytext);

}

"/\*"([^\*]|\\*+[^\*/])\*+"\*/" {

printf("\n\t%-30s is multi line comment\n",yytext);

}

{keyword} {

printf("\n\t%-30s is a Keyword\n",yytext);

int key=hash\_func(yytext);

if(searchInHash(key,yytext)==0)

insertToHash(key,yytext,"keyword");

}

{spl\_symbols} {

printf("\n\t%-30s is a Special Symbol\n",yytext);

int key=hash\_func(yytext);

if(searchInHash(key,yytext)==0)

insertToHash(key,yytext,"special symbol");

}

{digit}+{alpha}+|{alpha}\*{digit}\*@{alpha}\*{digit}\*|%{alpha}{alpha}+ {

strcpy(err\_id\_str[err\_id],yytext);

err\_id++;

}

"(" {

printf("\n\t%-30s is a Special Symbol\n",yytext);

co\_paranthesis++;

int key=hash\_func(yytext);

if(searchInHash(key,yytext)==0)

insertToHash(key,yytext,"special symbol");

top++;

a[top]='(';

}

"{" {

printf("\n\t%-30s is a Special Symbol\n",yytext);

co\_curly++;

int key=hash\_func(yytext);

if(searchInHash(key,yytext)==0)

insertToHash(key,yytext,"special symbol");

top++;

a[top]='{';

}

"[" {

printf("\n\t%-30s is a Special Symbol\n",yytext);

co\_square++;

int key=hash\_func(yytext);

if(searchInHash(key,yytext)==0)

insertToHash(key,yytext,"special symbol");

top++;

a[top]='[';

}

")" {

printf("\n\t%-30s is a Special Symbol\n",yytext);

cc\_paranthesis++;

if(a[top]!='(')

{

mismatch\_paranthesis=1;

}

else

{top--;}

int key=hash\_func(yytext);

if(searchInHash(key,yytext)==0)

insertToHash(key,yytext,"special symbol");

}

"}" {

printf("\n\t%-30s is a Special Symbol\n",yytext);

cc\_curly++;

if(a[top]!='{')

{

mismatch\_curly=1;

}

else

{top--;}

int key=hash\_func(yytext);

if(searchInHash(key,yytext)==0)

insertToHash(key,yytext,"special symbol");

}

"]" {

printf("\n\t%-30s is a Special Symbol\n",yytext);

cc\_square++;

if(a[top]!='[')

{

mismatch\_square=1;

}

else

{top--;}

int key=hash\_func(yytext);

if(searchInHash(key,yytext)==0)

insertToHash(key,yytext,"special symbol");

}

{quotes} {

printf("\n\t%-30s is a Special Symbol\n",yytext);

co\_quotes++;

int key=hash\_func(yytext);

if(searchInHash(key,yytext)==0)

insertToHash(key,yytext,"special symbol");

}

"/\*" { co\_comment++;

if(mul\_line\_comment==0)

{mul\_line\_comment++;}

else

{err\_comment=1;}

}

"\*/" { cc\_comment++;

if(mul\_line\_comment==1)

{mul\_line\_comment--;}

else

{err\_comment=1;}

}

{operator} {

printf("\n\t%-30s is an Operator\n",yytext);

int key=hash\_func(yytext);

if(searchInHash(key,yytext)==0)

insertToHash(key,yytext,"operator");

}

{alpha}({alpha}|{digit})\* {

printf("\n\t%-30s is an Identifier\n",yytext);

int key=hash\_func(yytext);

if(searchInHash(key,yytext)==0)

insertToHash(key,yytext,"identifier");

}

{digit} {

printf("\n\t%-30s is a Constant\n",yytext);

int key=hash\_func(yytext);

if(searchInHash(key,yytext)==0)

insertToHash(key,yytext,"constant");

}

\"(\\.|[^"\\])\*\" {

printf("\n\t%-30s is a String\n",yytext);

int key=hash\_func(yytext);

if(searchInHash(key,yytext)==0)

insertToHash(key,yytext,"string");

}

%%

/\*---------------------------------------------------------------\*/

struct hash \*hashTable = NULL;

int eleCount = 0;

struct node {

char name[20];

char type[20];

int key;

struct node \*next;

};

struct hash {

struct node \*head;

int count;

};

struct node \* createNode(int key, char \*name, char \*type) {

struct node \*newnode;

newnode = (struct node \*) malloc(sizeof(struct node));

newnode->key=key;

strcpy(newnode->name, name);

strcpy(newnode->type, type);

newnode->next = NULL;

return newnode;

}

void insertToHash(int key, char \*name, char \*type) {

int hashIndex = key % eleCount;

struct node \*newnode = createNode(key, name, type);

if (!hashTable[hashIndex].head) {

hashTable[hashIndex].head = newnode;

hashTable[hashIndex].count = 1;

return;

}

newnode->next = (hashTable[hashIndex].head);

hashTable[hashIndex].head = newnode;

hashTable[hashIndex].count++;

return;

}

int searchInHash(int key, char \*name) {

int hashIndex = key % eleCount, flag = 0;

struct node \*myNode;

myNode = hashTable[hashIndex].head;

if (!myNode) {

return flag;

}

while (myNode != NULL) {

if (strcmp(myNode->name,name)==0) {

flag = 1;

break;

}

myNode = myNode->next;

}

return flag;

}

void display() {

struct node \*myNode;

int i;

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

printf("\t|------------------------------------------------------------------------|\n");

printf("\t| | |\n");

printf("\t| Name | Type |\n");

printf("\t|------------------------------------------------------------------------|\n");

printf("\t| | |\n");

for (i = 0; i < eleCount; i++) {

if (hashTable[i].count == 0)

continue;

myNode = hashTable[i].head;

if (!myNode)

continue;

while (myNode != NULL) {

printf("\t|\t%-30s", myNode->name);

printf("\t|\t%-15s\n", myNode->type);

printf("\t|------------------------------------------------------------------------|\n");

myNode = myNode->next;

}

}

return;

}

int hash\_func(char val[20])

{

int sum=0;

for(int i=0; i<strlen(val);i++)

{

sum+=val[i];

}

sum%=50;

return sum;

}

void main(int argc[], char \*\*argv[])

{

//yyin=fopen("test.c","r");

int n;

eleCount=50;

hashTable = (struct hash \*) calloc(50, sizeof(struct hash));

yylex();

printf("\n\n\n\n\t\t\t\t\t ERRORS");

printf("\n\t\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

if(co\_paranthesis!=cc\_paranthesis)

{no\_of\_errors++;

printf("\n\t%d)---------------------UNMATCHED ( !!!-------------\n",no\_of\_errors);}

if(co\_curly!=cc\_curly)

{no\_of\_errors++;

printf("\n\t%d)---------------------UNMATCHED { !!!-------------\n",no\_of\_errors);}

if(co\_square!=cc\_square)

{no\_of\_errors++;

printf("\n\t%d)---------------------UNMATCHED [ !!!-------------\n",no\_of\_errors);}

if(co\_quotes%2!=0)

{no\_of\_errors++;

printf("\n\t%d)---------------------UNMATCHED QUOTES !!!-------------\n",no\_of\_errors);}

if(co\_comment!=cc\_comment)

{no\_of\_errors++;

printf("\n\t%d)---------------------UNMATCHED MULTI\_LINE COMMENT!!!-------------\n",no\_of\_errors);}

if(err\_comment==1)

{no\_of\_errors++;

printf("\n\t%d)---------------------NESTED MULTI\_LINE COMMENT!!!-------------\n",no\_of\_errors);}

if(mismatch\_paranthesis==1)

{no\_of\_errors++;

printf("\n\t%d)---------------------MISMATCHED PARANTHESES!!!-------------\n",no\_of\_errors);}

if(mismatch\_curly==1)

{no\_of\_errors++;

printf("\n\t%d)---------------------MISMATCHED CURLY BRACES!!!-------------\n",no\_of\_errors);}

if(mismatch\_square==1)

{no\_of\_errors++;

printf("\n\t%d)---------------------MISMATCHED SQUARE BRACES!!!-------------\n",no\_of\_errors);}

if(err\_id>0)

{

for (int i =0 ; i< err\_id ; i++)

{

no\_of\_errors++;

printf("\n\t%d)---------------------INVALID IDENTIFIER!!![-%s]-------------\n",no\_of\_errors,err\_id\_str[i]);

}

}

if(no\_of\_errors==0)

printf("\n\tNO LEXICAL ERRORS!");

printf("\n\n\n\n\t\t\t\t\tSYMBOL TABLE");

printf("\n\t\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

display();

printf("%d%d",co\_comment, cc\_comment);

}

int yywrap()

{

return 1;

}

**Regular Expressions**

alpha [a-zA-Z]

digit [0-9]

Pre-processor directive #include[ ]?<{alpha}\*.h>

User defined function {alpha}({alpha}|{digit}|[\_])\*\(({alpha}|{digit}|[\_]|[ ])\*\)

Single line comment "//".\*

Identifier {alpha}({alpha}|{digit})\*

String \"(\\.|[^"\\])\*\"

Invalid-Identifier {digit}+{alpha}+|{alpha}\*{digit}\*@{alpha}\*{digit}\*|%

{alpha}{alpha}+

**Output**

**Test Case-1**

Input:

#include<stdio.h>

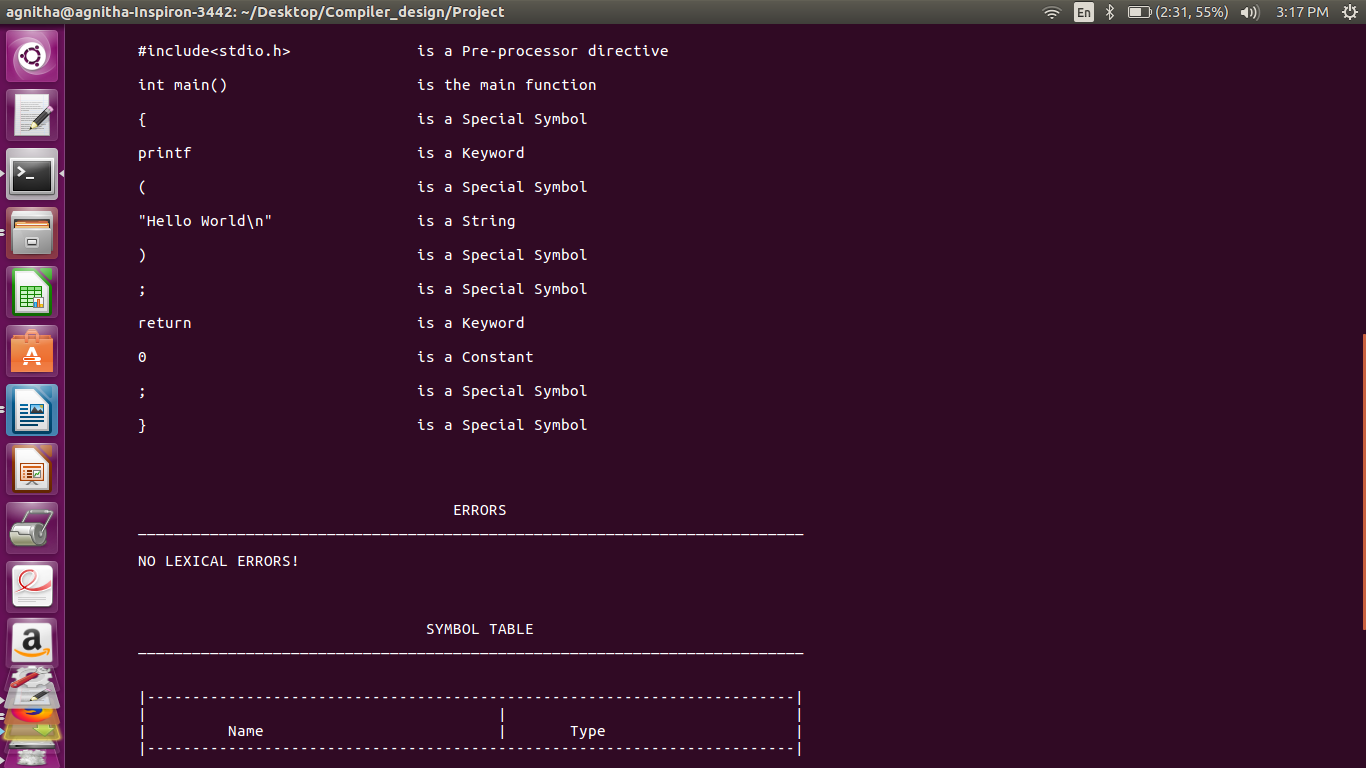
int main()

