**MAULANA AZAD**

**NATIONAL INSTITUTE OF TECHNOLOGY**

**BHOPAL, INDIA, 462003**



**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**

**Compiler Design LAB Project Report**

**Semester V**

## “MIPS Code Generation”

Under the Guidance of:

**Prof. Mr .Gagan Vishwakarma**

**CSE Department**

**MANIT Bhopal**

Submitted by:

|  |  |  |
| --- | --- | --- |
| **Raj Pratap Singh Tomar** | **Vikas Sao** | **Shubham Yadav** |
| Sch. No. 181112049 | Sch. No. 181112051 | Sch. No. 181112029 |
| CSE Department | CSE Department | CSE Department |
| MANIT Bhopal | MANIT Bhopal | MANIT Bhopal |

|  |
| --- |
| **Jitendra Patel**  Sch. No. 181112073  CSE Department  MANIT Bhopal |
|  |

|  |
| --- |
| **Vijay Yadav**  Sch. No. 181112074  CSE Department  MANIT Bhopal |

**MAULANA AZAD**

**NATIONAL INSTITUTE OF TECHNOLOGY**

**BHOPAL, INDIA, 462003**

(Computer Science & Engineering Department)



DECLARATION

We hereby declare that the project work entitled **“*MIPS Code Generation*”** submitted at Maulana Azad National Institute of Technology-Bhopal, is a record of an original work done by us under the **guidance of Prof.**Gagan Vishwakarma, Assistant Professor, Department of Computer Science and Engineering (CSE), MANIT-Bhopal, and this project work has not been submitted for the award of any other Degree or diploma / associate ship / fellowship and similar project if any.

#### Raj Pratap singh tomar Vikas Sao Shubham Yadav

(181112049) (181112051) (181112029)

#### Vijay Yadav Jitendra Patel

(181112074) (181112073)

Prof. Gagan Vishwakarma

(Project Guide)

# ACKNOWLEDGEMENT

Our first experience of project has been successfully, thanks to the support staff of many friends & colleagues with gratitude. We wish to acknowledge all of them. However, we wish to makes special mention of the following.

First of all we are thankful of our project guide **Prof. Gagan Vishwakarma** under whose guideline we were able to complete our project. We are wholeheartedly thankful to him for giving us his value able time & attention & for providing us a systematic way for completing our project in time.

**ABSTRACT**

The topic of the project is Using the compiler generation tools LEX and YACC for

generating MIPS code for the following language feature Function Call, Declaration

and constant expression.

With the help this project we want to explain basic concept of lexical analyzer.

In this project we also explore Yacc and lex tools of compiler. We use platform

Flex, bison and dev to run this program.

**TABLE OF CONTENTS**

Abstract

0.

0.1 Lexical Analyzer

0.2 Symantic Analyzer

1. Introduction
   1. Function Call
   2. Declaration
   3. Constant expression

2. Literature Review and Survey

3. Gaps identified

4. Proposed work and methodology

4.1 Proposed work

4.2 Methodology

5. Conclusion

6. References

Stage 1

**STAGE 1 (LEXICAL ANALYZER)**

In this phase, a program is given to your code.

If that program follows the lexical rules, it must print the tokens of that program in the output.

Otherwise , it must print the appropriate error.

