

CD LAB PROJECT REPORT

Development of a compiler for a new programming language

TEAM-1

Team member 1: Hemant Kumar (CS18B014)

Team member 2: Rohit Shakya (CS18B029)

Team member 3: Sourabh Misal (CS18B022)

Team member 4: K Pavan Kumar (CS18B015)

03.05.2021

6TH sem compiler lab

As part of our Compiler design Laboratory Course we have developed a fully functional Compiler which takes .txt file as input and gives assembly(.s) file as output. In this project We have developed a new programming language and then we have designed a compiler for it in different phases.

Firstly we have designed a new toy programming language which has basic functionalities. And then we have created a compiler for our new programming language in different phases. In the first phase we have captured different tokens in source language then we have parsed all those tokens and found out if they follow the grammar of our source language in syntax analyzer. Then in the semantic analyzer we have inserted identifiers inside the symbol table and we have also type checked all those identifiers wherever needed. Then in the Intermediate code generation phase we have taken syntax trees as input and converted them to equivalent intermediate code. Then in

the last phase we took ICG as input and converted it into MIPS assembly language.

Structure of the report -

Chapter 1: Language and tool choices	2
Chapter 2: Major components of the project	3
2.1 Scanner.....	3
2.2 Parser.....	3
2.3 Symbol table.....	4
2.4 Semantic analyser.....	5
2.5 Type checker.....	6
2.6 Intermediate code generator.....	6
2.7 code generator.....	6
Chapter 3: Flow diagram	7
Chapter 4: Source Code organisation.....	8
Chapter5: Final testing and evaluation with screen shots.....	9
Chapter 3: Limitations of the compiler	13
Chapter 3: Conclusion	14
Chapter 3: Contributions	15

Language and Tool Choices -

We have created a toy programming language which is easy to learn and use. Although it has very few functionalities but it is adequate for writing normal programs such as simple sorting algorithm , factorial , sum of n numbers. So our main goal for creating a source language was to have a simple programming language which is easy to understand through its keywords. We wanted our language to have such keywords which speak about its functionalities. So that is why certain keywords of our language like **call** , **input** , **output** , **integer** etc. have been used instead and we can see these keywords describe fully what these keywords do in the programming language. So for creating other basic functionalities and syntax we have used **C language** as reference.

Tools Used

1. **Lex** - Lex is a tool used to generate a lexical analyzer.this tool is easy to use and it has very simple syntax and fulfills all the requirements of our project.
2. **Yacc** - It is a tool used to generate a parser. Yacc is a common tool which most of the developers also use and it has a massive community because of that it is easy to find solutions for different queries and also it is very easy to use.
3. **Language used** - C is used for integration of different parts of the compiler. C is a very popular language for designing programming languages and as our other tools like lex and yacc also use it so it is very easy to use one programming language throughout the project.
4. **Target language** - As we knew the basics of MIPS assembly

language so it was very easy for us to convert our intermediate code to mips assembly language that is why we chose it over other languages.

5. **QTSpim** - We have used QTSpim before while compiling different MIPS assembly code so that is why we knew how easy it was to use and it is really simple to run mips code inside it.

Major components of the project -

Scanner -

As we have used lexer as a scanner, we need to provide regular expressions so it can make meaningful tokens from a stream of characters coming from source code. We need to differentiate between different lexemes in this phase - they can be keywords, identifiers, operators etc.

We can understand regular expressions from this example -

- Letter - a | b | z | A | B | | Z
- Digit - 1 | 2 | 3 | 9
- Identifiers - letter (letter | digit | _)
- Integer - digit (digit)*

Parser -

As we have used yacc as a parser, we need to provide some production rules in the form of context free grammar, so it can check the syntax of the statements in the source code. Parser will make the

syntax tree by the tokens produced by the lexer. Our language syntax should be followed by the programmers. If the programmer had made any mistake in the code then the programmer will be notified that he had syntax error in his code, if not then there will be an error at the time of execution.