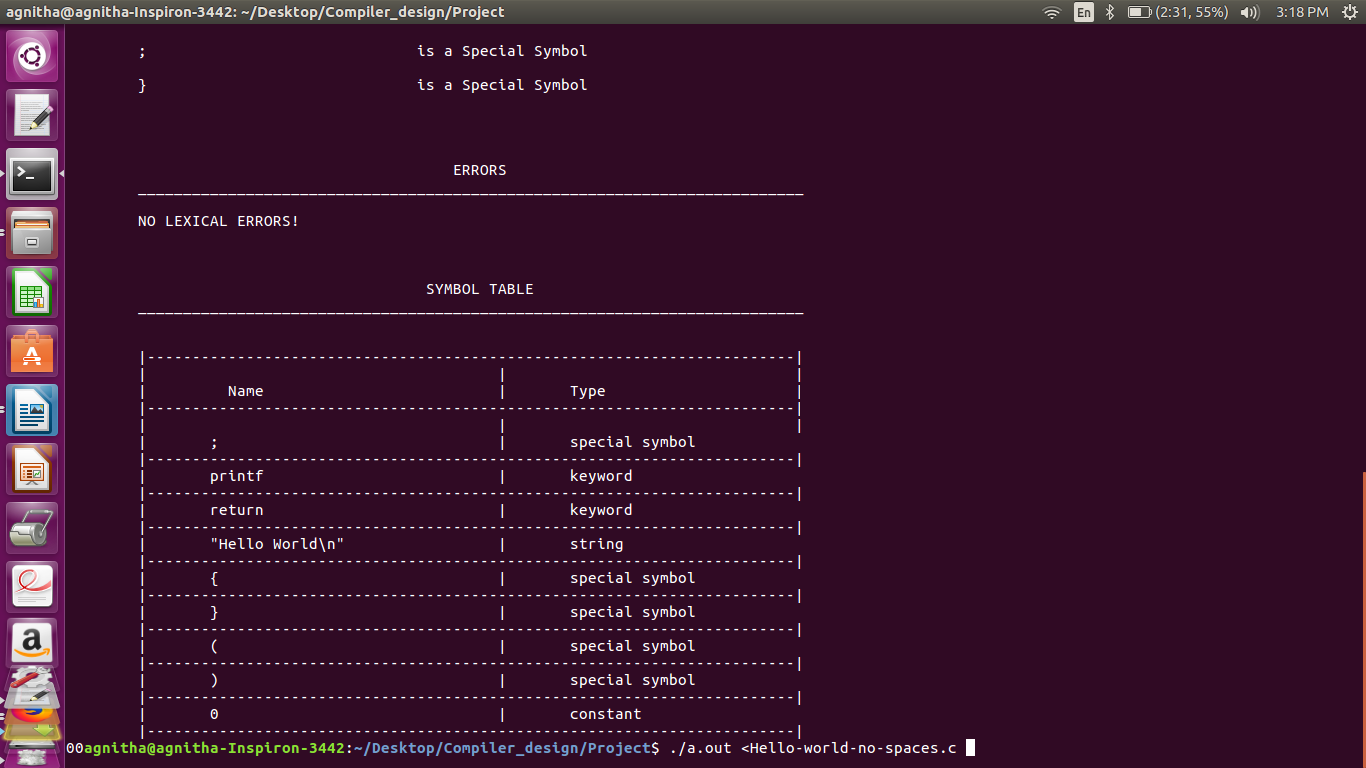
{

printf("Hello World\n");

return 0;

}

****



**Test Case-2**

#include<stdio.h>

void main()

{

int a=5;

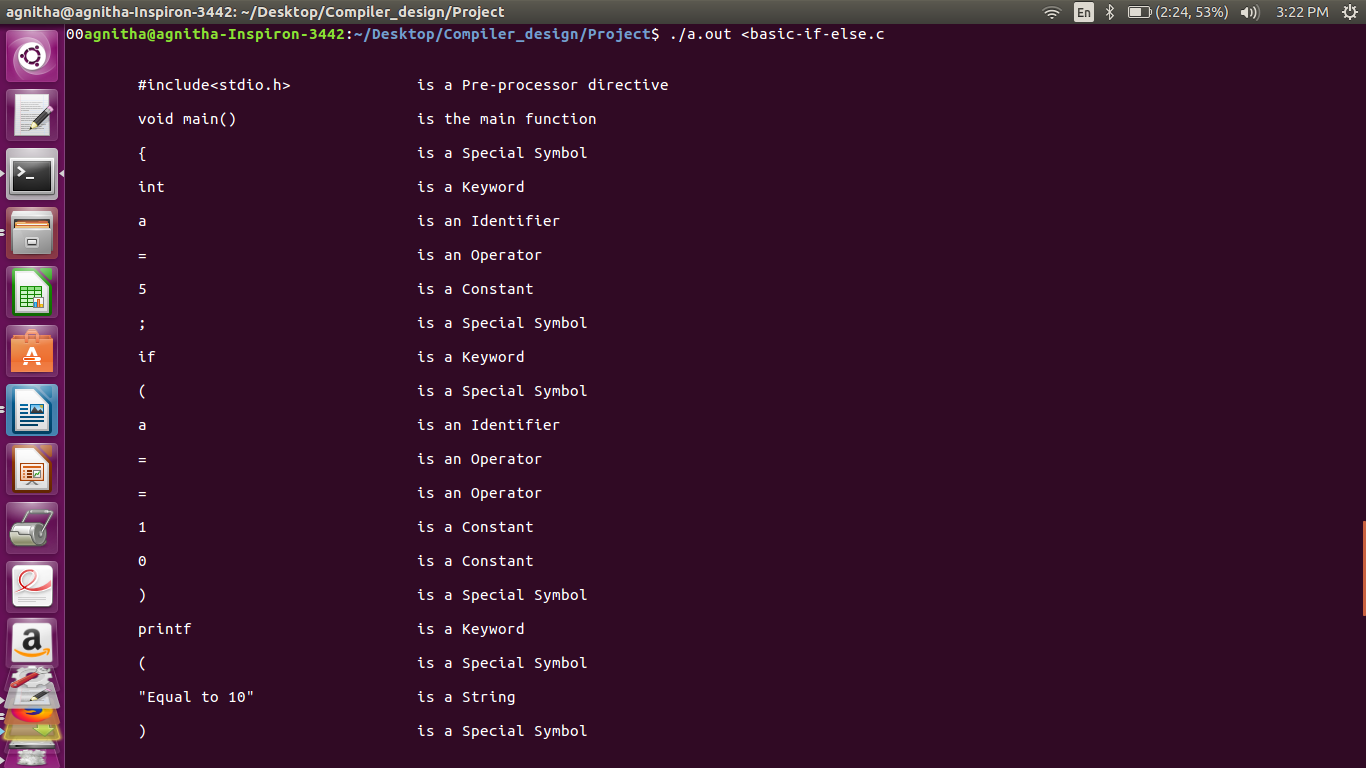
if(a==10)

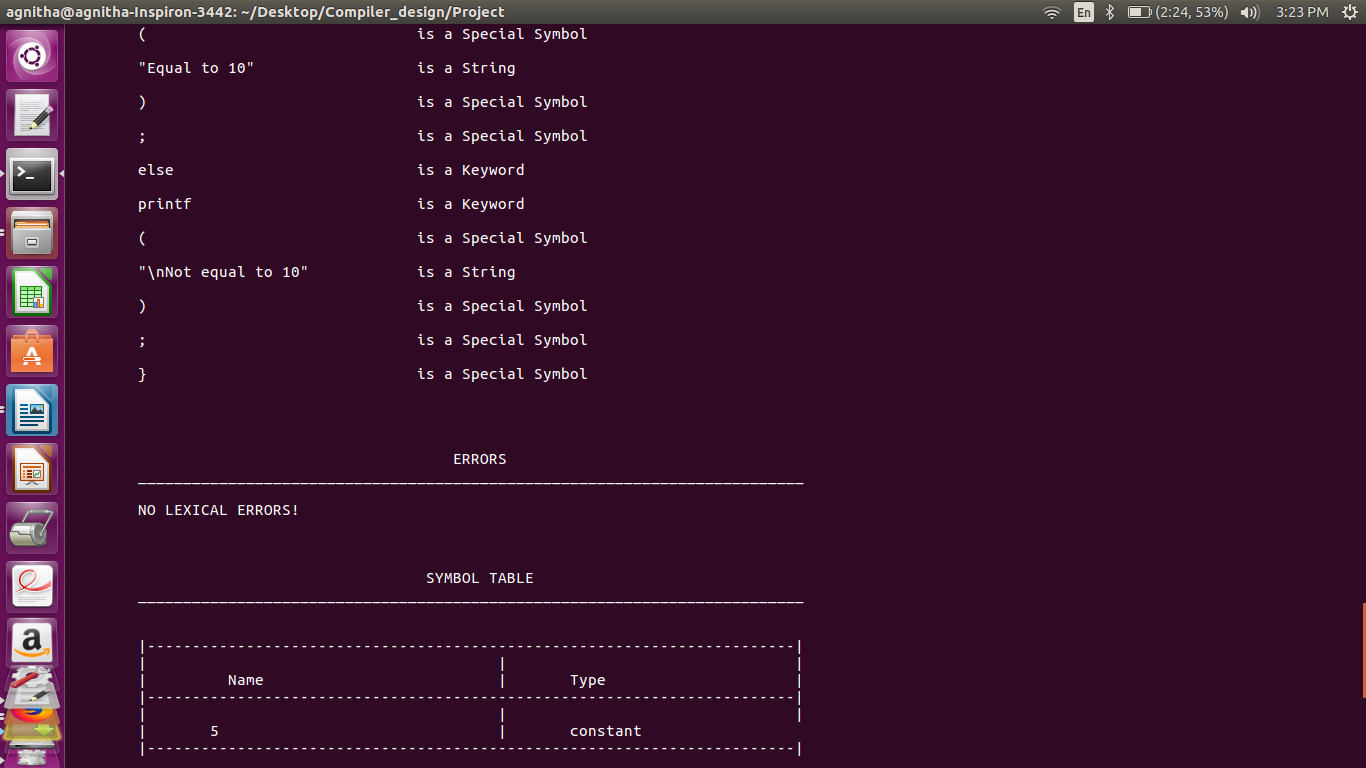
printf("Equal to 10");

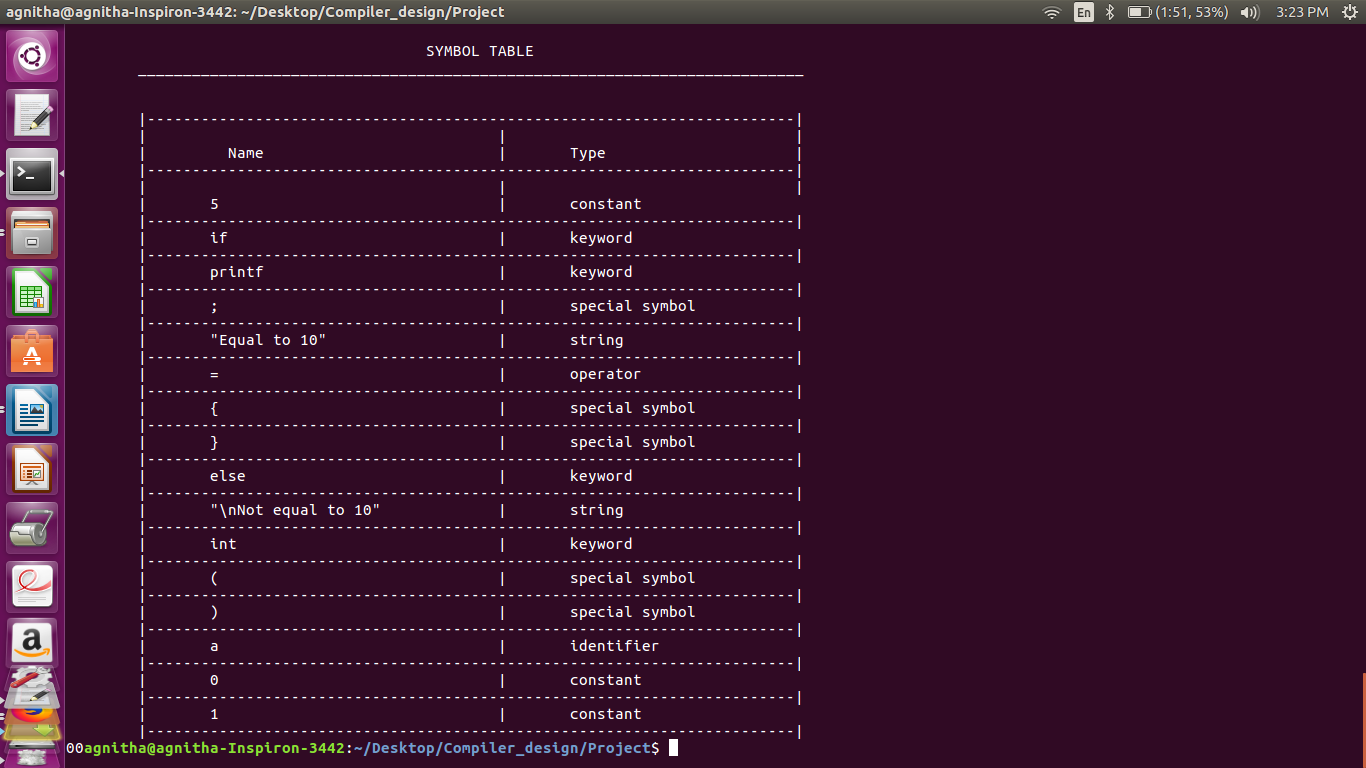
else

printf("\nNot equal to 10");