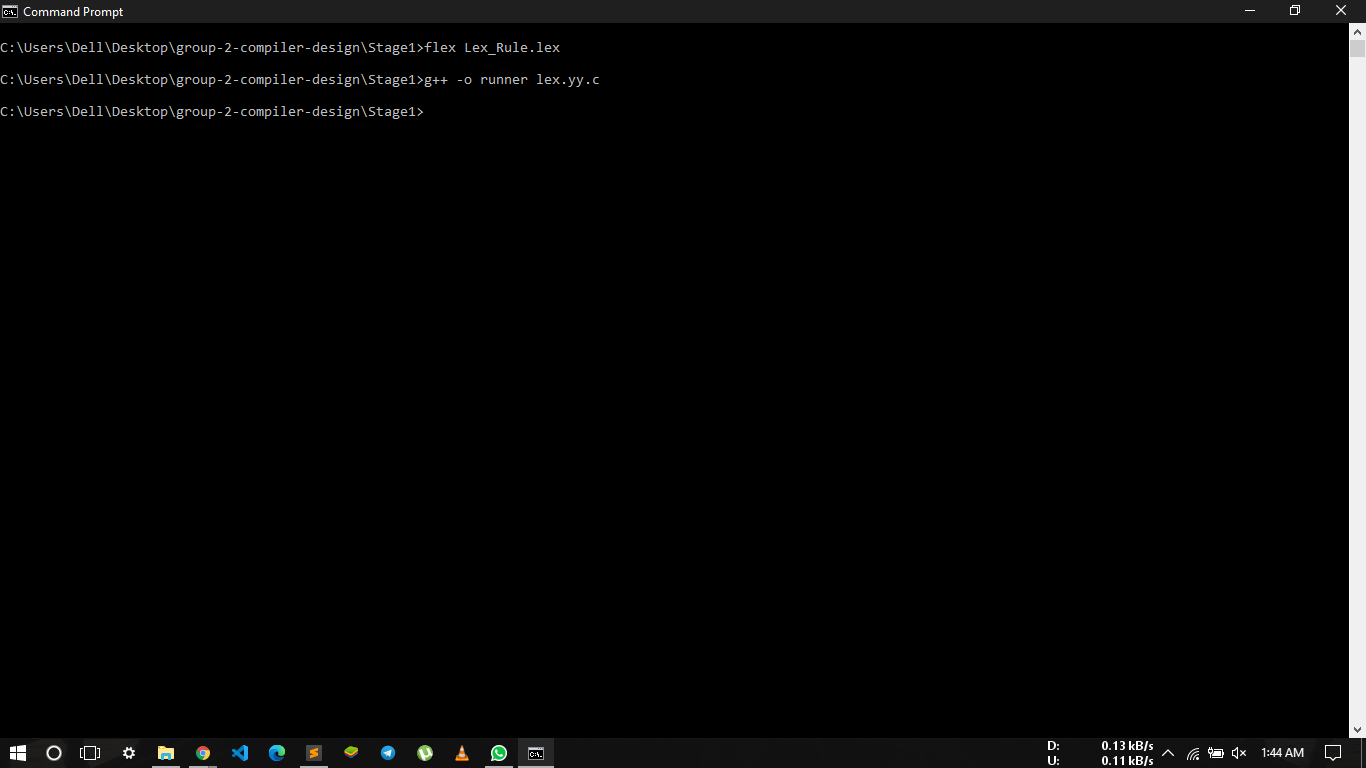
**To make the file**

>>>flex Lex\_Rule.lex // creates lex.yy.c

>>>g++ -o runner lex.yy.c // creates the nunner.exe file

**To test:**

>>>runner [testfilename]



Stage 2

**STAGE 2(SYMANTIC ANAYSIS)**

In this phase, a program is given to your code, if that program has followed the lexical and syntactic rules of the language, it will create an \_\_Abstract Syntax Tree\_\_, otherwise, it must print the appropriate error with line and column number.

**To make the file**

>>>bison Bison\_Rule.y // creates Bison\_Rule.tab.c

>>>flex Lex\_Rule.lex // creates lex.yy.c

>>>bison -d Bison\_Rule.y -o myapp.cpp // creates the myapp.cpp and myapp.hpp

file

>>>g++ -o runner Bison\_Rule.tab.c lex.yy.c // creates the nunner.exe file using

the Bison\_Rule.tab.c and lex.yy.c

**Instructions:**

>>>my [testCaseName] 0

• To display tokens and their values along with the display of terminals

that go to Epsilon.

>>>my [testCaseName] 1

• To display tokens along with the terminals that go to Epsilon.