We can understand production rules in parser from this example -

- Program : declarations VOID MAIN statements
- statements - statements assignexpr
 - | statements IF logicaexpr statements
 - | statements WHILE logicaexpr statements
 - | ;
- logicaexpr : logicaexpr '<' logicaexpr
 - | logicaexpr '>' logicaexpr
 - | ID
 - | NUM
- declarations : type id
 - | declaration id
- type : integer | float

Symbol Table -

So there are different scope's inside our code and corresponding to each scope there is a symbol table. Scope's are defined as a tree data

structure where we have linked every node of scope to its parent . as we know that if some identifier is not present in the particular Scope then it might be possible that it is present in its parent scope that was the reason for having this sort of data structure for scope and then we have implemented our symbol table in hash table as it is fast to look for identifiers and find out that it is present inside symbol table or not. We have used a struct data type which accommodates the name , data-type , value ,etc. of Identifier So that we can use them later throughout our project.We have a root node that corresponds to global scope, then child nodes are for different functions, and their child nodes for loop, and if-else blocks likewise.

Semantic analyzer -

Till this phase we have all the production rules for our syntax(if-else, for, while, expression, logical expressions, assignment and variable declaration) and we need to write semantic rules to check whether

1. Both the types in the assignment expressions are the same or not.
2. Variables used in the other expression are already declared in that scope or not.
3. Variables used in arithmetic expressions should have the ability to perform the task.
4. Function's return data type should match with the return data type in function declaration .
5. Parameters given in function call should be equal and have the same data type as defined in the declaration of function.

So these were some of tasks accomplished in the semantic analyzer phase.

Type checker -

So we have implemented a type checker in semantic analyzer and it first looks for the identifier inside the symbol table and then checks the data type of the identifier and performs appropriate tasks accordingly.

Intermediate Code Generator -

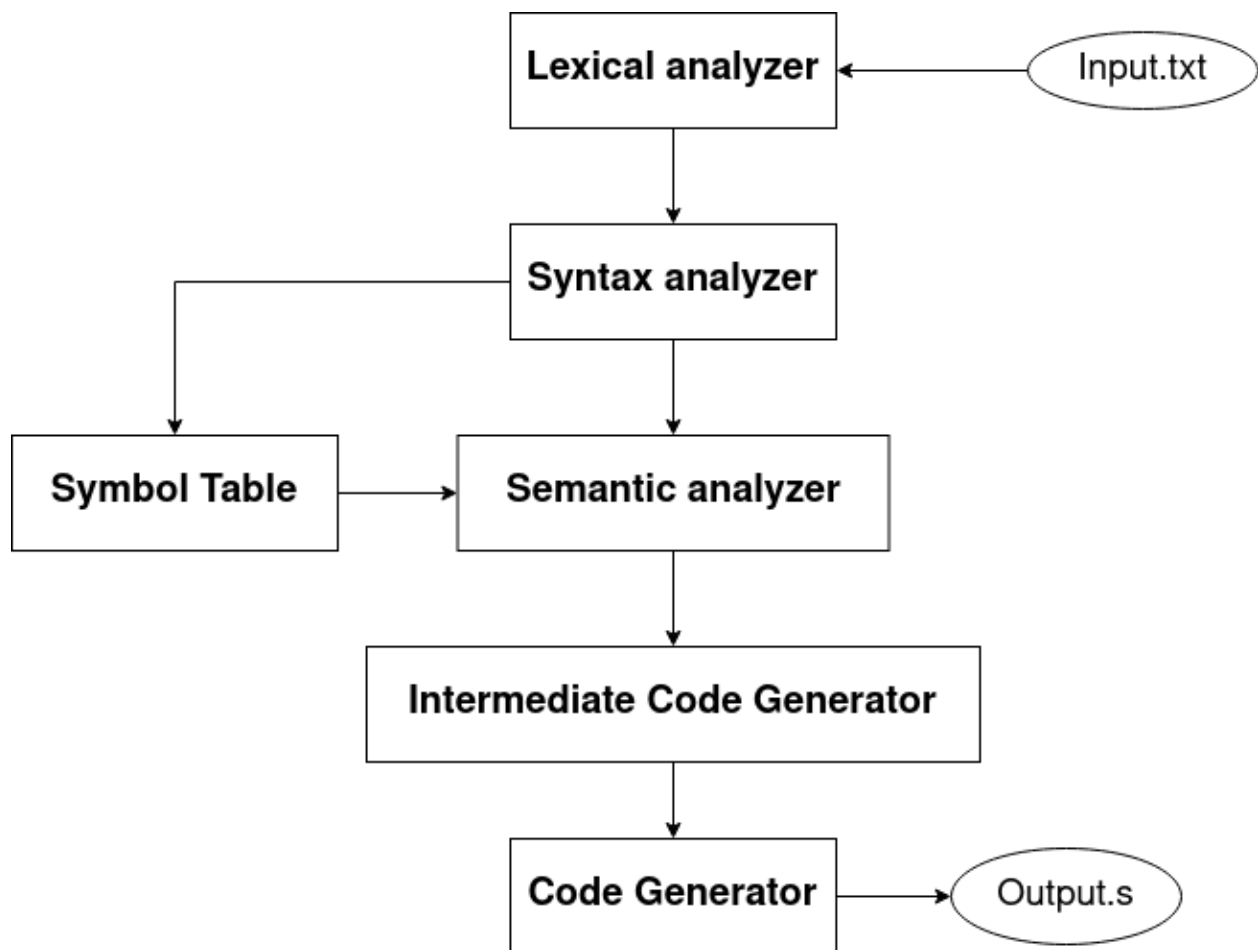
We have used **three address code** in the **quadruplets** form for the purpose of storing intermediate code. We have made a struct data type which stores four data variables (result, argument1, argument2, operation). This struct data type successfully stores one instruction in ICG equivalent to the corresponding syntax tree. Then we have created an array of that struct data type to store all instructions of ICG. we have converted ICG using a syntax tree which we have got in parser.

