**COMPILER DESIGN LAB**

**PROJECT 1**

**LEXICAL ANALYSER**

**PROJECT REPORT**

**Submitted by:**

**Amitkumar Patel 15CO205**

**Aswin Manoj 15CO209**

**INTRODUCTION**

A compiler is a software program that transforms high-level source code that is written by a developer in a high-level programming language into a low level object code (binary code) in machine language, which can be understood by the processor. The process of converting high-level programming into machine language is known as compilation. The processor executes object code, which indicates when binary high and low signals are required in the arithmetic logic unit of the processor.

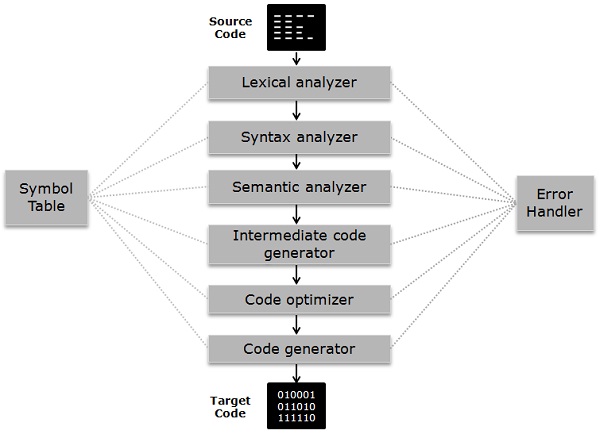
There are basically two phases of compilers, namely Analysis phase and Synthesis phase. Analysis phase creates an intermediate representation from the given source code. Synthesis phase creates an equivalent target program from the intermediate representation.

The Analysis phase or front-end of a compiler can be characterised by four machine-independent stages:

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Intermediate Code Generation

While the Synthesis phase is comprised of two highly machine dependent phases:

1. Code Optimization
2. Code Generation



**ANALYSIS PHASE OF A COMPILER**

1. **Lexical Analysis**

Lexical Analysis or Linear Analysis or Scanning, in which the stream of characters making up the source program is read from left-to-right and grouped into tokens, sequence of characters having a collective meaning. The blanks separating the characters of the tokens and the comments statements appearing within the program would normally be eliminated during lexical analysis.

1. **Syntax Analysis**

Syntax Analysis or Hierarchical Analysis, in which characters or tokens are grouped hierarchically into nested collections with collective meaning. Hierarchical analysis also termed as Parsing, involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output. Usually, the grammatical phrases of the source program are represented by a Parse tree.

1. **Semantic Analysis**

Semantic Analysis, in which certain checks are performed to ensure that the components of a program fit together meaningfully. The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase. It uses hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements.

1. **Intermediate Code Generation**

It involves the generation of intermediate code, in a form which can be readily executed by the machine. There are many popular intermediate codes. Example – Three address code etc. Intermediate code is converted to machine language using the last two phases which are platform dependent.

Till intermediate code, it is same for every compiler out there, but after that, it depends on the platform. To build a new compiler we don’t need to build it from scratch. We can take the intermediate code from the already existing compiler and build the last two parts.

**SYNTHESIS PHASE OF A COMPILER**

1. **Code Optimization**

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

1. **Code Generation**

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

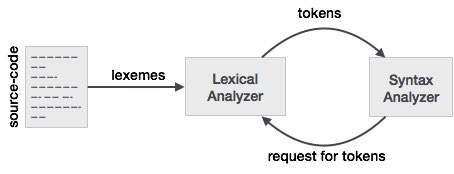
**SYMBOL TABLE**

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

**LEXICAL ANALYSER**

Lexical analysis is the first phase of a compiler. It takes the modified source code from language pre-processors that are written in the form of sentences. The lexical analyser breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyser finds a token invalid, it generates an error. The lexical analyser works closely with the syntax analyser. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyser when it demands.



## **Tokens**

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

## **Longest Match Rule**

When the lexical analyzer read the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed.

**DESIGN AND IMPLEMENTATION**

1. SYSTEM REQUIREMENTS

The system should have following specifications:

* gcc compiler for compiling the C program.
* Lex as the lexical analysis tool.