>>>my [testCaseName] 2

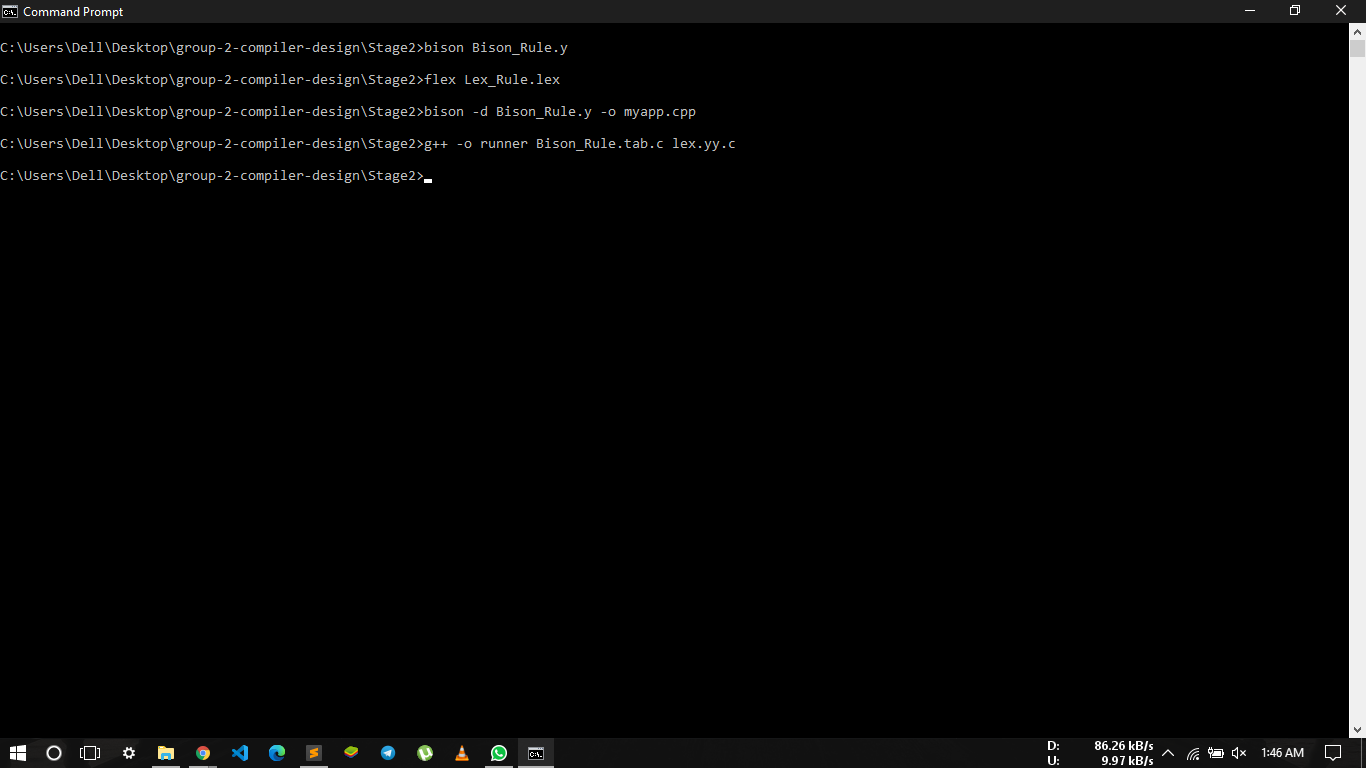
• To display tokens and their values.

