

COMPILER DESIGN LAB

Project report

TEAM5

Contributors:	Sripranav Mannepalli	CS18B036
	S.R.V.S. Maheswara Reddy	CS18B032
	Chirag Gupta	CS18B006
	AVS Hrudai	CS18B001

TABLE OF CONTENTS:

PGNO:

1.	Introduction	2
2.	Language And Tool Choices For Implementation	3
3.	Major components of the Project	
3.1.	Lexical analysis (scanner)	3
3.2.	Syntax Analysis	5
3.3.	Symbol table	6
3.4.	Semantic Analysis	6
3.5.	Type Checking	8
3.6.	Code Generation	9
3.7.	Register Allocation	10
4.	Flow/interactions between different components	12
5.	Source Code Organization	12
6.	Final testing and evaluation with screenshots	14
7.	Test Plan	17
8.	Limitations of our compiler	20
9.	Conclusion	20
10.	Contributions	21

INTRODUCTION :

Compiler : A compiler is a computer program that translates computer code written in one programming language (the source language) into another language (the target language). The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language, object code, or machine code) to create an executable program.

The **first phase** of our compiler is **lexical analysis** or scanning. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token that it passes on to the subsequent phase, syntax analysis.

The **second phase** of our compiler is **syntax analysis** or parsing. The parser uses the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. It analyses the syntactical structure and checks if the given input is in the correct syntax of the programming language or not.

The **third phase** in our compiler is **semantic analysis**. The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands.

The **last phase** in our compiler is **code generation**. We generate x86-64 bit nasm assembly code here in this phase. We can run this assembly code in Linux and Windows 64 bit machines.

LANGUAGE AND TOOL CHOICES FOR IMPLEMENTATION:

- **Lex:** We used Lex tool for lexical analysis (scanning). This transforms the input stream into a sequence of tokens. These tokens are further used by the yacc tool for further compilation steps.
- **Yacc:** We used a yacc tool(parser) for syntax, semantic analysis and as well as generating the further assembly code in our implementation.
- **Language:** We used C++ language for implementing the compiler, all the actions in parser are written in C++ language.
- **Target language :** X86 NASM 64 bit assembly language

Language specifications are in a separate file.

MAJOR COMPONENTS OF THE PROJECT:

1) Lexical Analysis:

- **Lexical Analysis** is the first phase of our compiler. It converts the input program into a sequence of Tokens.
- **Lex** is a lexical analysis tool that can be used to identify specific text strings in a structured way from source text.
- **Lex** file is divided into three sections, separated by lines that contain only two percent signs, as follows:
 - 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.

Design of our lexical analyser:

The Lexical analyser identifies the following as 'Tokens'.

Keywords: *int, character, main, float, for, while, print, scan, list, matrix, return, if, else.*

Operators: `"+", "-", "*", "/", "=", "@@", "@", "++", "--", '+=' ,
'-=' , "|", "&", "^".`

Comparators: `'==', '<', '>', '<=', '>='.`

Delimiters: `'(', ')', '{', '}', '[', ']', '.', ',',`

Logical operators: `"||", "&&", "!"`

Regular expressions : All the identifiers, function identifiers etc have different regular expressions which can be seen in the lex file.

We also have a global variable **extern char mytext[10000]** which stores the current lexeme value from **yytext** so that we can access the value in subsequent processes.

2) Syntax analysis:

- After **lexical analysis** , we first checks the syntactic structure of the given input,i.e. whether the given input is in the correct syntax (of the language in which the input has been written) or not.It does so by building a data structure, called a **Parse tree** or **Syntax tree**.The Syntax tree is constructed by using the predefined Grammar of the language and the input string.If the given input string can be produced with the help of the syntax tree (in the derivation process),the input string is found to be in the correct syntax.
- This is done in a yacc file which contains the **declarations, rules** and **programs** and defines the actions which should be taken for various cases.