1. MODULES DESCRIPTION

Our lexical analyser project contains two files. First lex file for parsing the given input code as a file and dividing it into tokens which we have defined in our code. We parse the input code for the following tokens:

* Keywords
* Identifiers
* Strings or literals
* Integer and floating point constants
* Header and Define declarations
* Single Line Comments
* Multi-Line Comments
* Functions
* Operators ( Arithmetic, Relational, Assignment, Logical)
* NULL and EOF

The second is a header file for the declaration of structure for the symbol table implementation. It also contains necessary functions for lookup, insertion and displaying of the symbol table generated from the input source code.

**ERROR DETECTION**

Any word or syntax that is not specified in the above list of accepted tokens will appear as errors followed by the text that is responcible for it along with the line no in which it occured

**IMPLEMENTATION**

**Lex file**

%{

#include <stdio.h>

#include <stdlib.h>

#include "mytable.h"

int line = 1;

int ptr = 0;

%}

keywords 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

L [a-z|A-Z]

A [a-zA-Z\_0-9]

W [0-9]+

N [1-9]+

F [0-9]+[.][0-9]+[f]?

identifier [a-z][a-z|A-Z|0-9|\_]\*

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

headers #include<[a-z]+.h>

define #define

%%

"\n" {line++;}

{headers} { printf("%s is a header\n", yytext);}

{define} { printf("%s is #define\n",yytext);}

"//" { printf("Comment encountered.\n"); }

"/\*"([^\*]|\\*+[^.])\*\\*+"/" { printf("Multi-line comment encountered.\n"); }

"main" { printf("Main encountered\n"); }

"printf(" { printf("printf encountered\n"); }

"{" { printf("Open bracket encountered\n"); }

"}" { printf("Closed bracket encountered\n"); }

[" "|";"|"\t"|"\n"|"("|")"|","] { printf("");}

"["|"]" { printf("");}

{keywords}" "{identifier}[(]([a-z|" "|","|"\*"|]|"[]")\*[)] { printf("%s is a function\n"); }

{strings} { printf("%s is a string\n",yytext);}

{keywords} { printf("%s is a keyword\n", yytext);

insert(yytext,2);}

{identifier} { insert(yytext,1);

printf("%s Identifier encountered\n",yytext);

}

[==|>=|<=|>|<|~|!=] { printf("%s is a logical operator\n",yytext);

insert(yytext,4);}

[=|\*|+|\-|/|%|^|&|'|\||/|\\|.] { printf("%s is an operator\n",yytext);

insert(yytext,4);}

{W}|{N} { printf("%s is an Integer constant\n", yytext);

insert(yytext,3);}

{F} { printf("%s is a float constant\n", yytext);

insert(yytext,3);}

"NULL" { printf("NULL encountered\n"); }

"EOF" { printf("EOF encountered\n"); }

<\*>.|\n { printf("Error in %s at line %d\n", yytext,line);}

%%

int main(){

yyin = fopen("comment-error.c","r");

yylex();

prin();

return 0;

}

int yywrap(){

return 1;

}

**Symbol table header**

|  |
| --- |
| #define NSYMS 100 |
|  |

|  |
| --- |
| #include<string.h> |
|  |

|  |
| --- |
| struct symtab { |
|  |

|  |
| --- |
| char name[10]; |
|  |

|  |
| --- |
| int value; |
|  |

|  |
| --- |
| char \*type; |
|  |

|  |
| --- |
| struct symtab \*next; |
|  |

|  |
| --- |
| }; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| struct symtab \*symlook(); |
|  |

|  |
| --- |
| struct symtab \*table; |
|  |

|  |
| --- |
| void prin() |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| printf("1 identifiers \n2 keywords\n3 constants \n4 operators"); |
|  |

|  |
| --- |
| struct symtab \*s = table; |
|  |

|  |
| --- |
| char map[4][20] = {"Identifier","Keyword","Constants","Operators"}; |
|  |