Code Generator -

Since we are having all the code and three address code with us at this stage. We need to just convert this three address code which we got from Intermediate code generation phase into mips assembly language. By using equivalent operations of different ICG instructions. like - add, sub, mul, div, beq, bne, la, li, lw, sw, j(jump) etc. are some of the basic mips operations which we have used as

equivalent to different ICG instructions.

Flow Diagram -



Source Code Organization -

1. Source file should be of **.txt** extension and should be written in source language taking reference from language manual.
2. There is no need to add any header files inside source code.
3. For building the project Anyone can download project from the given link <https://github.com/r-shakya/compiler>
4. Go inside MIPS directory Run make file in the terminal

Using **"make"** command .

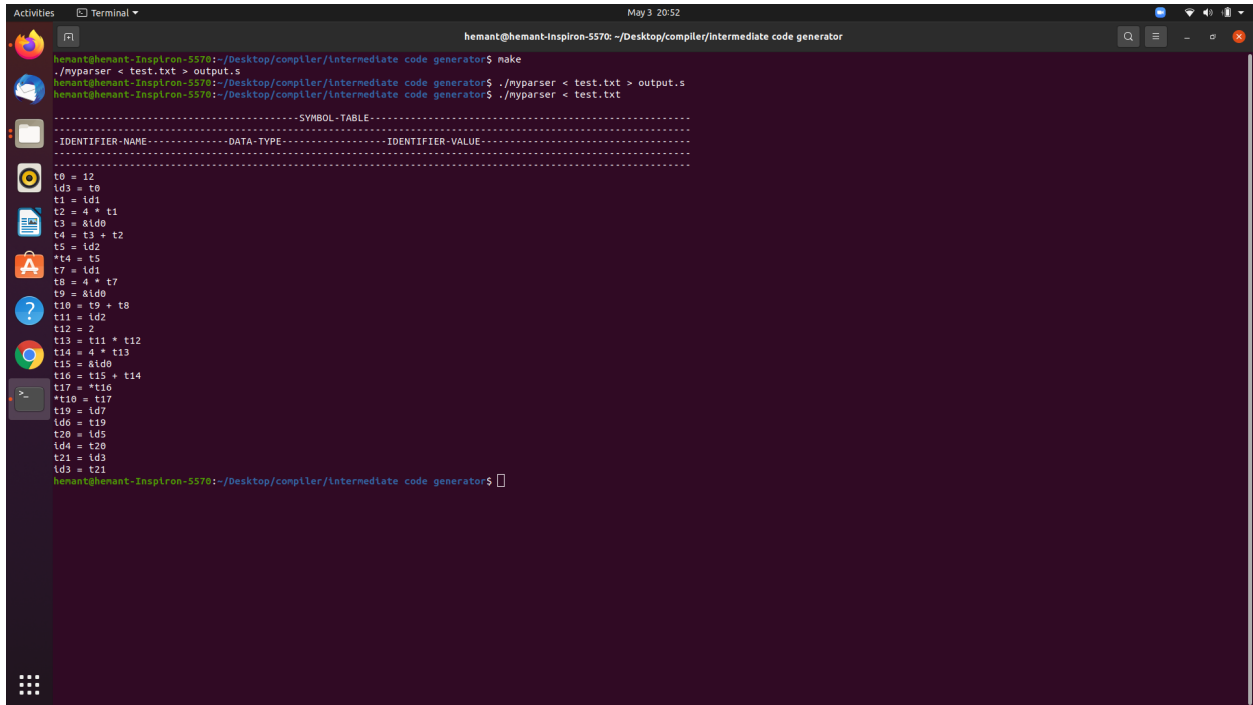
Now give the input file to myparser and store output inside output.s using following command