Design of our Syntax Analyser:

Grammar that we have designed for our compiler:

```
1 PROGRAM: DECLARATION_LIST
2
3 DECLARATION_LIST: DECLARATION_LIST DECLARATION | DECLARATION
4
5 DECLARATION: VARIABLE_DECLARATION | FUNCTION_DECLARATION
6
7 VARIABLE_DECLARATION: VARIABLE_TYPE IDENTIFIER_NT SEMICOLON
8 | LIST_TYPE IDENTIFIER_NT OSB NON_N_NUMBER CSB SEMICOLON
9 | MATRIX_TYPE IDENTIFIER_NT OSB INTEGER_NT COMMA INTEGER_NT CSB SEMICOLON
10
11 LIST_TYPE : LIST
12
13 MATRIX_TYPE : MATRIX
14
15 FUNCTION_DECLARATION: VARIABLE_TYPE FUNCTION_IDENTIFIER_NT ONB PARAMS CNB COMPOUND_STATEMENT
16
17 PARAMS: PARAM_LIST_NT | EPSILON
18
19 PARAM_LIST_NT: PARAM_LIST_NT COMMA PARAM | PARAM
20
21 PARAM: VARIABLE_TYPE IDENTIFIER_NT | IDENTIFIER_NT
22
23 STATEMENT_LIST: STATEMENT_LIST STATEMENT | STATEMENT
24
25 STATEMENT: ASSIGNMENT_STATEMENT
26 | COMPOUND_STATEMENT
27 | IF_STATEMENT | WHILE_STATEMENT | FOR_STATEMENT
28 | INCDEC_STATEMENT
29 | LOCAL_DECLARATION
30 | PRINT_STATEMENT | SCAN_STATEMENT | RETURN_STATEMENT
31
32 RETURN_STATEMENT : RETURN IDENTIFIER_NT SEMICOLON | RETURN INTEGER_NT SEMICOLON
33
34 PRINT_STATEMENT : PRINT ONB PRINT_SCAN_ITEM CNB SEMICOLON
35
36 SCAN_STATEMENT : SCAN ONB PRINT_SCAN_ITEM CNB SEMICOLON
37
38 COMPOUND_STATEMENT: OFB STATEMENT_LIST CFB
39
40 IF_STATEMENT: IF ONB EXPRESSION CNB STATEMENT | IF ONB EXPRESSION CNB STATEMENT ELSE STATEMENT
41
42 WHILE_STATEMENT: WHILE ONB EXPRESSION CNB STATEMENT
43
44 FOR_STATEMENT: FOR ONB ASSIGNMENT_STATEMENT EXPRESSION SEMICOLON INCDEC_STATEMENT CNB STATEMENT
45
46 INCDEC_STATEMENT: IDENTIFIER_NT INC SEMICOLON | IDENTIFIER_NT DEC SEMICOLON
47
48 VARIABLE_TYPE: INT | STRING | CHARACTER | FLOAT
```

```
49
50 LOCAL_DECLARATION: VARIABLE_TYPE IDENTIFIER_NT SEMICOLON
51 | LIST_TYPE IDENTIFIER_NT OSB INTEGER_NT CSB SEMICOLON
52 | MATRIX_TYPE IDENTIFIER_NT OSB INTEGER_NT COMMA INTEGER_NT CSB SEMICOLON
53
54 ASSIGNMENT_STATEMENT: IDENTIFIER_NT EQUALTO EXPRESSION SEMICOLON
55
56 EXPRESSION: PEXPRESSION
57 | PEXPRESSION PLUS PEXPRESSION | PEXPRESSION MINUS PEXPRESSION
58 | PEXPRESSION MULTIPLY PEXPRESSION | PEXPRESSION DIVIDE PEXPRESSION
59 | PEXPRESSION BAND PEXPRESSION | PEXPRESSION BOR PEXPRESSION
60 | PEXPRESSION BXOR PEXPRESSION
61 | PEXPRESSION GE PEXPRESSION | PEXPRESSION LE PEXPRESSION
62 | PEXPRESSION GT PEXPRESSION | PEXPRESSION LT PEXPRESSION
63 | PEXPRESSION EE PEXPRESSION | PEXPRESSION NEQ PEXPRESSION
64 | SIZE_EXPRESSION
65 | NOT EXPRESSION
66 | EXPRESSION AND EXPRESSION | EXPRESSION OR EXPRESSION
67
68 SIZE_EXPRESSION : ATSIZE IDENTIFIER_NT
69 | AATSIZE IDENTIFIER_NT
70
71 LIST_ELEMENT : LIST_ELEMENT INTEGER_NT SEMICOLON | INTEGER_NT SEMICOLON
72
73 MATRIX_ELEMENT : MATRIX_ELEMENT MATVAR_NT SEMICOLON | MATVAR_NT SEMICOLON
74
75 MATVAR_NT : MATVAR_NT COMMA INTEGER_NT | INTEGER_NT
76
77 PEXPRESSION: FUNCTION_IDENTIFIER_NT ONB PARAMS CNB
78 | INTEGER_NT | IDENTIFIER_NT
79 | IDENTIFIER_NT OSB INTEGER_NT CSB
80 | IDENTIFIER_NT OSB IDENTIFIER_NT CSB
81 | ONB EXPRESSION CNB
82 | FLOAT_NT | CHARACTER_NT
83 | OSB LIST_ELEMENT CSB | OSB MATRIX_ELEMENT CSB
84
85 PRINT_SCAN_ITEM : IDENTIFIER_NT
86
87 INTEGER_NT: NON_N_NUMBER | N_NUMBER
88
89 FLOAT_NT : FLOATING
90
91 CHARACTER_NT : CHAR
92
93 IDENTIFIER_NT: IDENTIFIER
94 | IDENTIFIER_NT OSB INTEGER_NT CSB | IDENTIFIER_NT OSB IDENTIFIER_NT CSB
95
96 FUNCTION_IDENTIFIER_NT : FUNCTION_IDENTIFIER
```