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

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

|  |
| --- |
| while(s->next!=NULL){ |
|  |

|  |
| --- |
|  |
|  |

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

|  |
| --- |
| printf("%\*c", 15-strlen(s->name), ' '); |
|  |

|  |
| --- |
| printf("%s",map[s->value-1]); |
|  |

|  |
| --- |
| s = s->next; |
|  |

|  |
| --- |
| } |
|  |

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

|  |
| --- |
| printf("%\*c",15-strlen(s->name),' '); |
|  |

|  |
| --- |
| printf("%s\n",map[s->value-1]); |
|  |

|  |
| --- |
| return; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void insert(char \*s,int v){ |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| struct symtab \*p = table; |
|  |

|  |
| --- |
| if(p == NULL) { |
|  |

|  |
| --- |
| p = malloc(sizeof(struct symtab)); |
|  |

|  |
| --- |
| int i; |
|  |

|  |
| --- |
| strcpy(p->name,s); |
|  |

|  |
| --- |
| p->value = v; |
|  |

|  |
| --- |
| p->next = NULL; |
|  |

|  |
| --- |
| table = p; |
|  |

|  |
| --- |
| return; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| while(1){ |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| if(strcmp(p->name,s)==0){ |
|  |

|  |
| --- |
| return; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| if(p->next == NULL) |
|  |

|  |
| --- |
| break; |
|  |

|  |
| --- |
| p = p->next; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| struct symtab \*temp; |
|  |