}







**Test Case-3**

#include<stdio.h>

void main()

{

int a=12;

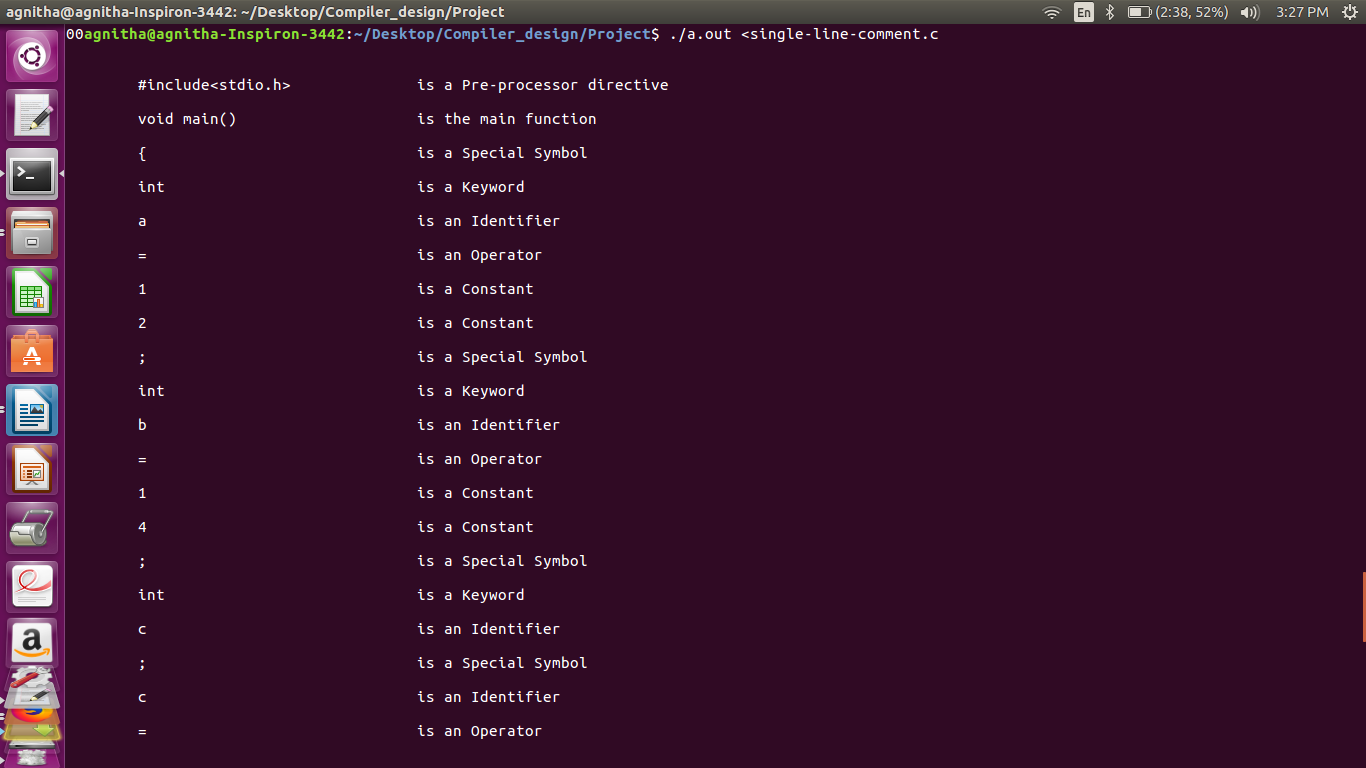
int b=14;

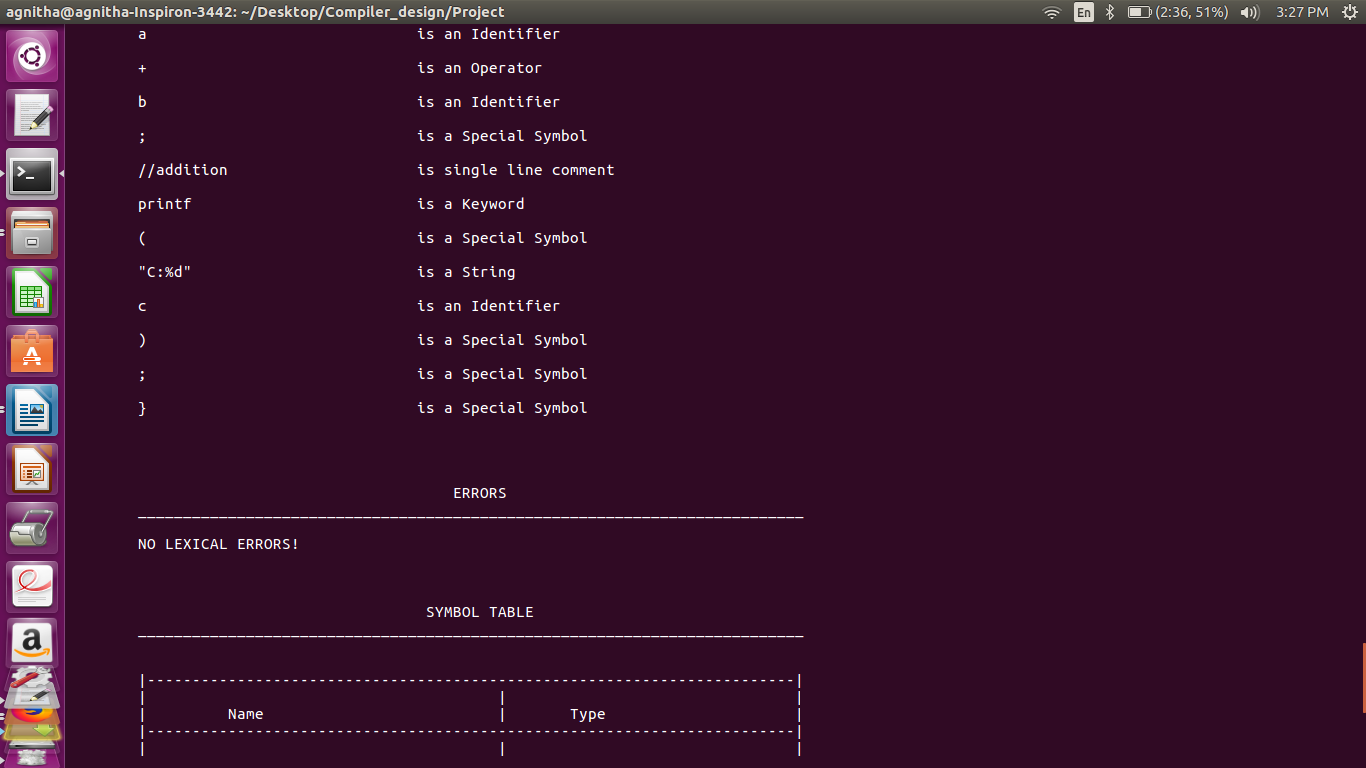
int c;

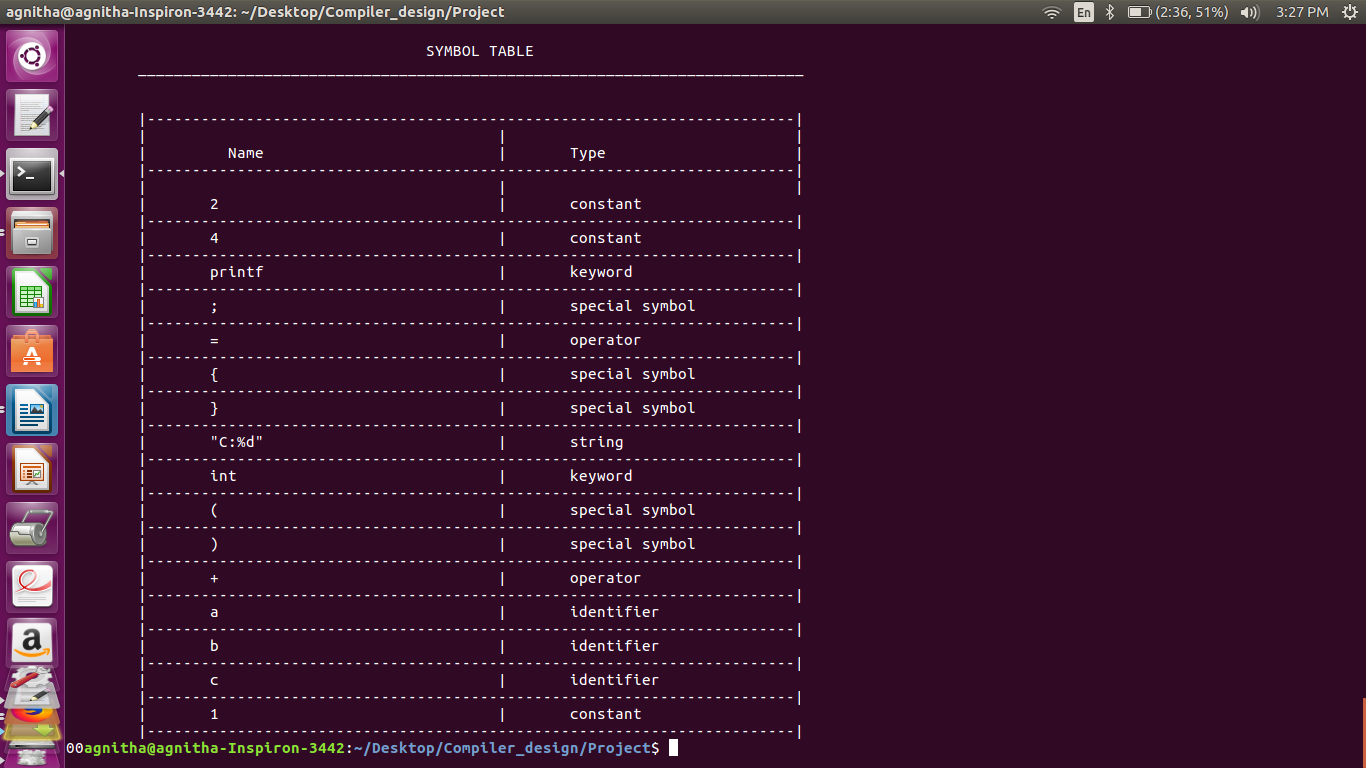
c=a+b; //addition

printf("C:%d",c);