We can see our abstract syntax tree construction in the yacc.y file.

3) Symbol Table :

We are maintaining all the **variable types** , **variable identifiers** and the **stack address** from **base pointer** in a symbol table :

```
map<pair<string,string>, int> symbol_table;
```

- We store the **identifier** , **type of the variable** and the **relative stack address** from the **base pointer** in the symbol table.
- Whenever we encounter a new symbol, we decrement the **relative stack address** by 8.

Example :

```
int a;  
int b;
```

Our symbol table looks like : [{ {a,"INT",-8} } , { {b,"INT",-16} }

We also did **scoping** and maintaining different **symbol tables for different scopes** (symbol table for each function):

```
vector<map<pair<string,string>, int>> all_scopes_symbol_tables;
```

- We store the symbol tables for different functions in this **all_scopes_symbol_tables**.
- We maintain a separate table to store the **identifier of function** and the **index** of its **symbol table** in the **function_scope_definer**.

```
map<string,int> function_scope_definer
```

4) Semantic analysis:

- After syntax analysis, Semantic analysis is the task of ensuring that the declarations and statements of a program are semantically correct, i.e, that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used.

- Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

Expansion : When a non-terminal is expanded to terminals as per a grammatical rule

Reduction :

- When a terminal is reduced to its corresponding non-terminal according to grammar rules. Syntax trees are parsed top-down and left to right. Whenever reduction occurs, we apply its corresponding semantic rules (actions).
- Semantic analysis uses Syntax Directed Translations to perform the above tasks.
- Semantic analyzer receives AST (Abstract Syntax Tree) from its previous stage (syntax analysis).
- Semantic analyzer attaches attribute information with AST, which is called Attributed AST.

Design of our Semantic Analyser:

We are storing different kinds of objects into nodes emitted by lex using "Union".

```
%union{
    class TreeNode* node;
}

class TreeNode {
public:
    vector<TreeNode*> children;    // children
    string nodeName;              // name of the node
    string lex_val;
    // lexical value, name of identifier etc.
```

```

TreeNode(string NodeName, vector<TreeNode*>
children){
    this->NodeName = NodeName;
    this->children = children;
    this->lex_val="";
}
TreeNode(string NodeName) {
    // for leaf nodes(No children nodes)
    this->NodeName = NodeName;
    children.assign(0, NULL);
    this->lex_val="";
}
};

```

So, each grammar symbol has the structure of a node so that they can pass information among the nodes of the generated tree.

We are

5) Type checking:

- As a part of the semantic analysis, type checking is also implemented.
- After augmenting rules to grammar, we do a traversal on the AST's root node to generate code. We can also check the order of traversal by calling the function "dotraversal" in the main.cpp file.
- We do type checking in this process. For example, if we encounter an assignment statement `a=b`, we check the both the symbol's type in our already built symbol table while We are initially building the AST.
- When we encounter array declaration, we pass the array dimension number along with the identifier into the symbol table. This makes sure that it's only a positive integer. We do this for both 1D and 2D arrays.
- Whenever we encounter an error, we pass it to the `yyerror` function.
- `void yyerror(string temp);`

- So, this raises the error and exits the whole process.

6) Code Generation:

- After Semantic analysis, we move to the code generation part.
- We do a traversal on the constructed AST to generate the assembly code.
- Whenever we start a program, we initialise the stack in assembler.
- This stack expands bottom.
- Now, whenever we encounter any variable declaration statement in the AST traversal, we decrement the stack pointer by 8. Whenever we encounter any assignment statement during the ast traversal , we store the assigned value of the particular variable in the relative address in the stack segment.
- Now, coming to operations (arithmetic and bitwise) in assignment statements, we again get the relative stack address from the symbol table if the operators are identifiers. Else, we use the lexeme value.
- We also do type checking in the process.