|  |
| --- |
| temp = malloc(sizeof(struct symtab)); |
|  |

|  |
| --- |
| strcpy(temp->name,s); |
|  |

|  |
| --- |
| temp->value = v; |
|  |

|  |
| --- |
| temp->next = NULL; |
|  |

|  |
| --- |
| p->next = temp; |
|  |

|  |
| --- |
| return; |
|  |

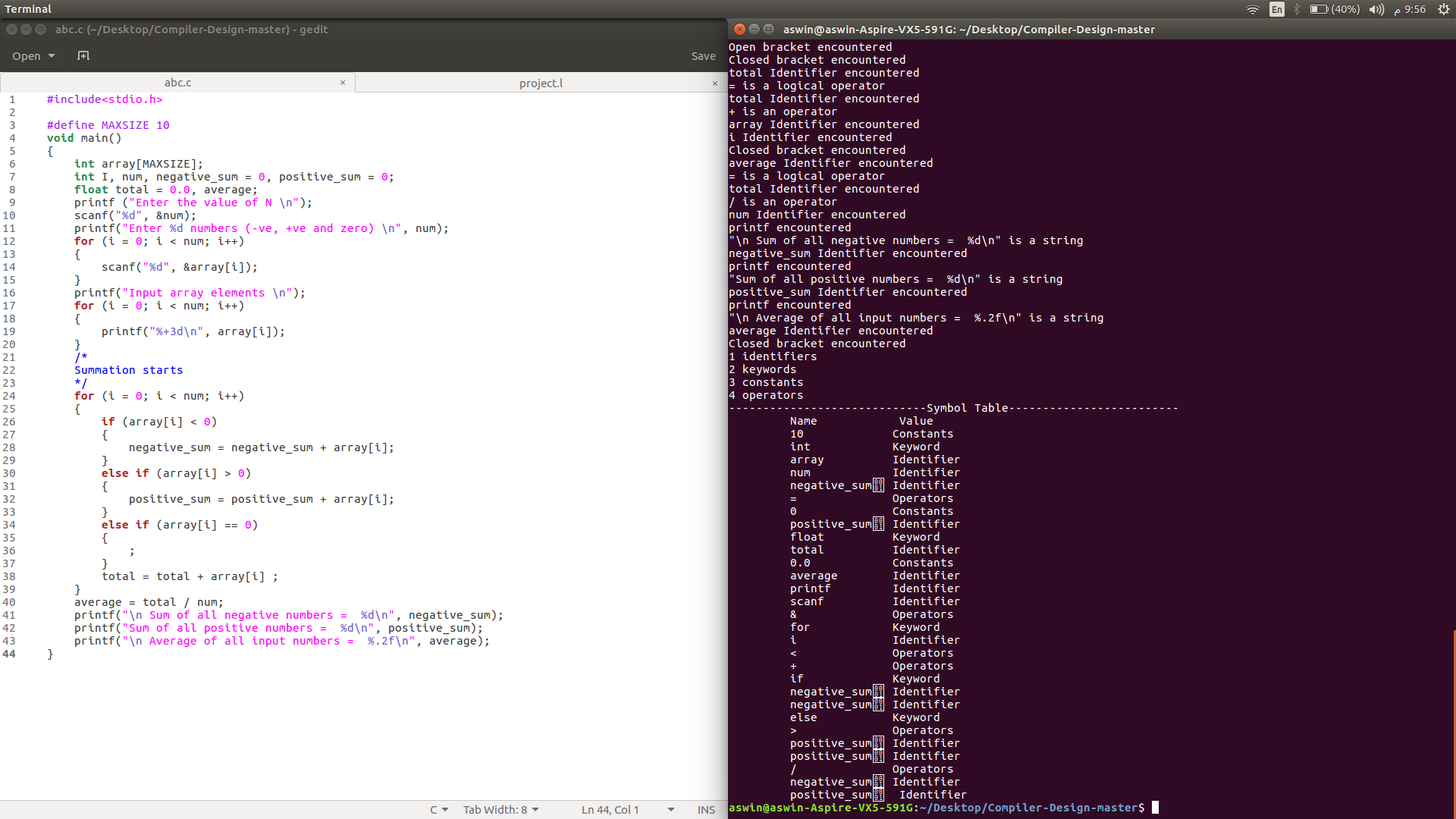
|  |
| --- |
| } |
|  |

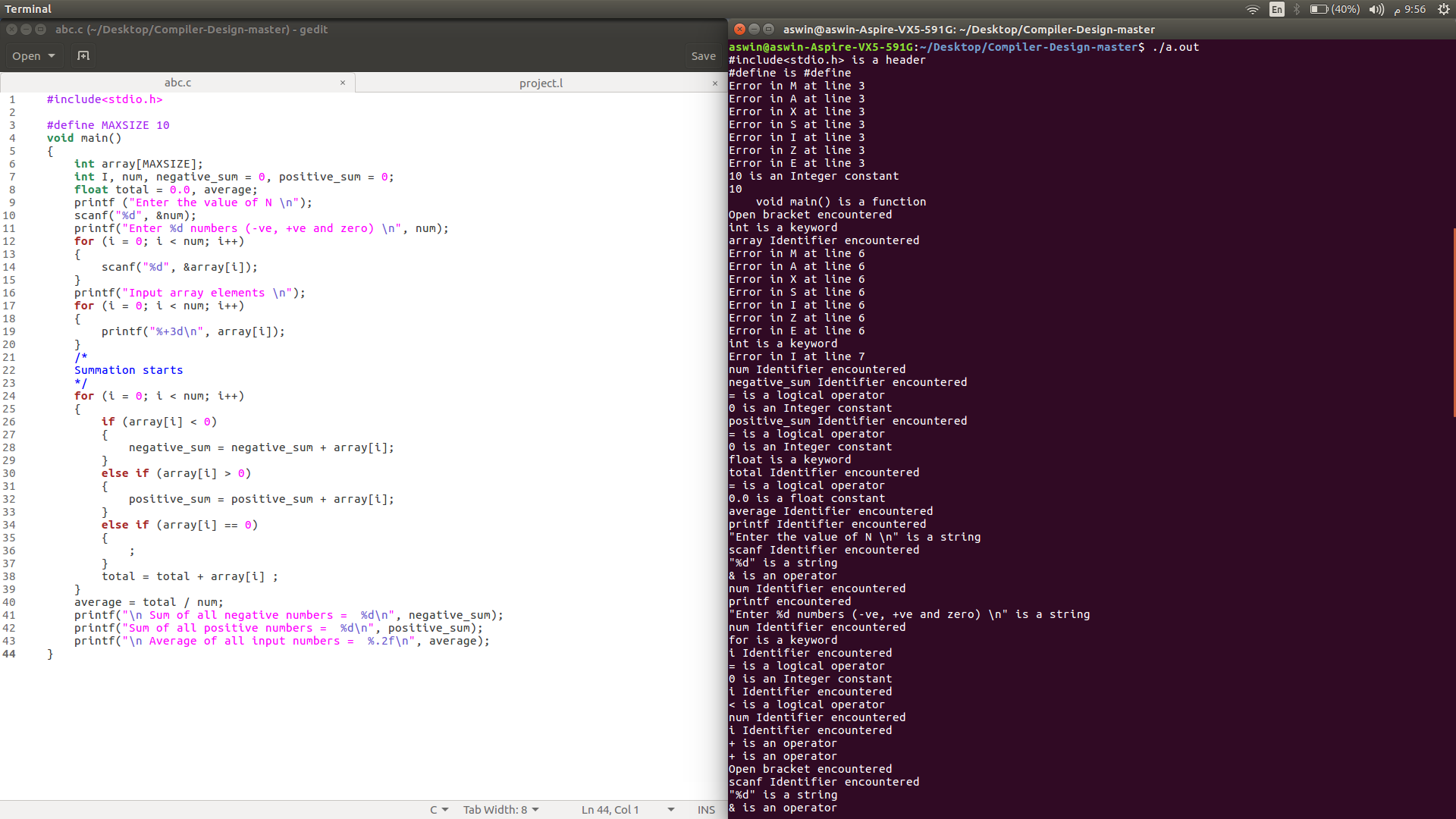
|  |
| --- |
|  |
|  |

.