./myparser < input_file > output_file.s

5. Now for running Given **output.s** file we have to download QTSpim from <http://spimsimulator.sourceforge.net/>

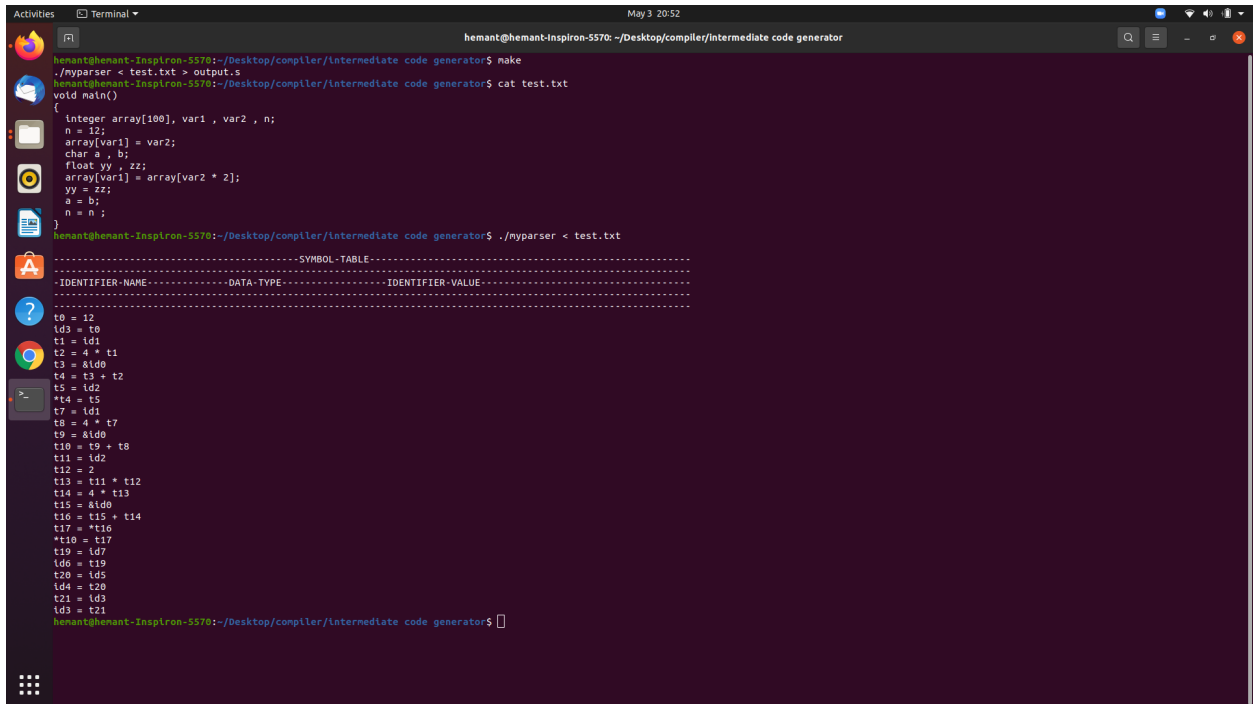
And now we can open **output.s** file inside QTSpim and Run.