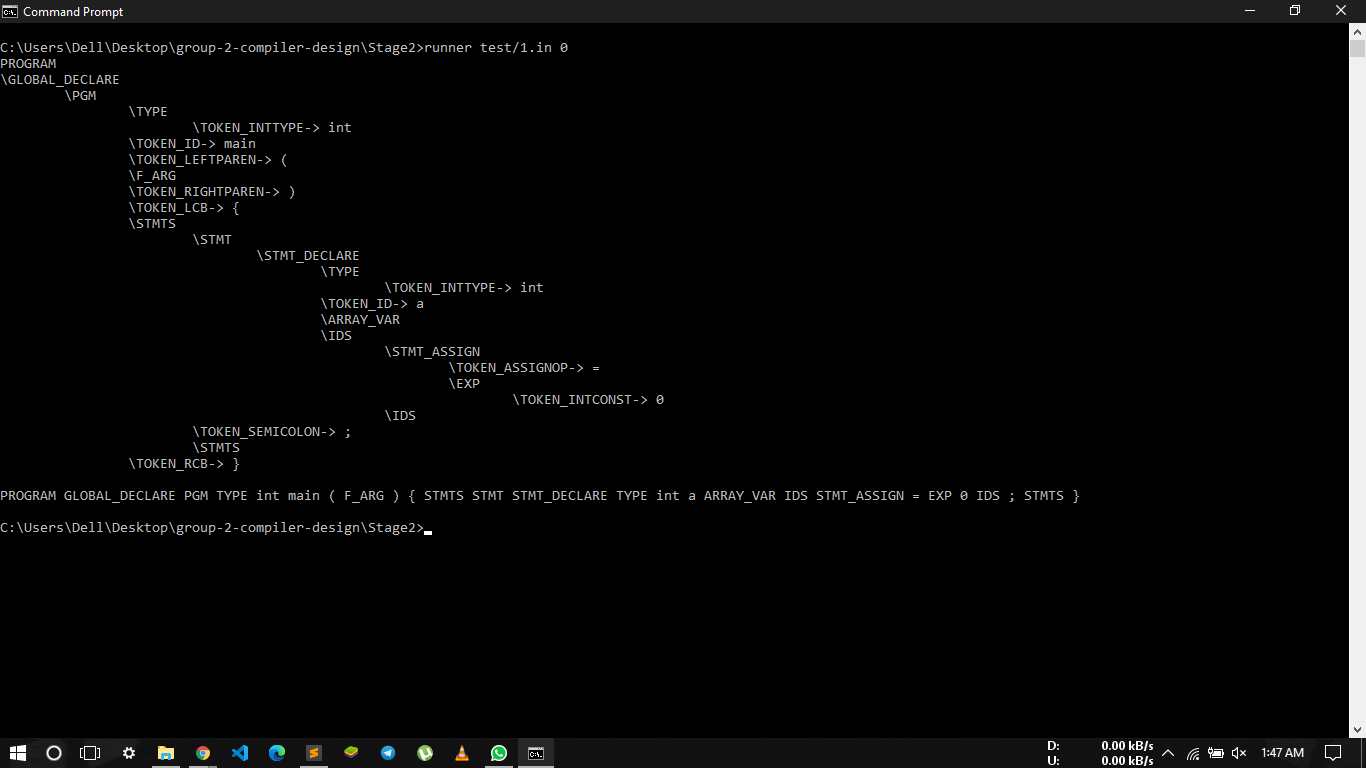
>>>my [testCaseName] 3

• To display tokens.

**To test:**

>>>runner [testfilename] [mode]





Stage 3

**STAGE 3(GENERATE MIPS CODE)**

In this phase, a program is given to your code, if that program has followed the lexical and syntactic rules of the language, it will create a \_\_MIPS code\_\_, otherwise, it must print the appropriate error with line and column number.