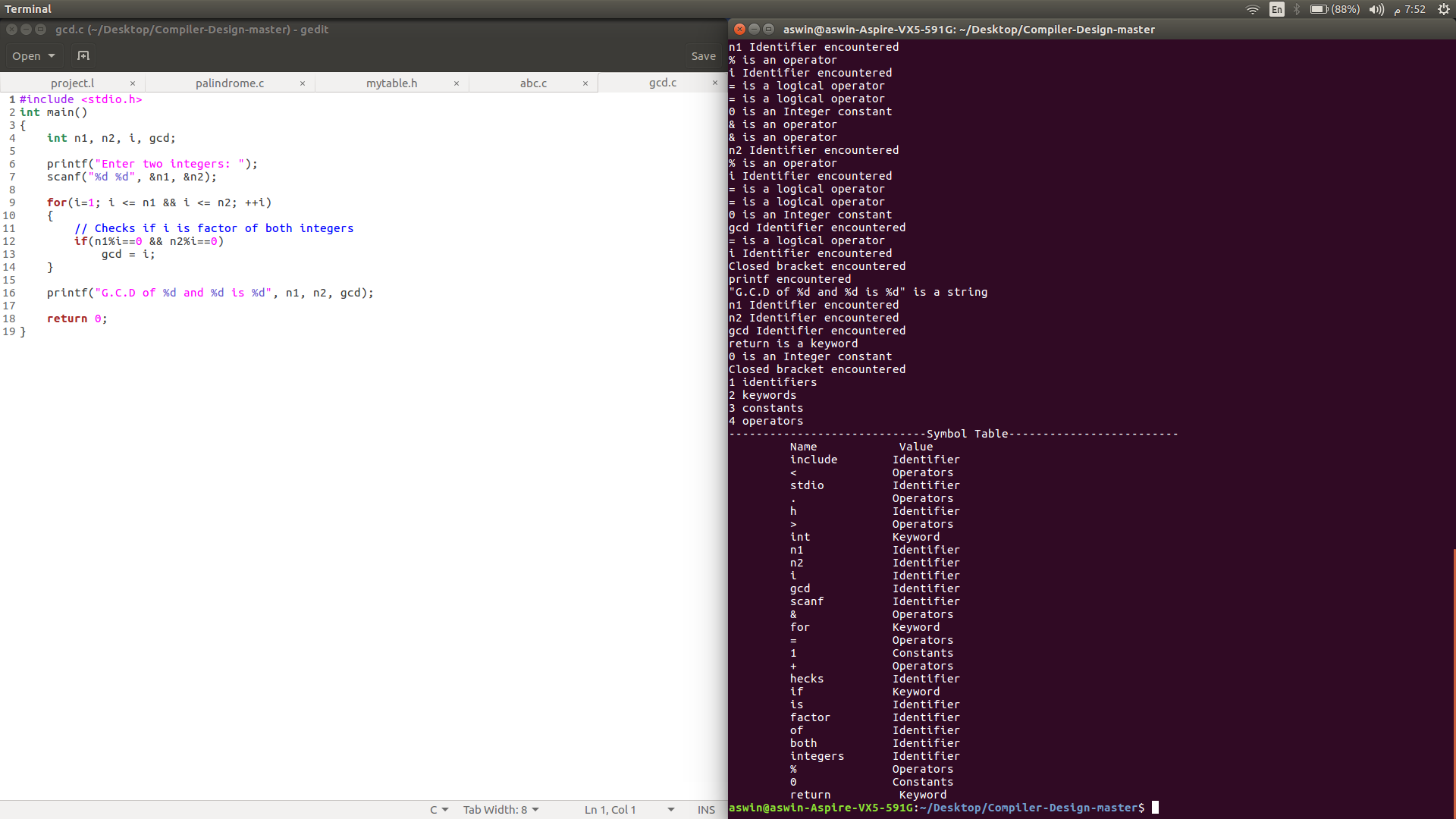
**RESULTS**

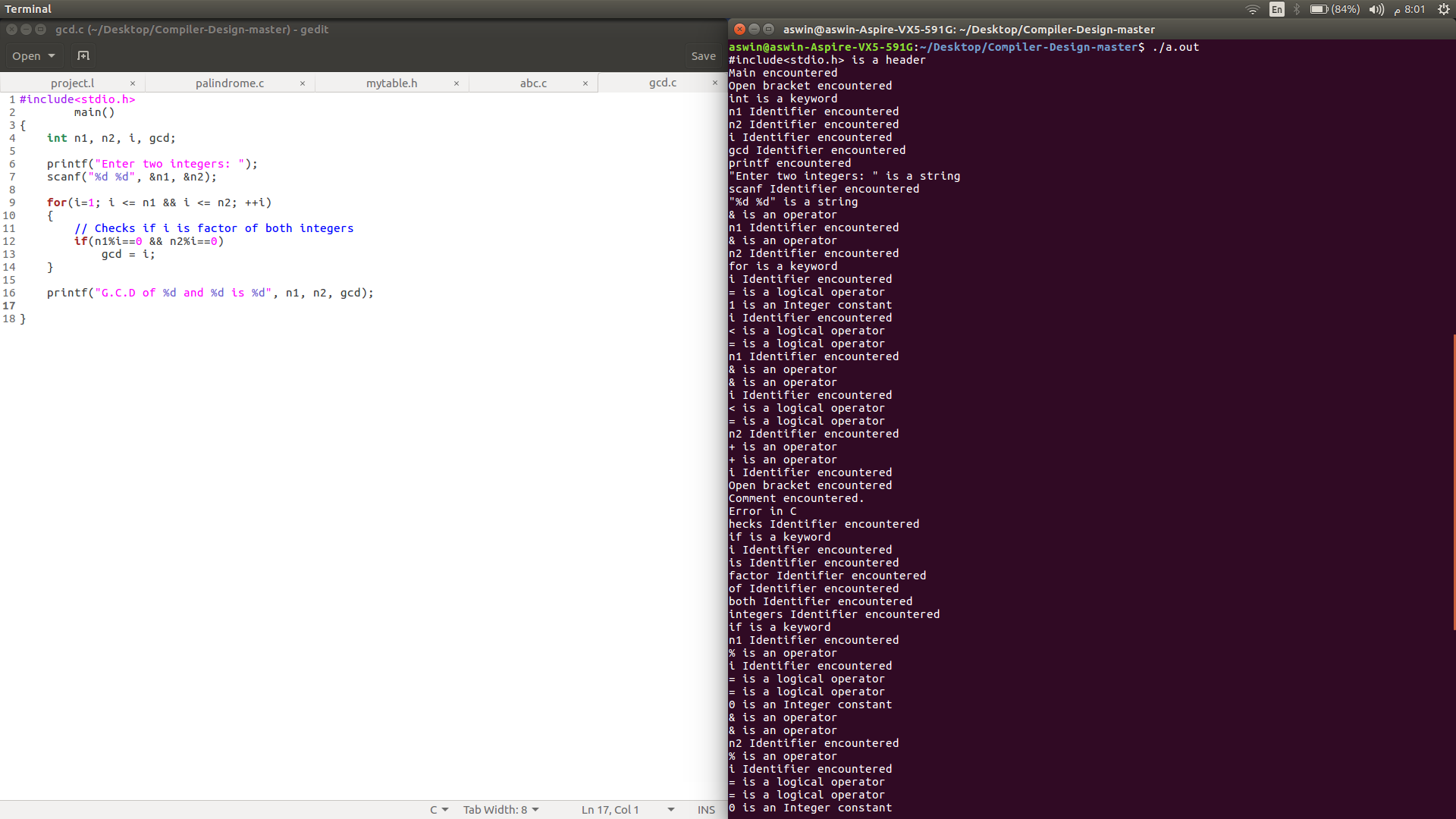