Final testing and evaluation with screenshots



```
hemant@hemant-Inspiron-5570: ~/Desktop/compiler/intermediate code generator$ make
./myparser < test.txt > output.s
hemant@hemant-Inspiron-5570:~/Desktop/compiler/intermediate code generator$ ./myparser < test.txt > output.s
hemant@hemant-Inspiron-5570:~/Desktop/compiler/intermediate code generator$ ./myparser < test.txt

-----SYMBOL TABLE-----
IDENTIFIER-NAME DATA-TYPE IDENTIFIER-VALUE
-----
t0 = 12
ld3 = t0
t1 = ld1
t2 = 4 * t1
t3 = &ld0
t4 = t3 + t2
t5 = ld2
*t4 = t5
t7 = ld1
t8 = 4 * t7
t9 = &ld0
t10 = t9 + t8
t11 = ld2
t12 = 2
t13 = t11 * t12
t14 = 4 * t13
t15 = &ld0
t16 = t15 + t14
t17 = *t16
*t10 = t17
t19 = ld7
ld6 = t19
t20 = ld5
ld4 = t20
t21 = ld3
ld3 = t21
hemant@hemant-Inspiron-5570:~/Desktop/compiler/intermediate code generator$
```



```
hemant@hemant-Inspiron-5570:~/Desktop/compiler/intermediate code generator$ make
./myparser < test.txt > output.s
hemant@hemant-Inspiron-5570:~/Desktop/compiler/intermediate code generator$ cat test.txt
void main()
{
    integer array[100], var1, var2, n;
    n = 12;
    array[var1] = var2;
    char c, b;
    float yy, zz;
    array[var1] = array[var2 * 2];
    yy = zz;
    n = b;
    n = n;
}
hemant@hemant-Inspiron-5570:~/Desktop/compiler/intermediate code generator$ ./myparser < test.txt

-----SYMBOL TABLE-----
IDENTIFIER-NAME DATA-TYPE IDENTIFIER-VALUE
-----
t0 = 12
ld3 = t0
t1 = ld1
t2 = 4 * t1
t3 = &ld0
t4 = t3 + t2
t5 = ld2
*t4 = t5
t7 = ld1
t8 = 4 * t7
t9 = &ld0
t10 = t9 + t8
t11 = ld2
t12 = 2
t13 = t11 * t12
t14 = 4 * t13
t15 = &ld0
t16 = t15 + t14
t17 = *t16
*t10 = t17
t19 = ld7
ld6 = t19
t20 = ld5
ld4 = t20
t21 = ld3
ld3 = t21
hemant@hemant-Inspiron-5570:~/Desktop/compiler/intermediate code generator$
```

```
hemant@hemant-Inspiron-5570:~/Desktop/compiler/intermediate code generator$ cat test.txt
char c;
Integer a,b,arr1[3];

Integer sumofnumber(Integer a)
{
    c=a+b;
    a[1]=c;
    send C+1;
}

void main(){
    Integer a[3]={1,2,3};
    Integer var1,var2;
    char var3;
    var3 = call("sumofnumber",var1);
}

hemant@hemant-Inspiron-5570:~/Desktop/compiler/intermediate code generator$ ./myparser < test.txt
-----SYMBOL TABLE-----
-IDENTIFIER-NAME-----DATA-TYPE-----IDENTIFIER-VALUE-----
a                    Integer                    3
arr1                 Integer
b                    Integer
c                    char
-----
ldnum before var type a = Integer
ldnum after $
a variable type does not match with left assignment
left assign data type -char, right assignment data type -Integer
hemant@hemant-Inspiron-5570:~/Desktop/compiler/intermediate code generator$
```

```
hemant@hemant-Inspiron-5570:~/Desktop/compiler/intermediate code generator$ cat test.txt
void main() {
    Integer n,t,total,temp;
    total=0;
    t=1;

    while(t<=5){
        total=total+t;
    }

    temp=total;
}

hemant@hemant-Inspiron-5570:~/Desktop/compiler/intermediate code generator$ ./myparser < test.txt
-----SYMBOL TABLE-----
-IDENTIFIER-NAME-----DATA-TYPE-----IDENTIFIER-VALUE-----
t0 = 0
t2 = t0
t1 = 1
t4 = t1
t0:
t2 = t0
t3 = 5
t4 = t2 < t3
if not t4 goto l1
-----
-IDENTIFIER-NAME-----DATA-TYPE-----IDENTIFIER-VALUE-----
l                    Integer
n                    Integer
total               Integer
temp               Integer
-----
t5 = t2
t6 = t0
t7 = t5 + t6
t2 = t7
goto t0
l1:
t0 = t2
t3 = t0
hemant@hemant-Inspiron-5570:~/Desktop/compiler/intermediate code generator$
```