}







**Test Case-4**

/\* This is a simple program

with a multiline comment

\*/

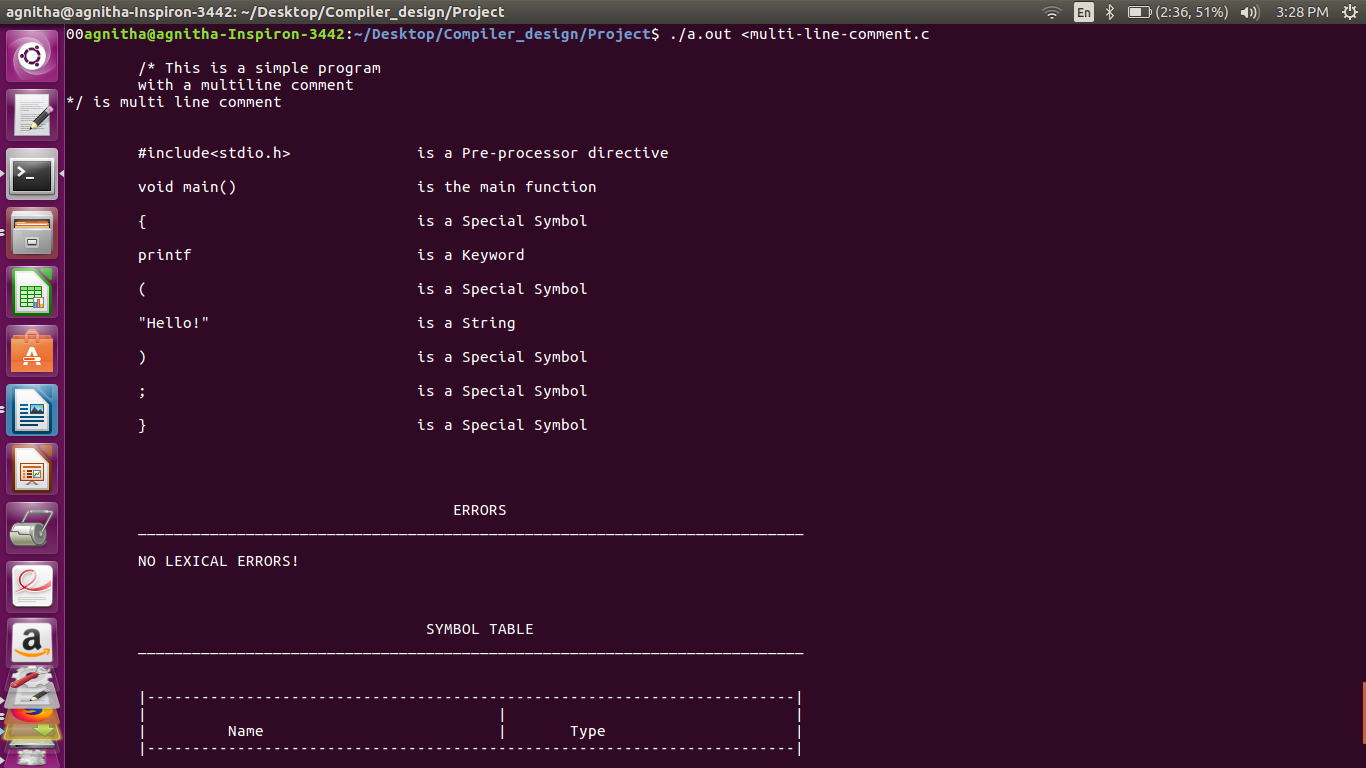
#include<stdio.h>

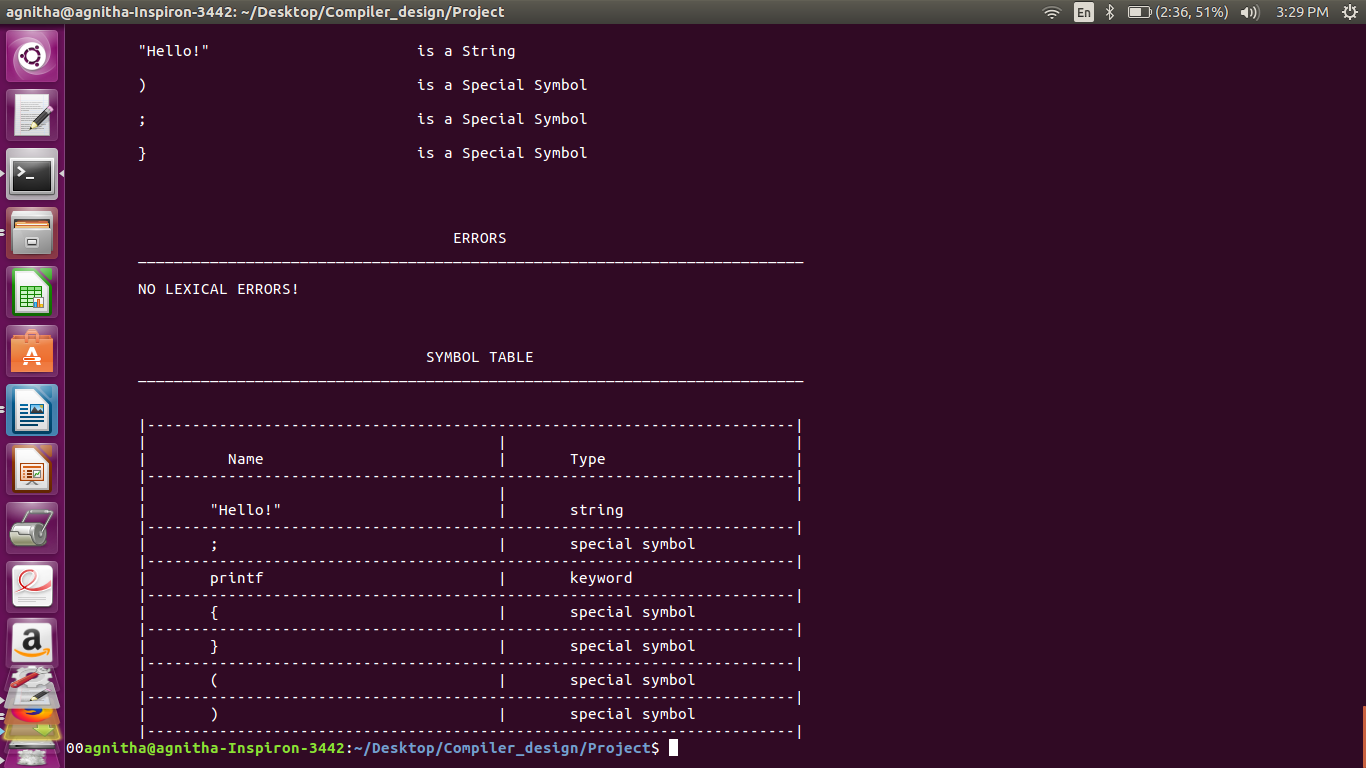
void main()

{

printf("Hello!");

}





**Test Case-5**

#include <stdio.h>

int count=0;

int i, j;

void find\_prime(int n)

{

for(i=2 ; i<=n ; i++)

{

count = 0;

for(j=1 ; j<=i ; j++)

{

if(i%j==0)

count++;

}

if(count == 2)

printf("\n %d is prime" , i);

else

printf("\n %d is composite", i);

}

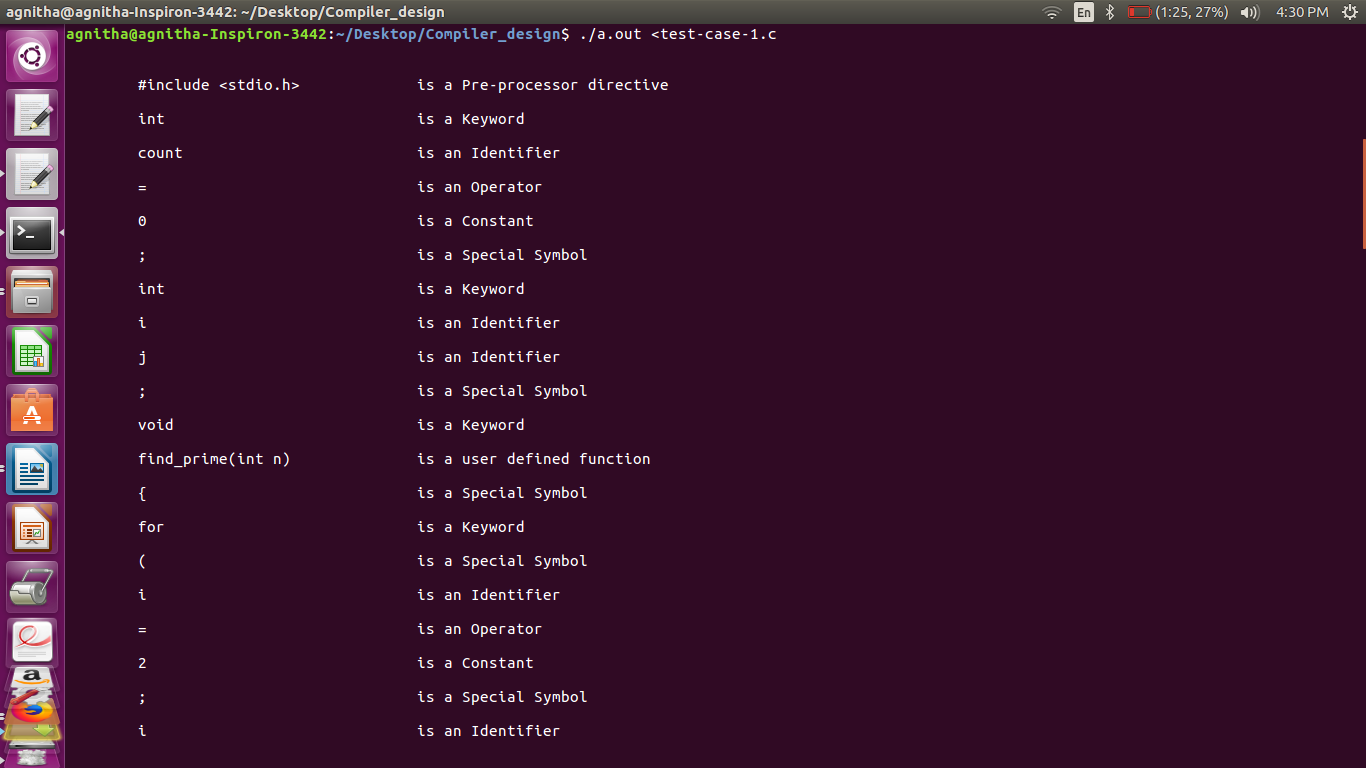
}

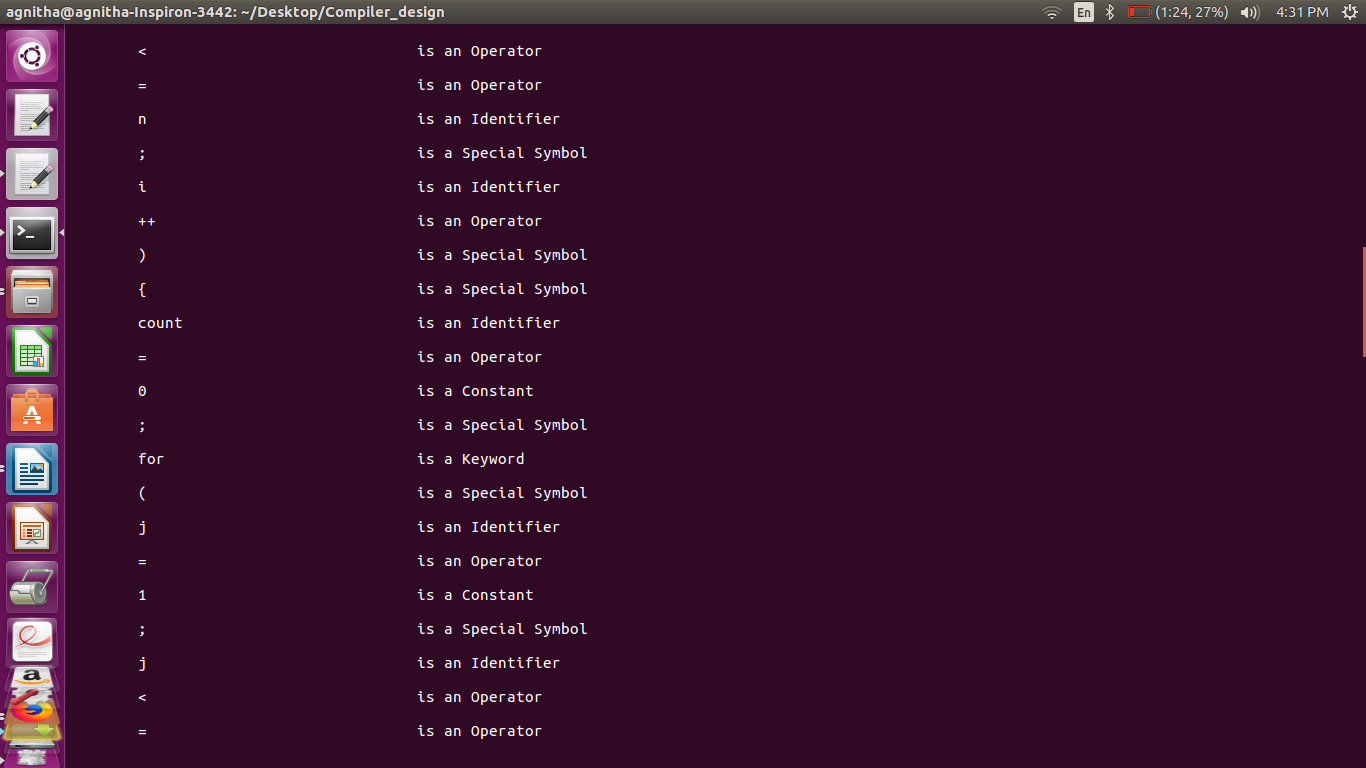
void main()