**Test-case #1**

****

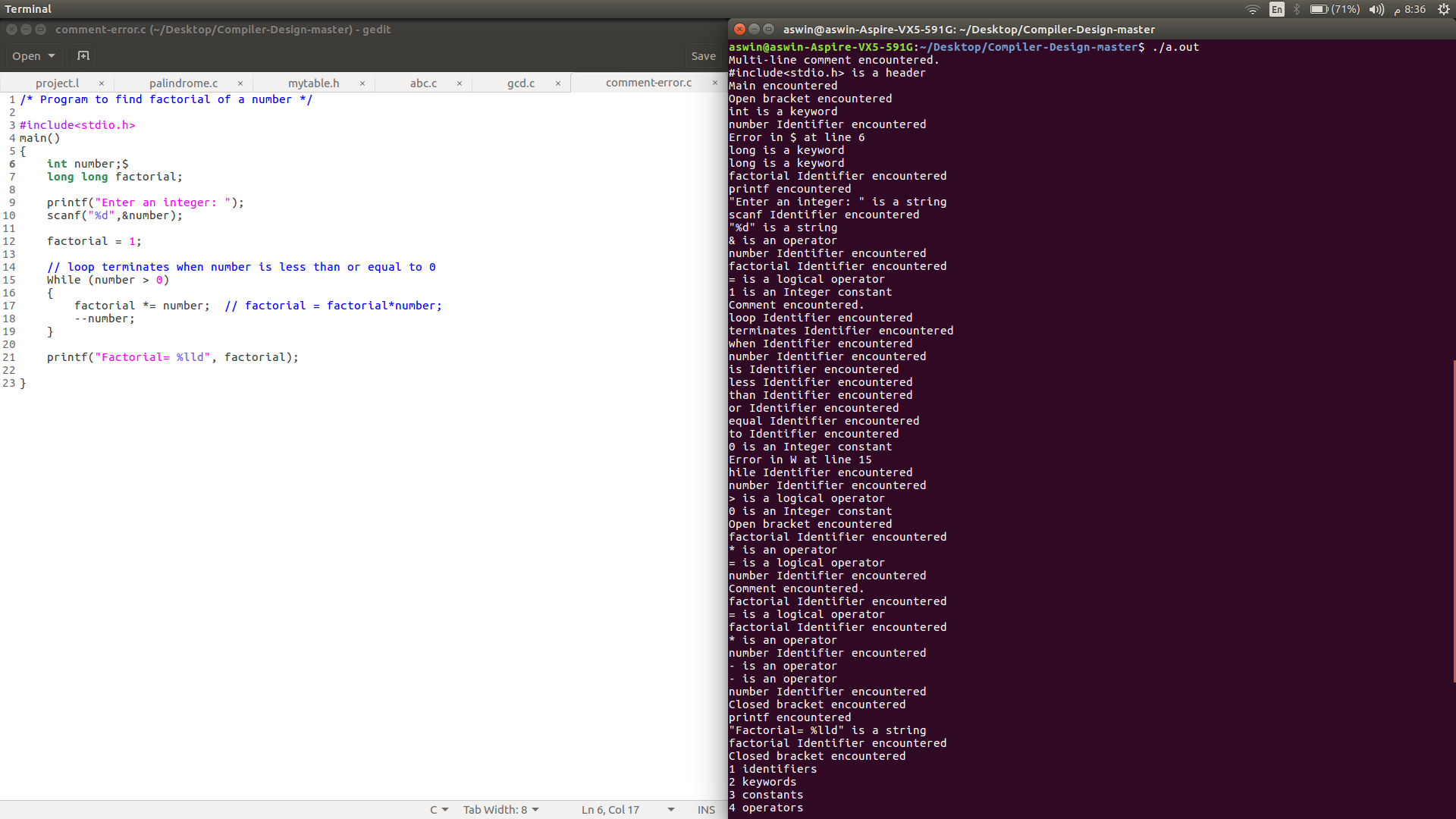
****