The screenshot shows a terminal window titled "Terminal" with the user "hemant" at the host "hemant@hemant-inspiron-5570". The terminal displays the following commands and output:

```
hemant@hemant-inspiron-5570:~/Desktop/compiler/MIPS$ make
./myparser < test.txt > output.s
hemant@hemant-inspiron-5570:~/Desktop/compiler/MIPS$ cat test.txt
void main(){
    Integer temp,sum,i,j,n;
    input(n);
    for (i = 1; i < n + 1; i = i + 1){
        sum = sum + i;
    }
    output(sum);
}
hemant@hemant-inspiron-5570:~/Desktop/compiler/MIPS$ ./myparser < test.txt

.data
ld0:      .word 0;
ld1:      .word 0;
ld2:      .word 0;
ld3:      .word 0;
ld4:      .word 0;

.text
.globl main
main:
    li $v0, 5
    syscall
    sw $v0, ld4
    li $t0, 1
    sw $t0, ld2
ld0:
    lw $t1, ld2
    lw $t2, ld4
    li $t3, 1
    add $t4, $t2, $t3
    slt $t5, $t1, $t4
    beq $t5, $zero, l3
    j l2
l1:
    lw $t6, ld2
    li $t7, 1
    add $t8, $t6, $t7
    sw $t8, ld2
    j l0
l2:
    lw $t9, ld1
    lw $t0, ld2
    add $t1, $t9, $t0
    sw $t1, ld1
    j l1
```

The image shows a screenshot of the QtSpim MIPS simulator. The main window displays assembly code for a program that prints "Hello, world!". The code is as follows:
User Text Segment [00400000]..[00400000]
0: lw \$a0 0(\$sp) # argc
4: addiu \$a1 \$sp 4 # argv
6: addiu \$a2 \$a1 4 # envp
8: call \$v0 \$a0 2
7: addu \$a2 \$a2 \$v0
8: jal main
9: nop
10: li \$v0 10
11: syscall
12: syscall
13: sw \$v0, \$t4
14: li \$t0, 1
15: sw \$t0, \$t2
16: lw \$t1, \$t2
17: lw \$t2, \$t4
18: li \$t3, 1
19: add \$t4, \$t2, \$t3
20: sll \$t5, \$t1, \$t4
21: j 30; beq \$t5, \$zero, 13
22: j 12
23: lw \$t6, \$t2
24: li \$t7, 1
25: add \$t8, \$t6, \$t7
26: sw \$t8, \$t2
27: j 36; sw \$t8, \$t2
28: j 37; j 10
A console window is open on the left, showing the MIPS register file (R0-R31) and the current instruction being executed. The registers are as follows:
R0 [x]
R1 [x]
R2 [x]
R3 [x]
R4 [x]
R5 [x]
R6 [x]
R7 [x]
R8 [x]
R9 [x]
R10 [x]
R11 [x]
R12 [x]
R13 [x]
R14 [x]
R15 [x]
R16 [x]
R17 [x]
R18 [x]
R19 [x]
R20 [x]
R21 [x] = 0 [00400074] 3c011001 lui \$t1, 4097
R22 [x] = 0 [00400078] ac380008 sw \$t4, 8(\$t1)
R23 [x] = 0 [0040007c] 08100010 j 0x00400040 [10]
The console also shows the memory and registers cleared, and the QtSpim version 9.1.21 of January 17, 2020.

```
hemant@hemant-Inspiron-5570: ~/Desktop/compiler/MIPS$ cat Factorial.txt

Integer factorial(Integer n){
    Integer one;
    one = 1;
    if(n <= 0){
        send one;
    }
    Integer factn;
    Integer new_n;
    new_n = n - 1;
    factn = call(factorial,new_n);
    factn = n*factn;
    send factn;
}

void main()
{
    Integer n;
    output("Enter Val :");
    input(n);
    Integer factn;
    factn = call(factorial,n);
    output("Val of Fact(");
    output(n);
    output(") : ");
    output(factn);
    output("\n");
}

hemant@hemant-Inspiron-5570: ~/Desktop/compiler/MIPS$ ./myparser < Factorial.txt

.data
ld0:
.word 0;
ld1:
.word 0;
ld2:
.word 0;
ld3:
.word 0;
ld4:
.word 0;
ld5:
.asciiz "Enter Val :";
ld6:
.word 0;
ld7:
.asciiz "Val of Fact(";
ld8:
.asciiz ") : ";
ld9:
.asciiz "\n"
.text
```

hemant@hemant-Inspiron-5570: ~/Desktop/compiler/MIPS\$./myparser < Factorial.txt >output.s

hemant@hemant-Inspiron-5570: ~/Desktop/compiler/MIPS\$

Enter Val :6

Val of Fact(6) : 720

FP Regs

nt Regs [16]