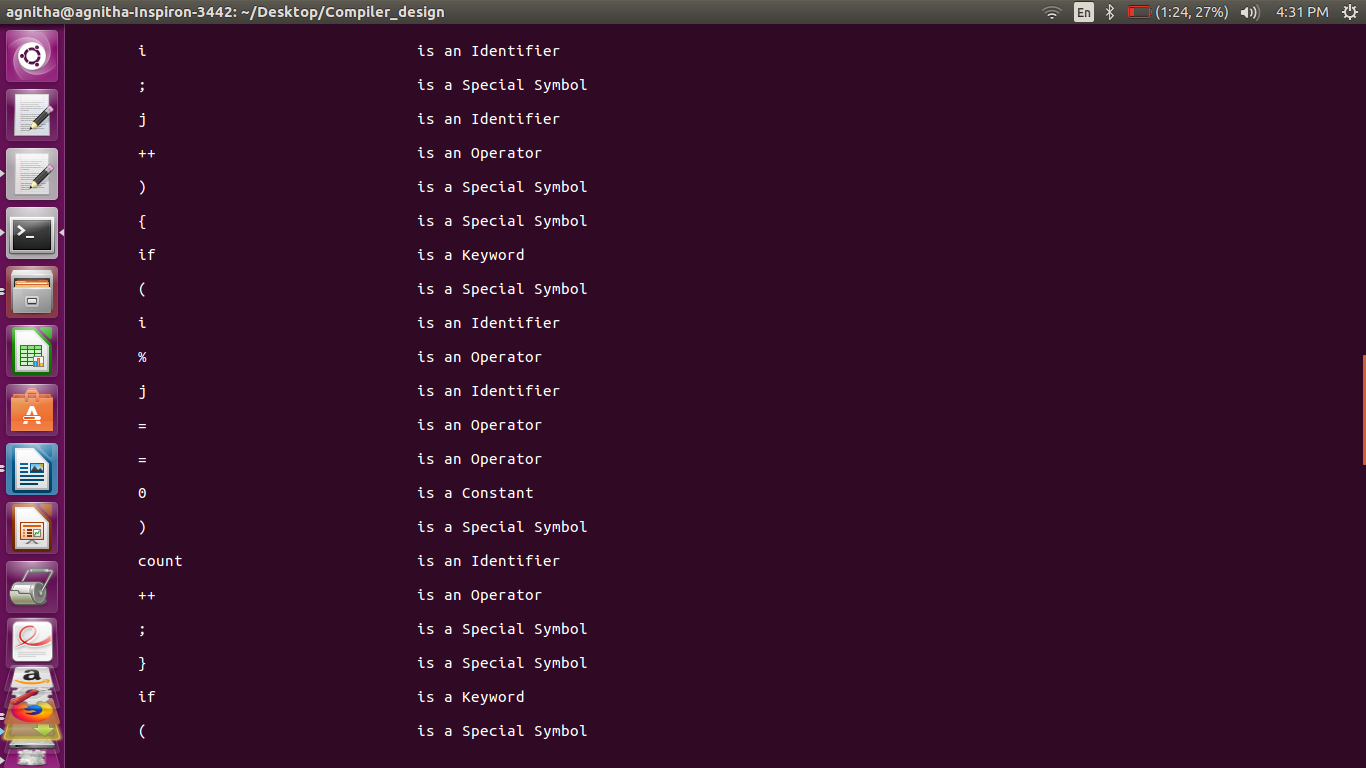
{

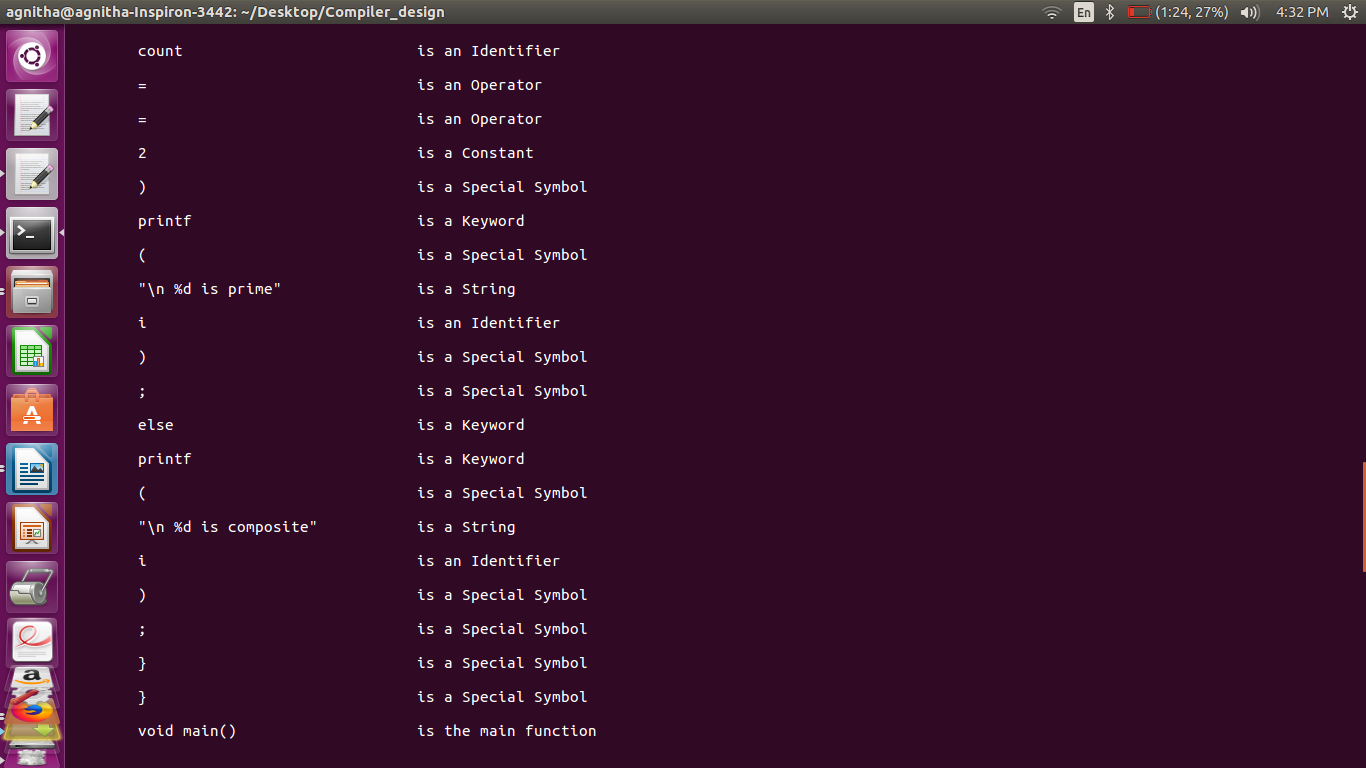
find\_prime(10);

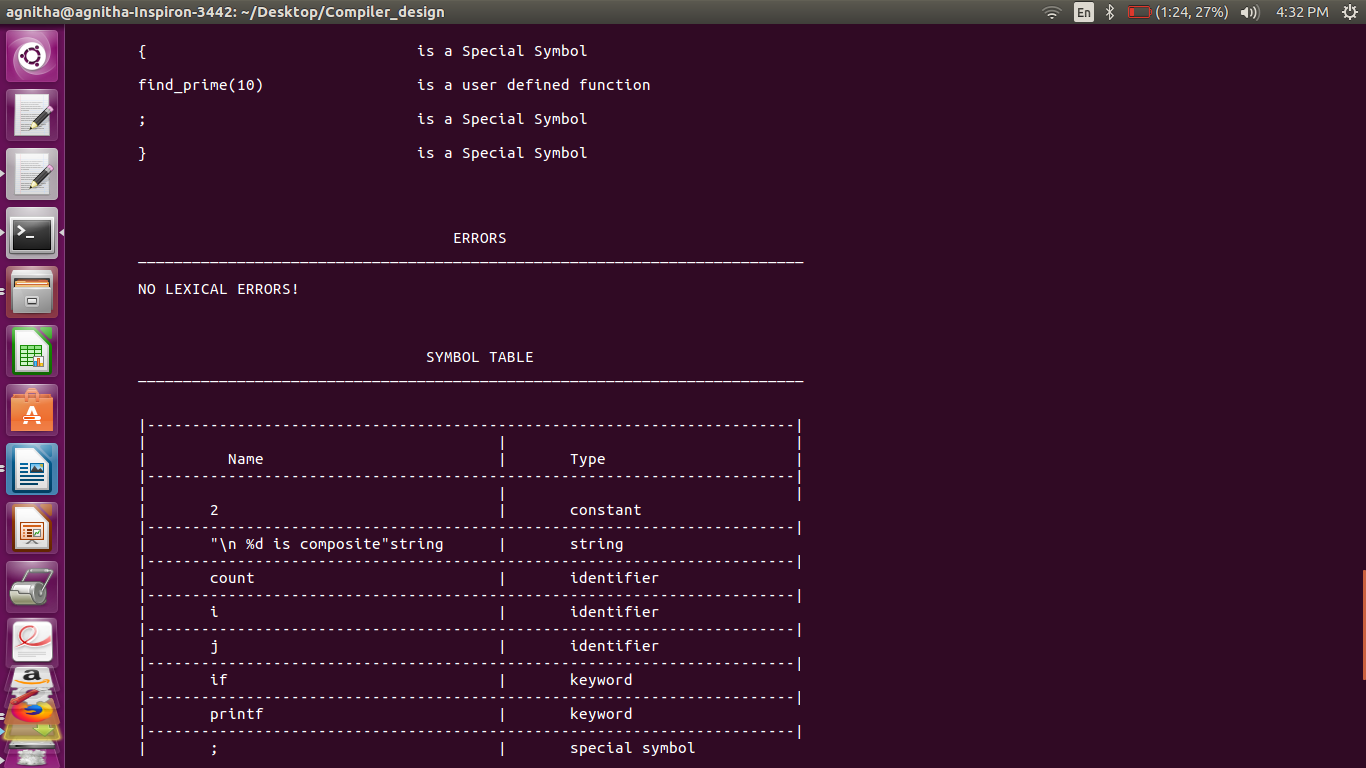
}

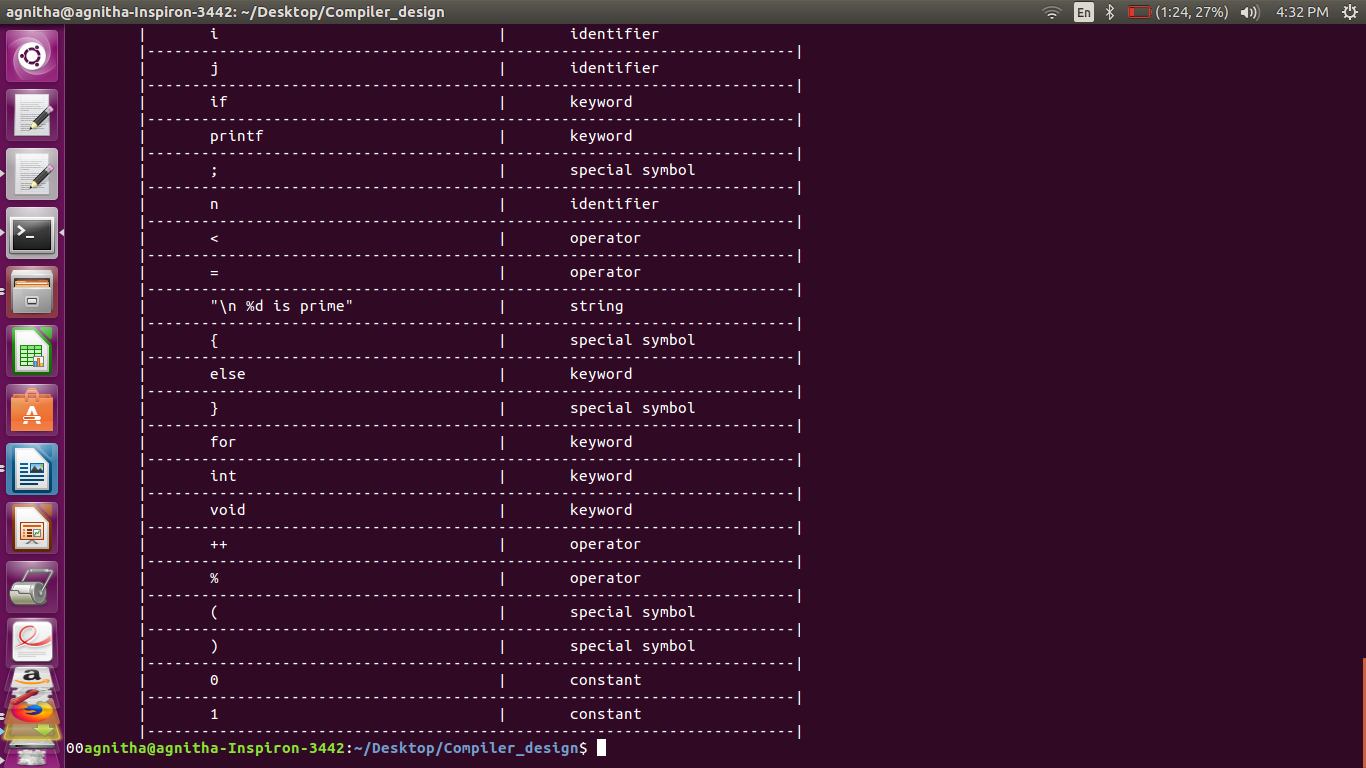












**Test Case-6**

/\*program to calculate sum of first

n positive integers \*/

#include <stdio.h>

void main()