**To make the file :**

>>>bison Bison\_Rule.y // creates Bison\_Rule.tab.c

>>>flex Lex\_Rule.lex // creates lex.yy.c

>>>bison -d Bison\_Rule.y -o myapp.cpp // creates the myapp.cpp

and myapp.hpp file

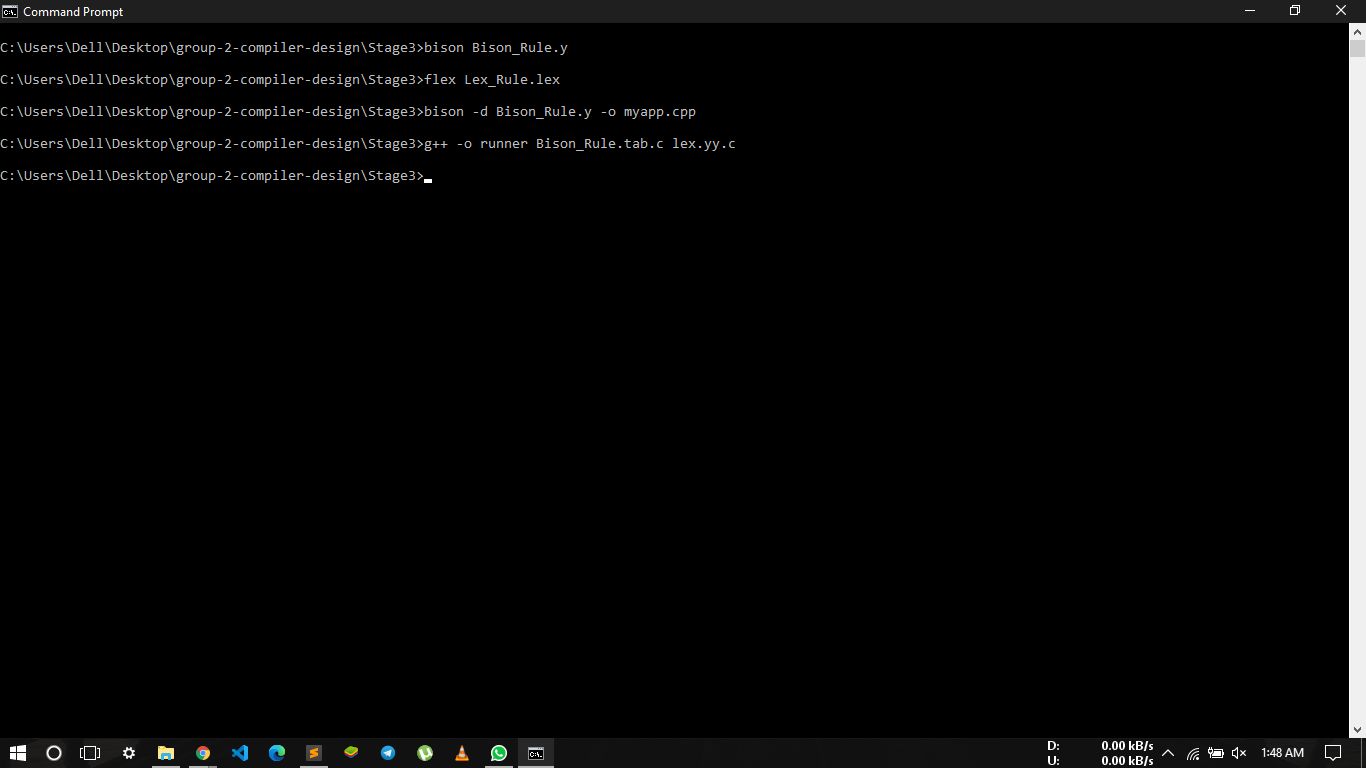
>>>g++ -o runner Bison\_Rule.tab.c lex.yy.c // creates the nunner.exe