Data

Text

PC = 400020

EPC = 0

Cause = 0

BadAddr = 0

Status = 3000fff0

HI = 0

LO = 240

R0 [r0] = 0

R1 [a0] = 10010000

R2 [v0] = a

R3 [v1] = 0

R4 [a0] = 10010036

R5 [a1] = 7ffff9bc

R6 [a2] = 0

R7 [a3] = 0

R8 [t0] = 4

R9 [t1] = 240

R10 [t2] = 0

R11 [t3] = 0

R12 [t4] = 0

R13 [t5] = 0

R14 [t6] = 0

R15 [t7] = 0

R16 [a0] = 240

R17 [a1] = 0

R18 [a2] = 5

R19 [a3] = 0

R20 [a4] = 0

R21 [a5] = 0

R22 [a6] = 0

R23 [a7] = 0

User Text Segment [00400000]..[00440000]

[00400000] 8fa40000 lw \$4, 0(\$29) ; 183: lw \$a0 0(\$sp) # argc

[00400004] 27a50004 addiu \$5, \$29, 4 ; 184: addiu \$a1 \$sp 4 # argv

[00400008] 24a40004 addiu \$6, \$5, 4 ; 185: addiu \$a2 \$a1 4 # envp

[0040000c] 00a41080 sll \$2, \$4, 2 ; 186: sll \$v0 \$a0 2

[00400010] 00c23021 addu \$6, \$6, \$2 ; 187: addu \$a2 \$a2 \$v0

[00400014] 0c100009 jal 0x00400024 (main) ; 188: jal main

[00400018] 00000000 nop ; 189: nop

[0040001c] 3402000a ori \$2, \$0, 10 ; 191: li \$v0 10

[00400020] 0000000c syscall ; 192: syscall # syscall 10 (exit)

[00400024] 34020004 ori \$2, \$0, 4 ; 193: li \$v0, 4

[00400028] 3c011001 lui \$1, 4097 [ld5] ; 194: la \$a0, ld5

[0040002c] 34240014 ori \$4, \$1, 20 [ld5] ; 195: ori \$a1, \$a1, 20

[00400030] 0000000c syscall ; 196: syscall

[00400034] 34020005 ori \$2, \$0, 5 ; 197: li \$v0, 5

[00400038] 0000000c syscall ; 198: syscall

[0040003c] 3c011001 lui \$1, 4097 ; 199: la \$a0, ld6

[00400040] ac220010 sw \$2, 16(\$1) ; 200: sw \$a0, ld6

[00400044] 3c011001 lui \$1, 4097 ; 201: la \$a0, ld7

[00400048] 8c330010 lw \$19, 16(\$1) ; 202: lw \$a3, ld7

[0040004c] 3c011001 lui \$1, 4097 ; 203: la \$a0, ld8

[00400050] ac330000 sw \$19, 0(\$1) ; 204: sw \$a3, 0(\$a0)

[00400054] 23bfff64 addi \$29, \$29, -64 ; 205: addi \$sp, \$sp, -64

[00400058] afbf0000 afbf0000 sw \$31, 0(\$29) ; 206: sw \$ra, 0(\$sp)

[0040005c] afbf0004 sw \$8, 4(\$29) ; 207: sw \$t0, 4(\$sp)

[00400060] afbf0008 sw \$9, 8(\$29) ; 208: sw \$t1, 8(\$sp)

[00400064] afbf000c sw \$10, 12(\$29) ; 209: sw \$t2, 12(\$sp)

[00400068] afbf0010 sw \$11, 16(\$29) ; 210: sw \$t3, 16(\$sp)

[0040006c] afbf0014 sw \$12, 20(\$29) ; 211: sw \$t4, 20(\$sp)