Floats:

- **x87** is a floating-point-related subset of the x86 architecture instruction set. It originated as an extension of the 8086 instruction set in the form of optional floating-point coprocessors that worked in tandem with corresponding x86 CPUs.
- Coming to arithmetic operations in floats,
 - We have to use the fpu stack from x87.
 - For example if we have to add two float numbers, we have to first load them into a stack , then perform the operation and then store them.

Loops :

- we use the jump statements to execute them. We have if , if-else, for , while loop constructs in our language.
- We also have support for nested loops with any level of nestedness.

1D & 2D Arrays:

- 1D arrays in our language are called lists.
- If we declared a list (1d array), `List a[5];`
- We store `{{"a","LIST"},-8}` in a symbol table row and `{"a",5}` in a separate table maintained to count the size of the array.
- So, Whenever we are accessing the array element or trying to assign the elements to the list, we access elements accordingly.
- For example if we want to access `a[2]`, we take the base relative stack address of list 'a' from symbol table and add `'2*-8=-16'` to go to the address of `a[2]`.
- Similarly , for the matrix.

Functions:

- Whenever we call a function, we push the current base pointer and then we make the base pointer as the current stack pointer.
- When the function is over, we again restore the stack pointer by equating it to the base pointer and we pop the base pointer equating it to the previous base pointer.
- We also have support for recursive functions with any level of recursion.

7) Register allocation and reducing the 'Register spills' :

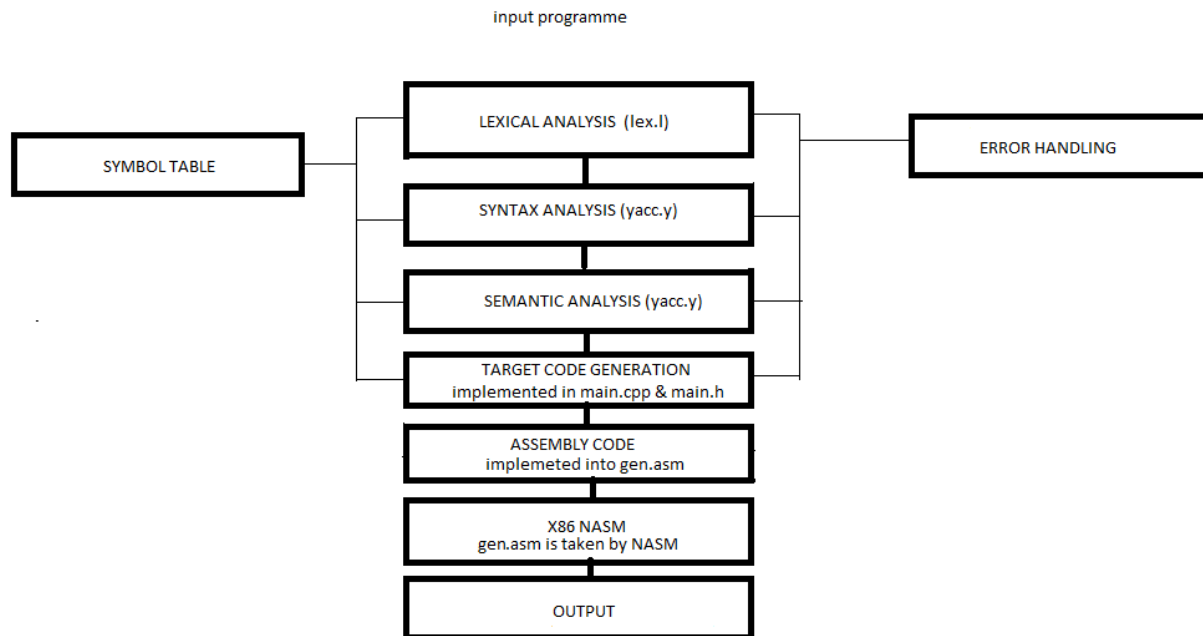
Reducing the memory calls.

- We can see that if we follow the naive approach to load the value of the identifier each and every time we need it, we may end up doing many memory calls which consume a lot of time.
- So, instead of making memory calls, we try to use the values of the identifier that is present in the register already (if the register is not changed and the identifier too).
- So, we need not load a and b from memory stack to registers for a second time again as we already loaded them to registers while we are doing $c=a+b$. So, we directly use the registers that we previously loaded to do operations for c.
- Hence, we are reducing the number of memory loads drastically thus increasing the performance.
- We are also decreasing the 'register pressure'.
So, Whenever we do memory call and store an identifier value in register, we are maintaining a table to store the register and the identifier so that, when we need to use the value of identifier again in the later stage , we try to use the register value directly(if the register value is not changed) instead of again making a memory call.