{

int num, i, sum = 0;

printf("Enter a positive integer: ");

scanf("%d", &num);

for(i = 1; i <= num; ++i)

{

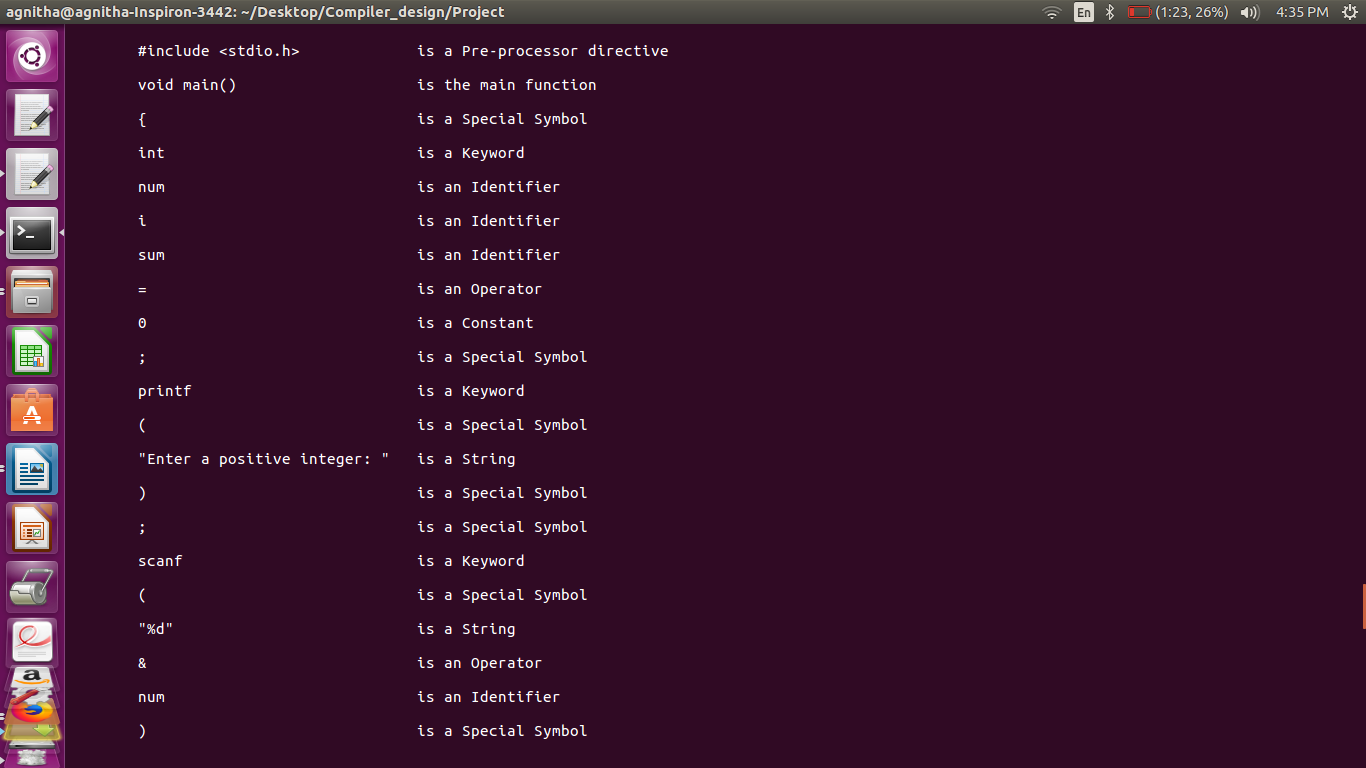
sum += i;

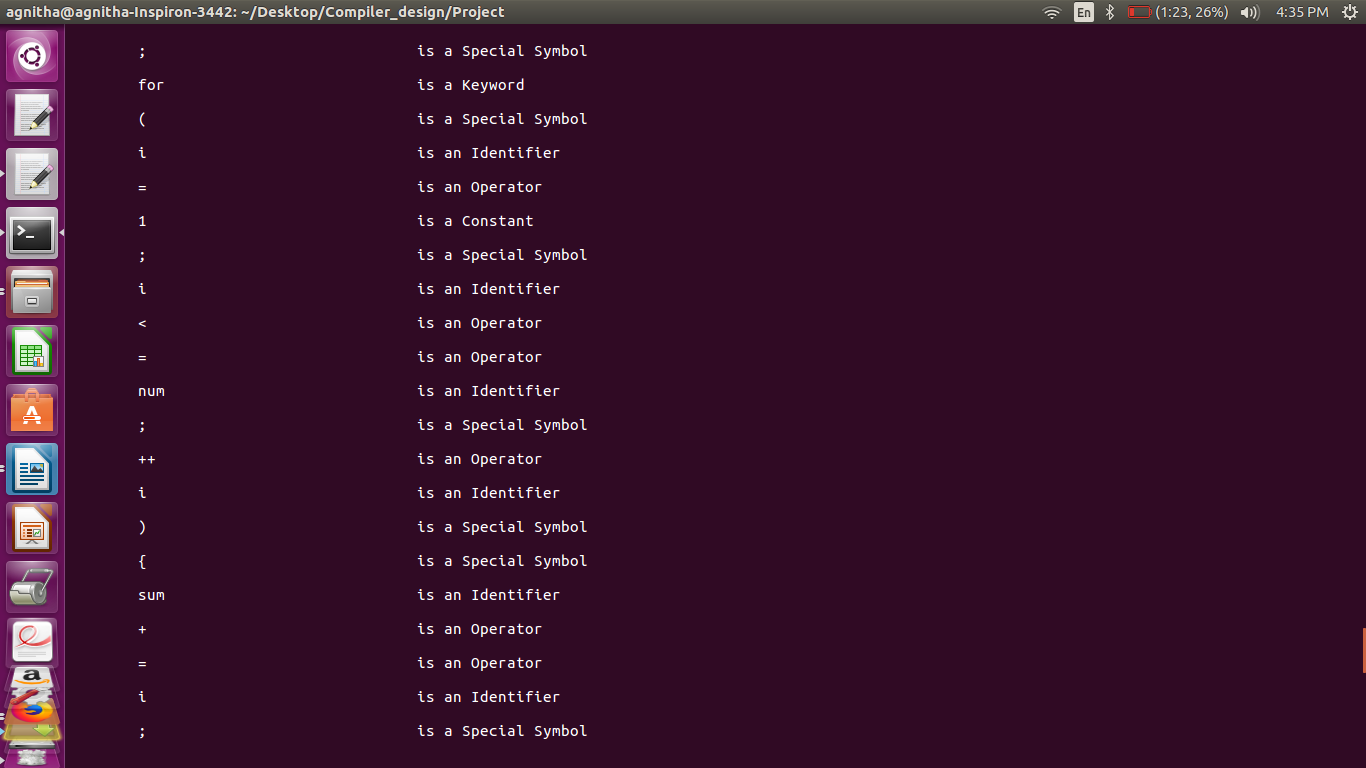
}

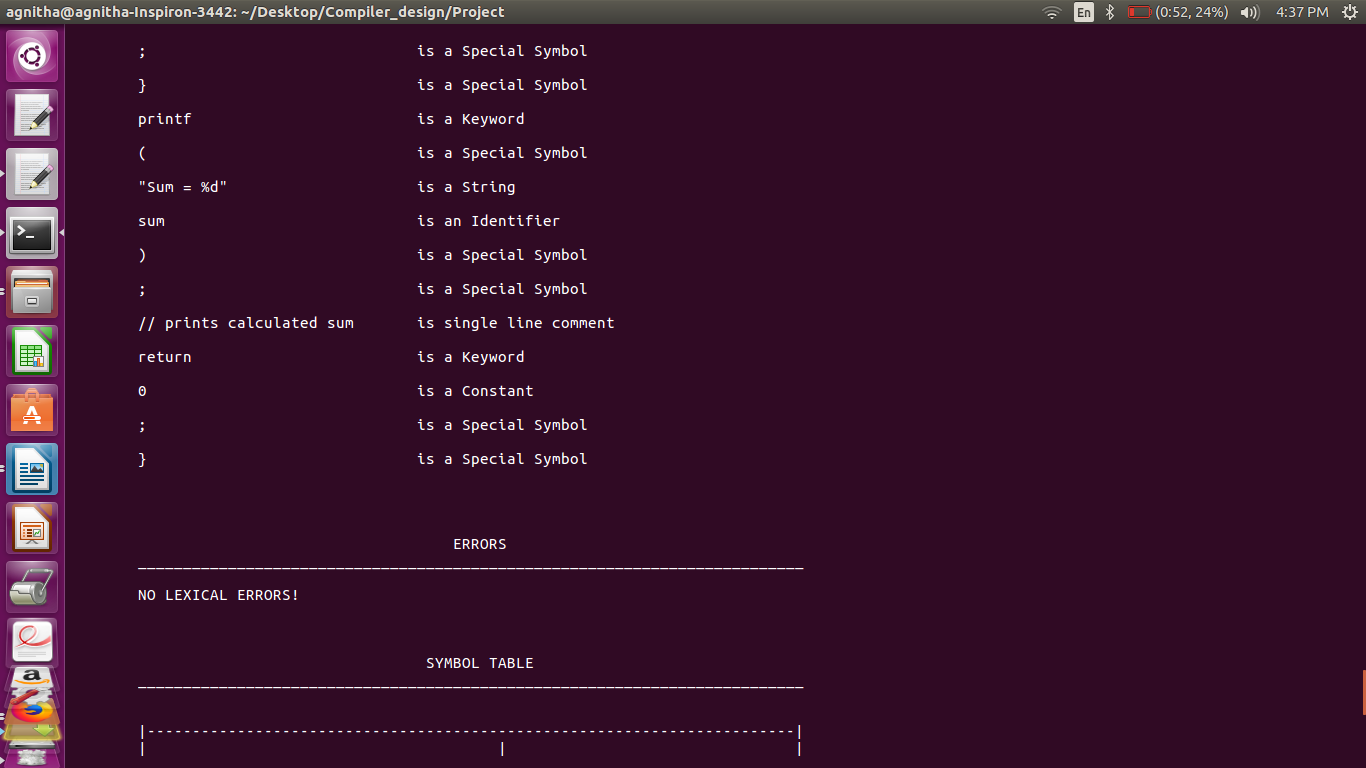
printf("Sum = %d", sum); // prints calculated sum