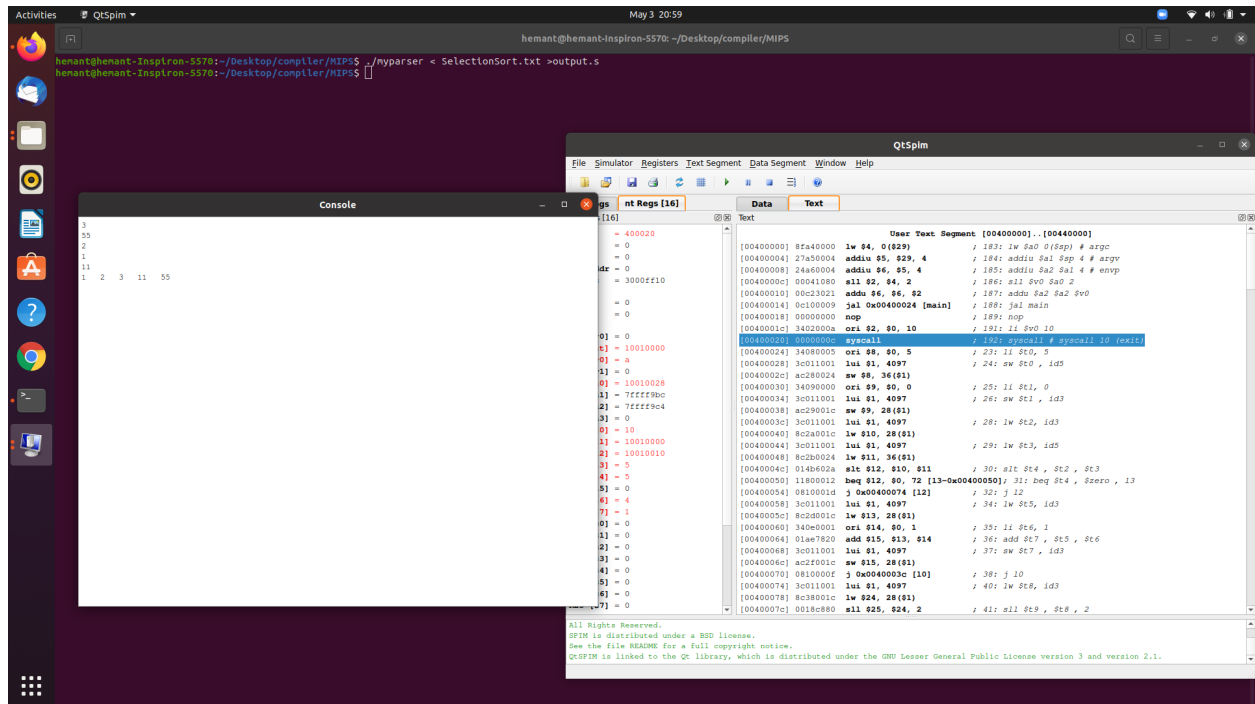
[00400070] afbf0018 sw \$13, 24(\$29) ; 212: sw \$t5, 24(\$sp)

[00400074] afbf001c sw \$14, 28(\$29) ; 213: sw \$t6, 28(\$sp)

[00400078] afbf0020 sw \$15, 32(\$29) ; 214: sw \$t7, 32(\$sp)

[0040007c] afbf0024 sw \$24, 36(\$29) ; 215: sw \$t8, 36(\$sp)

Memory and registers cleared



Limitations-

These are most of the limitations of our source language and compiler

1. Only integers are provided as data-type though they will not throw any syntax error .
2. 1-d array can only be used.
3. Dynamic memory allocation is not supported.
4. Pointers are not allowed
5. Type conversion is not possible.
6. User defined variables are not allowed.
7. Recursion can only be used with integers as parameters inside the function.
8. Function can only have four parameters at max.

9. In any expression there should be the same data type variable should be present.

Conclusion-

As we were designing our language everyone in the team participated in brainstorming and they came up with the different ideas of how they want their programming language to have syntax , semantics. Everyone was asking themselves different question regarding what it is that they feel should be appended in our programming language. We came up with different ideas and then integrated all those ideas into one programming language.

Before this project we had vague ideas about different phases which are involved in designing a compiler while building one we understood different trade offs of it. We also used different tools to make our project easier to implement. Now we have some ideas which involve building a compiler .

By doing this project in a team we got better understanding of version control as we have used git.

Contributors-

Hemant kumar(cs18b014) - making of symbol table, handled different phases of compiler semantic analyzer, intermediate code generation and code generation phases

Rohit Shakya(cs18b029) - making of syntax analyzer, partial work in the intermediate code generation and code generation phase, error handling by checking the code for different test cases.

Sourabh Misal(cs18b022) - making of lexical analyzer, some work in the other phases intermediate code generation and code generation, making of test cases so code can be checked for every aspect of different phases.

K Pavan Kumar(cs18b015) - writing valid and invalid programs in different phases, so we can check different aspects of our code, and fix the error. Partial work in lexical analyzer and intermediate code generation phase.