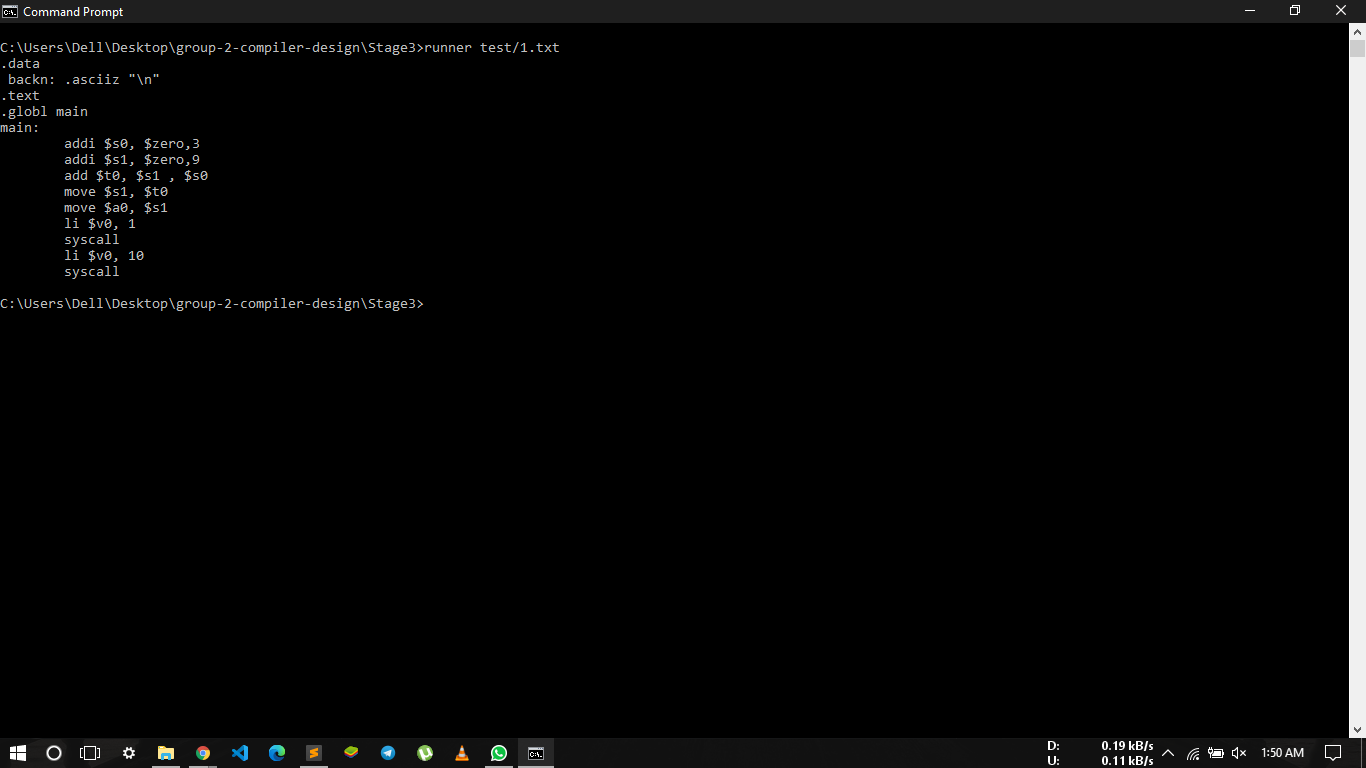
file using the

Bison\_Rule.tab.c and lex.yy.c

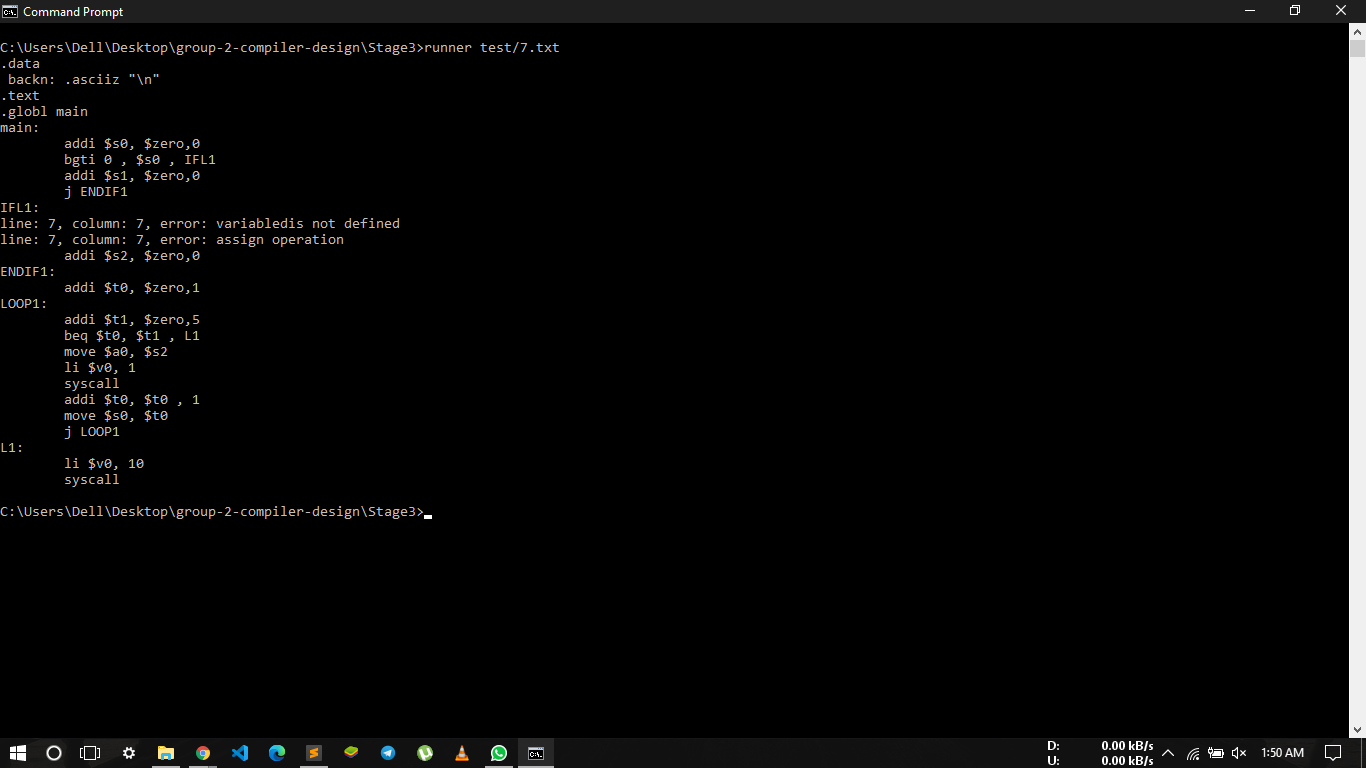
**To test:**

>>>runner [testfilename]