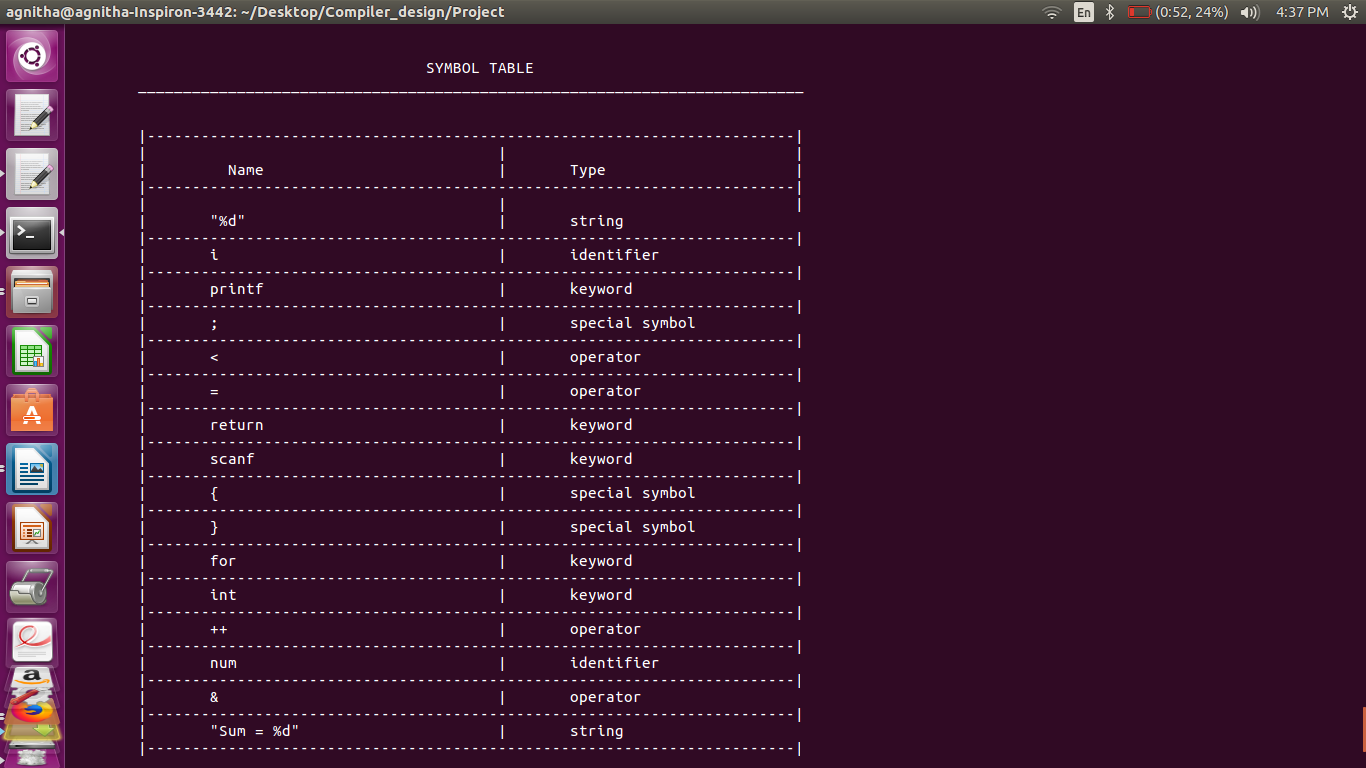
return 0;

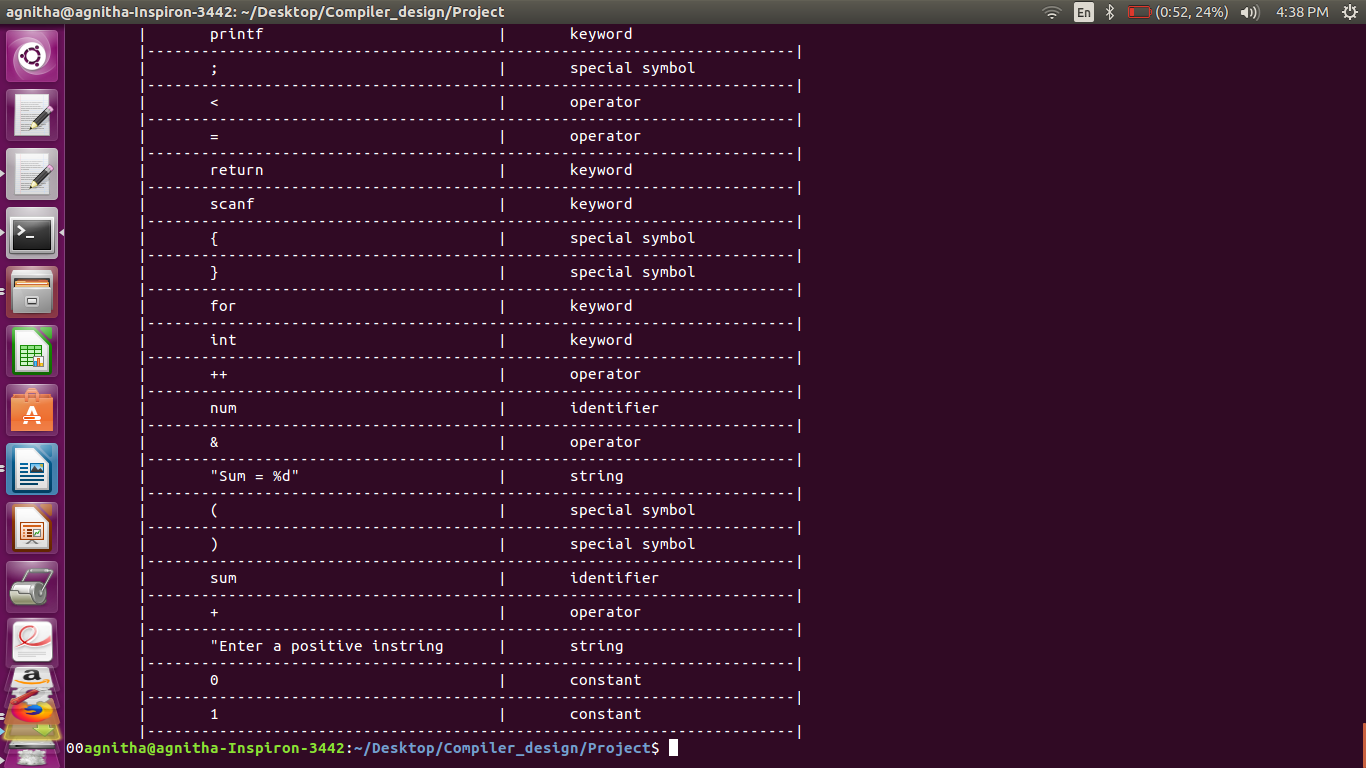
}











**Testcase-7**

#include <stdio.h>

void main)(

{

int a,b; //integer declaration

a=5;

b=8;

int c=a+b;

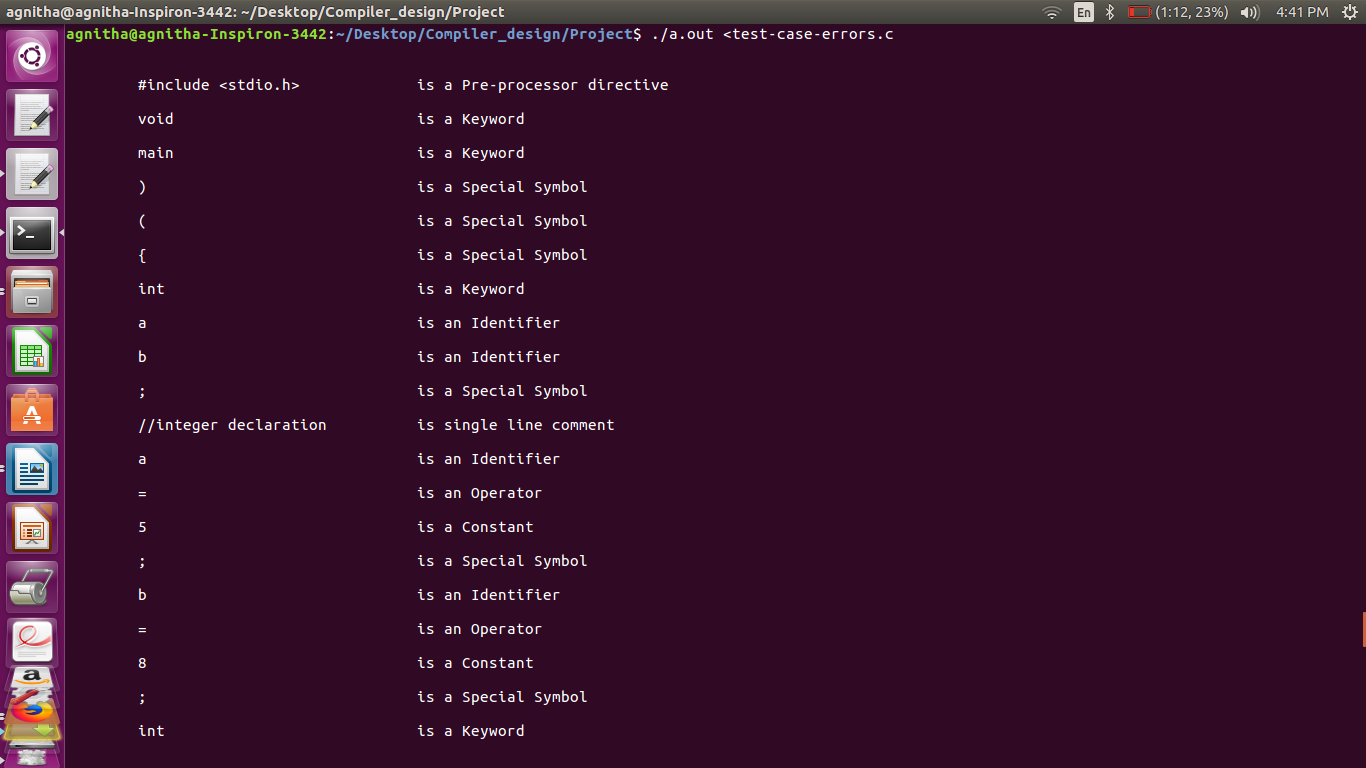
c=a%b;

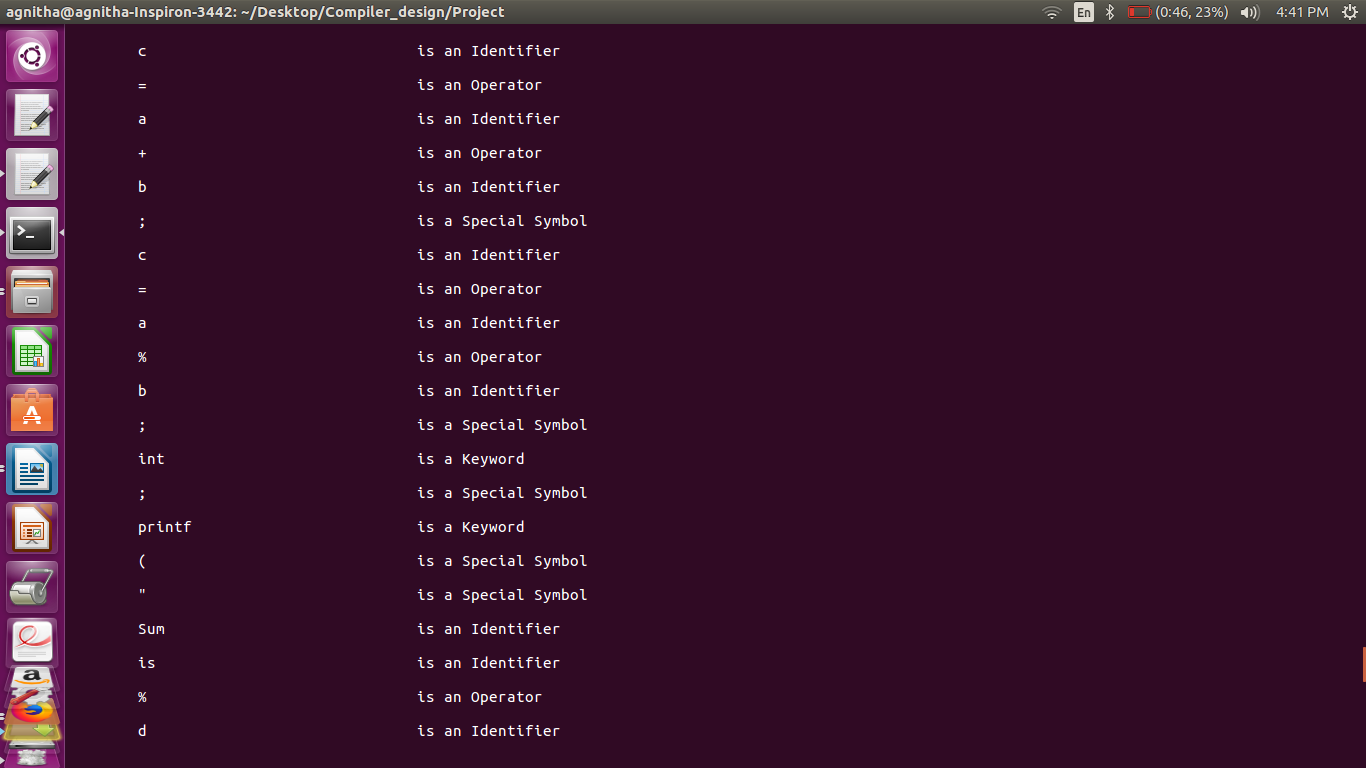
int %han;