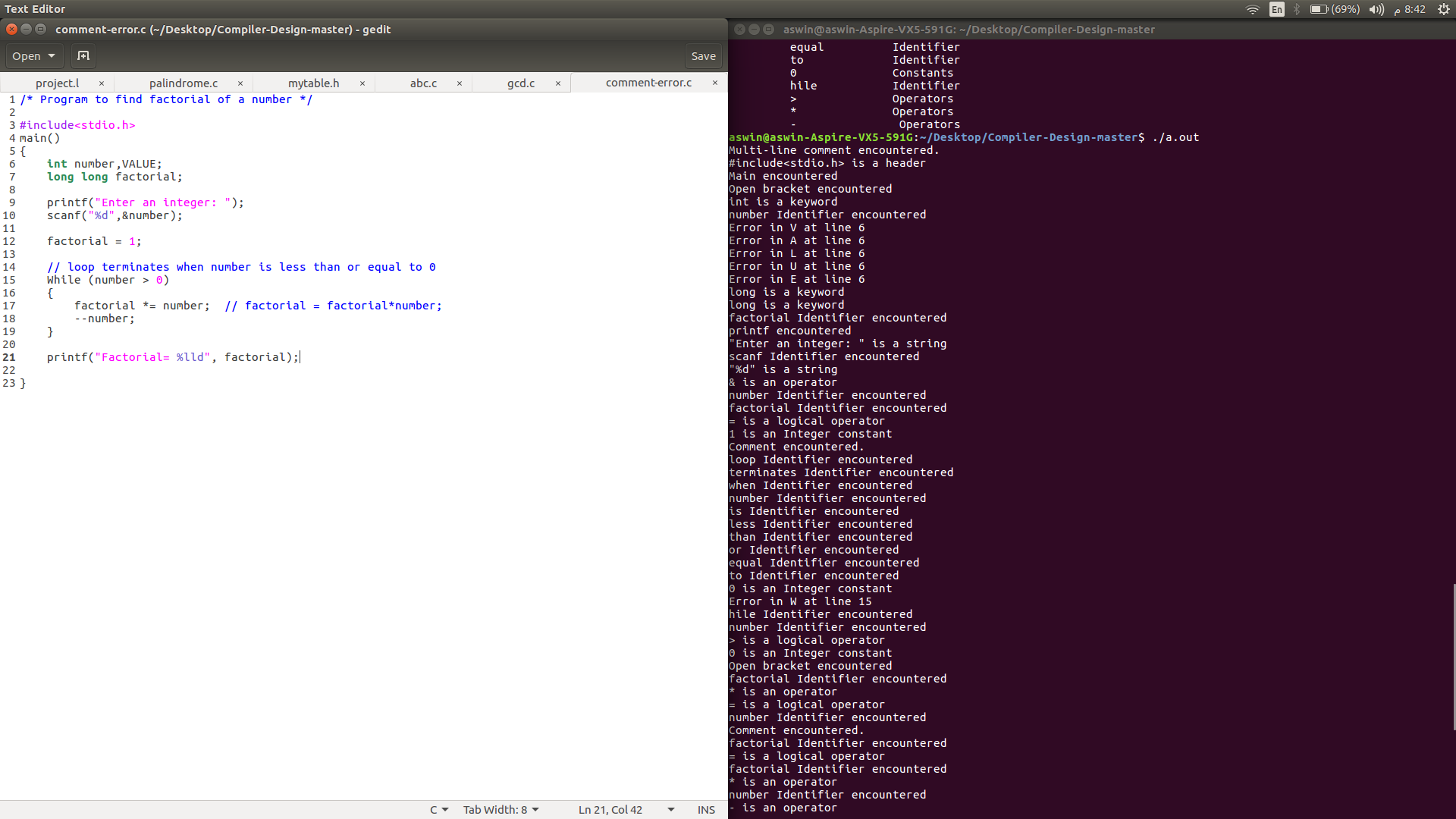
**Test-case #2**

****

****

**Test-case #3 (has errors)**

****

****