( mips code for test c++ file )

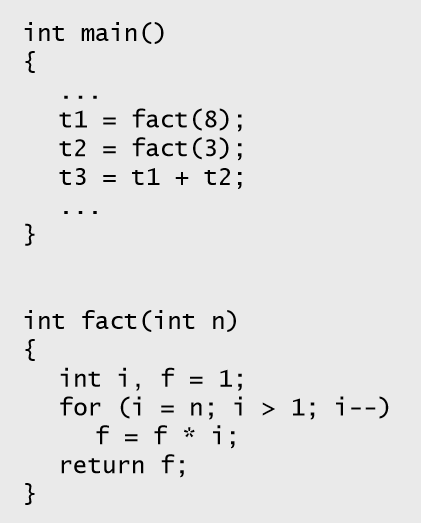


( mips code for test c++ file with errors )

**Function Call :**

A function call is a request made by a program or script that performs a predetermined function. In the example below, a batch file clears the screen and then calls another batch file.

**Function call in C :**  
A. Invoking a function changes the control flow of a program twice.   
 1. Calling the function   
 2. Returning from the function   
B. In this example the main function calls fact twice, and fact returns twice—but to   
 different locations in main.   
C. Each time fact is called, the CPU has to remember the appropriate return address.   
D. Notice that main itself is also a function! It is called by the operating system when you   
 run the program.



**Function Control Flow MIPS :**

A. MIPS uses the jump-and-link instruction jal to call functions.   
 1. The jal saves the return address (the address of the next   
 instruction) in the dedicated register $ra, before jumping to the   
 function.  
 2. jal is the only MIPS instruction that can access the value of the   
 program counter, so it can store the return address PC+4 in $ra.  
 jal Fact  
B. To transfer control back to the caller, the function just has to jump   
 to the address that was stored in $ra.  
 jr $ra  
C. Let’s now add the jal and jr instructions that are necessary for our   
 factorial example.

**Function Declaration :**

A *declaration* is a C language construct that introduces one or more [identifiers](https://en.cppreference.com/w/c/language/identifier) into the program and specifies their meaning and properties.  
Declarations may appear in any scope. Each declaration ends with a semicolon (just like [a statement](https://en.cppreference.com/w/c/language/statements)) and consists of two distinct parts .  
 Data Declarations :  
Placed in section of program identified with assembler directive . Data declares variable names used in program . Storage allocated in main memory (RAM) .

**Constant Expression :**

A constant expression can be evaluated at compile time.   
That means it has no variables in it.   
 For example: 5 + 7 / 3 is a constant expression.   
 Something like:  
 5 + some Number / 3 is not,  
assuming some Number is a variable ( ie , not itself a compile-time constant).

**MIPS Register set:**

MIPS assembly language employs a *convention* for use of registers. This convention is not enforced by the assembler or the hardware, but it *must* be followed by all MIPS assembly language programmers in order to avoid unexpected behaviour of modules written by different people.

|  |  |  |
| --- | --- | --- |
| **Register Number** | **Conventional Name** | **Usage** |
| $0 | $zero | Hard-wired to 0 |
| $1 | $at | Reserved for pseudo-instructions |
| $2 - $3 | $v0, $v1 | Return values from functions |
| $4 - $7 | $a0 - $a3 | Arguments to functions - not preserved by subprograms |
| $8 - $15 | $t0 - $t7 | Temporary data, not preserved by subprograms |
| $16 - $23 | $s0 - $s7 | Saved registers, preserved by subprograms |
| $24 - $25 | $t8 - $t9 | More temporary registers, not preserved by subprograms |
| $26 - $27 | $k0 - $k1 | Reserved for kernel. Do not use. |
| $28 | $gp | Global Area Pointer (base of global data segment) |
| $29 | $sp | Stack Pointer |
| $30 | $fp | Frame Pointer |
| $31 | $ra | Return Address |

As till now we have discussed about the what we are doing or what our project is about. We also discussed what the software we have used.