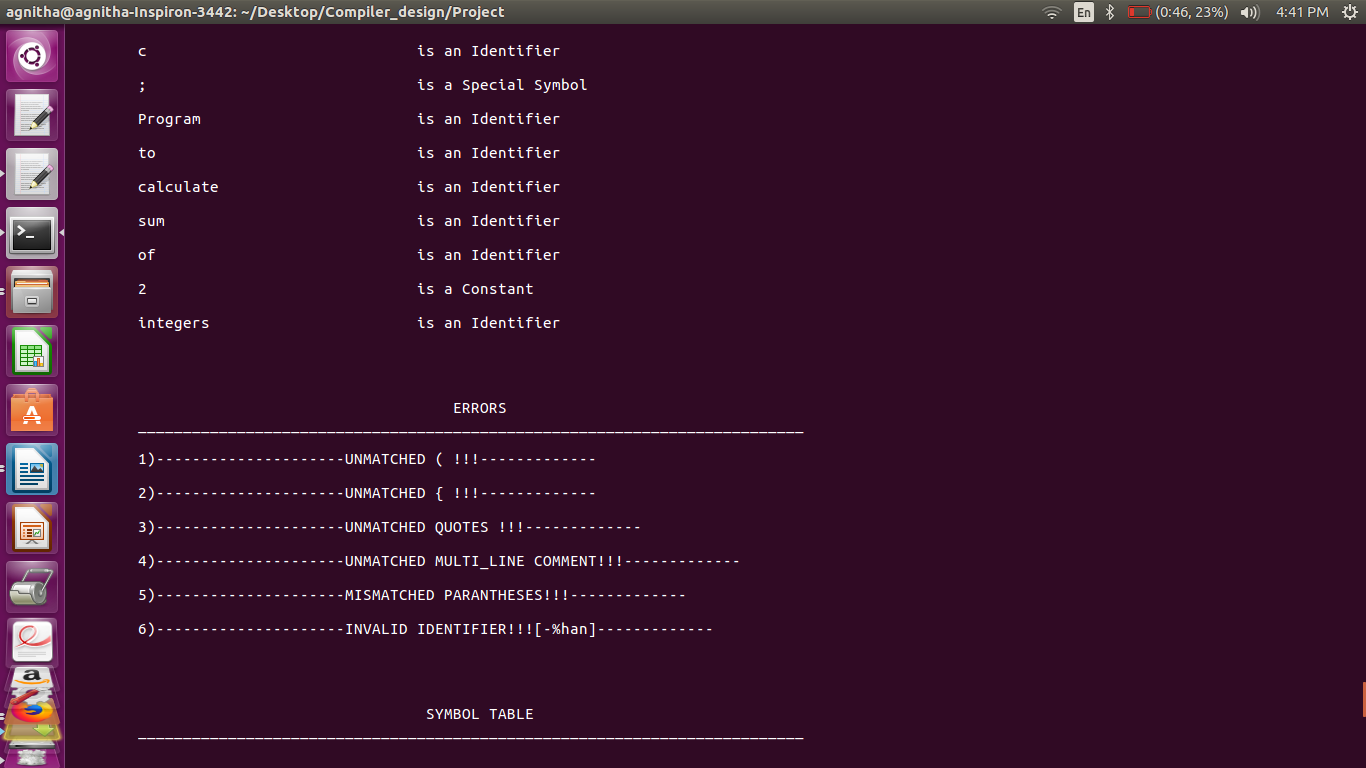
printf("Sum is %d , c ;

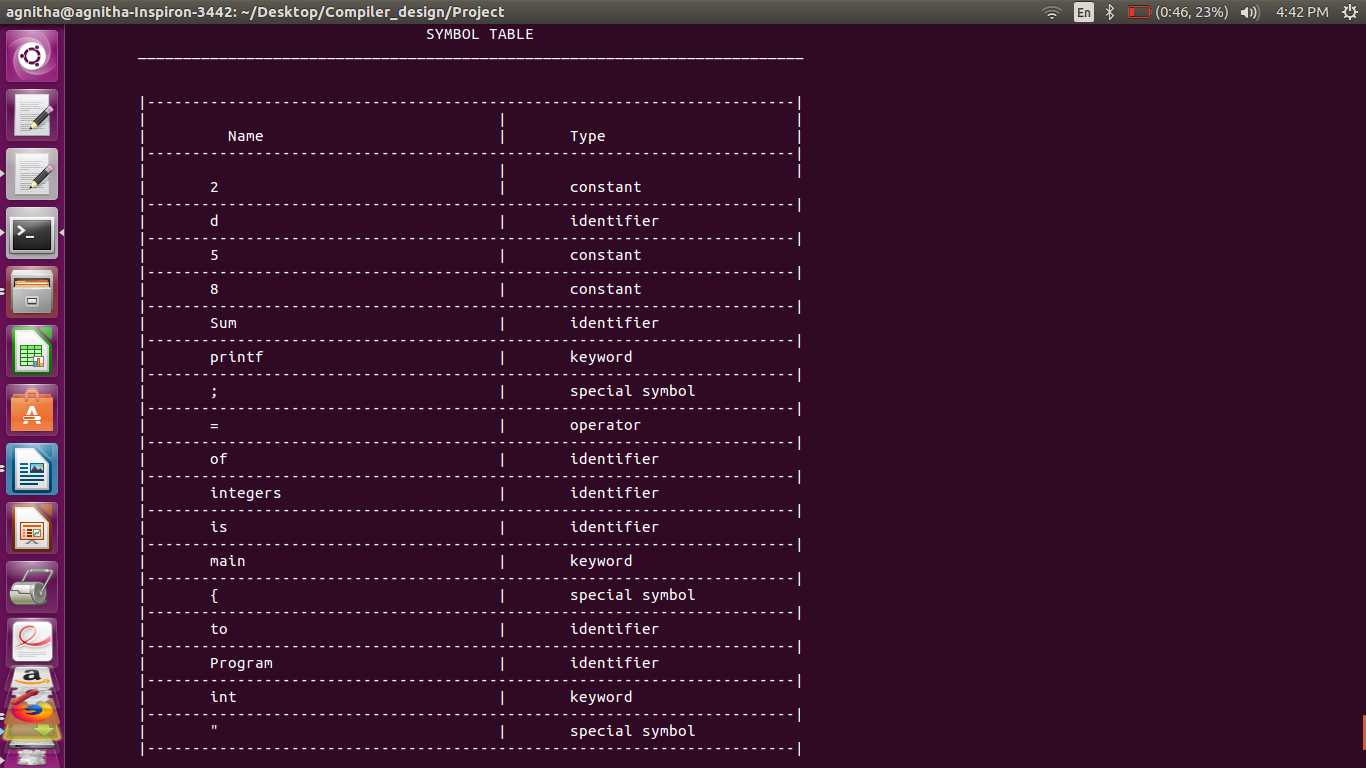
/\*Program to calculate

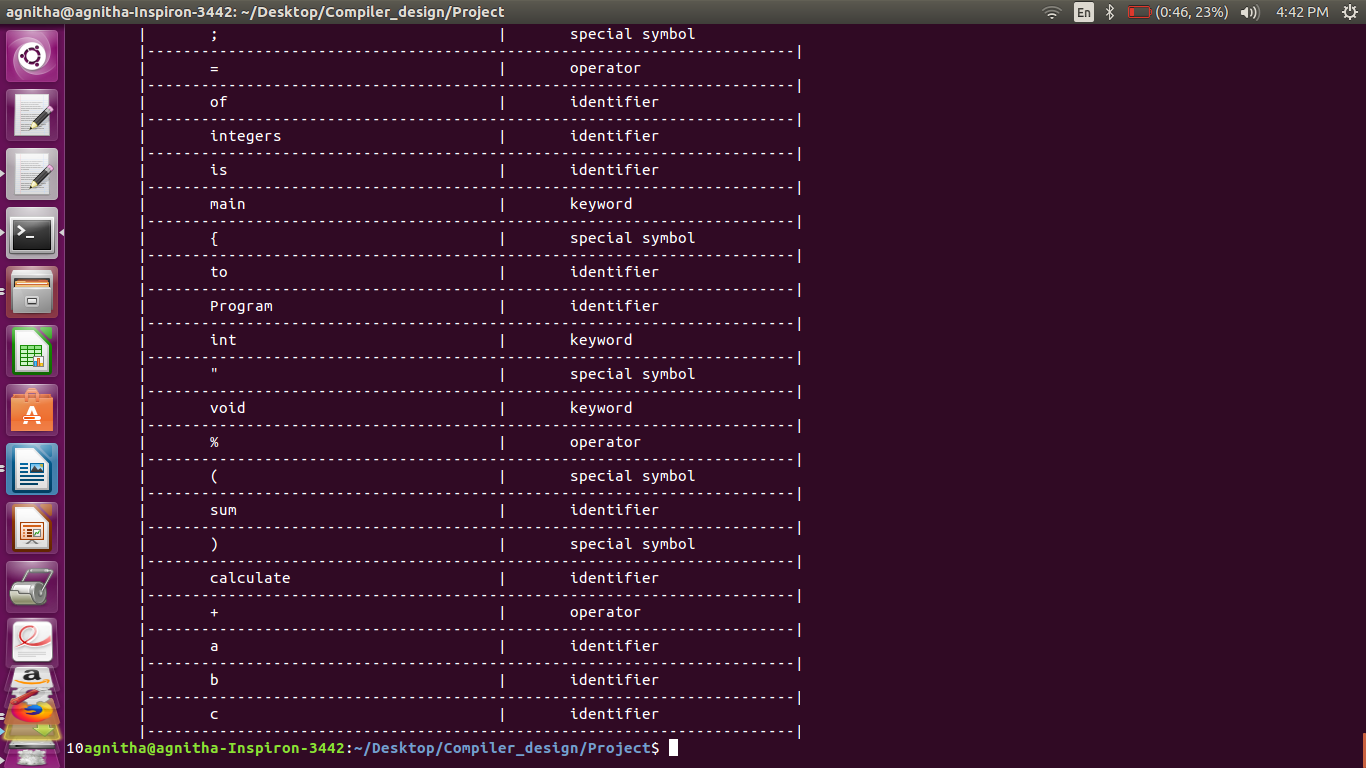
sum of 2 integers











**Description and Status of Test Cases:**

|  |  |  |
| --- | --- | --- |
| TEST CASE | DESCRIPTION | STATUS |
| 1 | Simple Hello World | PASS |
| 2 | Basic-if-else for | PASS |
| 3 | Single-line-comment | PASS |
| 4 | Multi-line-comment | PASS |
| 5 | Nested-for-loop for finding prime numbers within a range | PASS |
| 6 | Simple for loop that does arithmetic operations for calculating the sum-of-first-n-positive numbers | PASS |
| 7 | Program with errors like:   * Unmatched paranthesis * Unmatched curly braces * Unmatched quotes * Unmatched multi line comment * Mismatched paranthesis * Invalid identifier | PASS |

**Conclusion and Future Work**

The project aimed to build a lexical analyzer which converted source program in high level language to a sequence of tokens.

It took care of lexical errors and used hashing concepts to optimize the storage of tokens.

**Future work:**

The program uses a very primitive and crude way to accept nested comments. We plan on optimizing by doing more research.

**References**

* Compilers – Principles , Techniques and tools by Alfred V. Aho , Monica S. Larn , Ravi Sethi , Jeffrey D. Ullman
* Lex and Yacc by John R. Levine , Tony Mason , Doug Brown
* [https://en.wikipedia.org/wiki/](https://en.wikipedia.org/wiki/Symbol_table)
* [https://www.tutorialspoint.](https://www.tutorialspoint.com/data_structures_algorithms/hash_data_structure.htm)