**Tools and technologies used**

##### Software requirement

* Ubuntu Terminal
* MIPS assembler
* Flex ( Lex )
* Bison ( Yacc )

**Important files :**

Lex\_Rule : This file is for lexical analysis which searches for valid key words that is used in our program. when we run "Lex\_Rule.lex" file in command prompt, this will generate lex.yy.c

Bison\_Rule.y : In this file we are defining our rules(grammar) for which, what action we have to take. when we run "yacc -d Bison\_Rule..y" file in command prompt, this will generate y.tab.c and y.tab.h for our compiler

Some Description about files are as follows

* [exe file](https://github.com/experiment-lab/compiler_mips_codeGeneration/blob/main/base.exe) This exe file for our Compiler.
* [Lex\_Rule](https://github.com/experiment-lab/compiler_mips_codeGeneration/blob/main/lang.l) This file is lexical analysis for expressions. (Written in lex)
* [Bison\_Rule.tab](https://github.com/experiment-lab/compiler_mips_codeGeneration/blob/main/lang.tab.c) This file is generated by [Bison\_Rule](https://github.com/experiment-lab/compiler_mips_codeGeneration/blob/main/lang.l) file(Parser) (Written in C)
* [Bison\_Rule.tab.h](https://github.com/experiment-lab/compiler_mips_codeGeneration/blob/main/lang.tab.h)This file is generated by [Bison\_Rule](https://github.com/experiment-lab/compiler_mips_codeGeneration/blob/main/lang.l) file (header file)
* [Bison\_Rule.y](https://github.com/experiment-lab/compiler_mips_codeGeneration/blob/main/lang.y) This file have Grammar Section (Written in Yacc)
* [lex.yy.c](https://github.com/experiment-lab/compiler_mips_codeGeneration/blob/main/lex.yy.c) This file is generated by [Bison\_Rule](https://github.com/experiment-lab/compiler_mips_codeGeneration/blob/main/lang.l) file (Written in C)
* [runner](https://github.com/experiment-lab/compiler_mips_codeGeneration/blob/main/out.s) This file is Default Output file for program (Assembly language)

Conclusion

In a compiler the process of Intermediate code generation is independent of machine and the

process of conversion of Intermediate code to target code is independent of language used.

Thus we have done the front end of compilation process. It includes 3 phases of compilation lexical

analysis, syntax analysis and semantic analysis which is then followed by intermediate code

generation.

In computer programming, the translation of source code into object code by a compiler. This report

outlines the analysis phase in compiler construction. In it’s implementation and source language is

converted to assembly level language

Conclusion

In a compiler the process of Intermediate code generation is independent of machine and the

process of conversion of Intermediate code to target code is independent of language used.

Thus we have done the front end of compilation process. It includes 3 phases of compilation lexical

analysis, syntax analysis and semantic analysis which is then followed by intermediate code

generation.

In computer programming, the translation of source code into object code by a compiler. This report

outlines the analysis phase in compiler construction. In it’s implementation and source language is

converted to assembly level language

Conclusion

In a compiler the process of Intermediate code generation is independent of machine and the

process of conversion of Intermediate code to target code is independent of language used.

Thus we have done the front end of compilation process. It includes 3 phases of compilation lexical

analysis, syntax analysis and semantic analysis which is then followed by intermediate code

generation.

In computer programming, the translation of source code into object code by a compiler. This report

outlines the analysis phase in compiler construction. In it’s implementation and source language is

converted to assembly level language

Conclusion

In a compiler the process of Intermediate code generation is independent of machine and the

process of conversion of Intermediate code to target code is independent of language used.

Thus we have done the front end of compilation process. It includes 3 phases of compilation lexical

analysis, syntax analysis and semantic analysis which is then followed by intermediate code

generation.

In computer programming, the translation of source code into object code by a compiler. This report

outlines the analysis phase in compiler construction. In it’s implementation and source language is

converted to assembly level language.

**Conclusion**

In a compiler the process of Intermediate code generation is independent of machine and the process of conversion of Intermediate code to target code is independent of language used.

Thus we have done the front end of compilation process. It includes 3 phases of compilation lexical analysis, syntax analysis and semantic analysis which is then followed by intermediate code generation.

In computer programming, the translation of source code into [object code](https://en.wikipedia.org/wiki/Object_code) by a [compiler](https://en.wikipedia.org/wiki/Compiler). This report outlines the analysis phase in compiler construction. In it’s implementation and source language is converted to assembly level language.