LRU policy in Register replacement:

- While we are doing the above process and we get to a point that all the registers are full, then we replace the register with the least recently used policy (lru).
- So, When all the registers are holding some identifier from a memory location, we need to
- We wrote this piece of code in the registers.cpp file.

FLOW/INTERACTION BETWEEN DIFFERENT COMPONENTS.



SOURCE CODE ORGANIZATION

The Project code is entirely written in a folder named "CUSTOM_COMPILER".

- **Source files, header files**
 - **Lex.l** is the lex file.
 - **Yacc.y** is the yacc file.
 - **Main.cpp** and **registers.cpp** are cpp files.
 - **Main.h** is a header file.
 - **Makefile**
 - **Test_script.py** is a python file

- **TEST folder**
- **gen.asm** is an asm file.
- **Instructions for building the project and generating the compiler executable**

```
Run "make clean "
    "make"
    "./a.out < TEST/ {file name} "
    "make run nasm"
```

- ({file name} is the name of the file that needs to be tested) to check the output of the particular test case.
- The generated assembly code is in the "**gen.asm**" file.

We also wrote a python script to automate the testing process.

```
Run "make clean "
    "make"
    "python3 test_script.py" or "make autorun"
    "make run nasm"
```

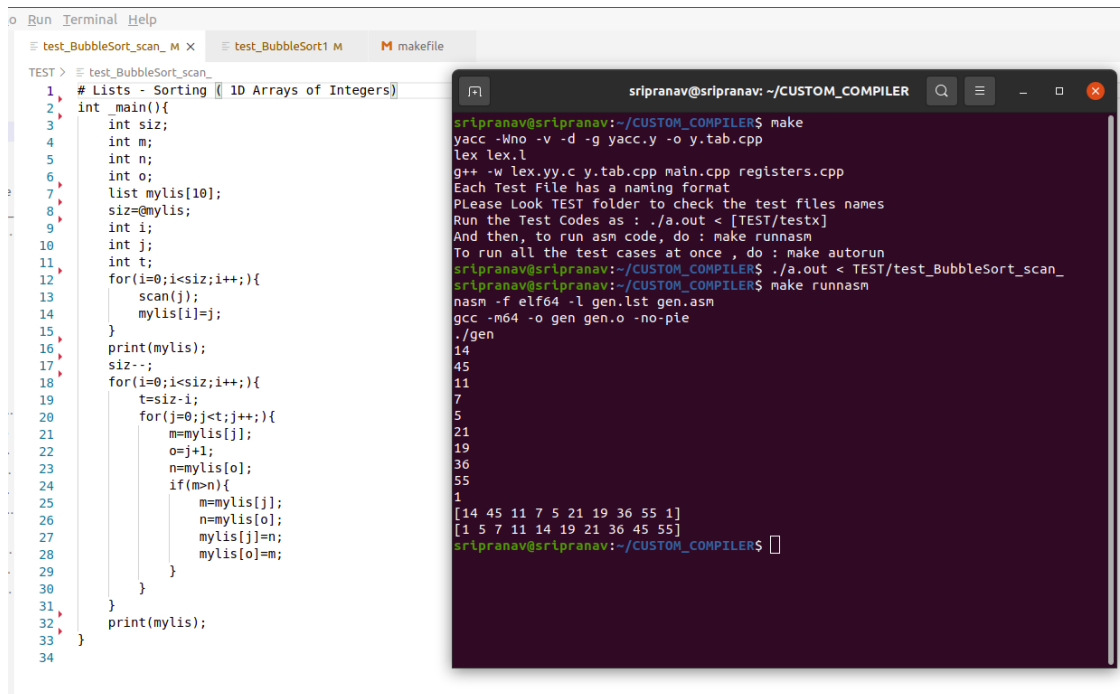
- This script iterates over all the test cases and automatically runs all the general test files.

- **Usage of compiler executable**
 - Use ./a.out < {path of the code} and then it generates assembly code into gen.asm file.
 - Now use make runnasm to run the assembly file and show the output.

FINAL TESTING AND EVALUATION WITH SCREENSHOTS:

1)Bubble Sorting

- taking input from user
- printing the sorted array from the input array



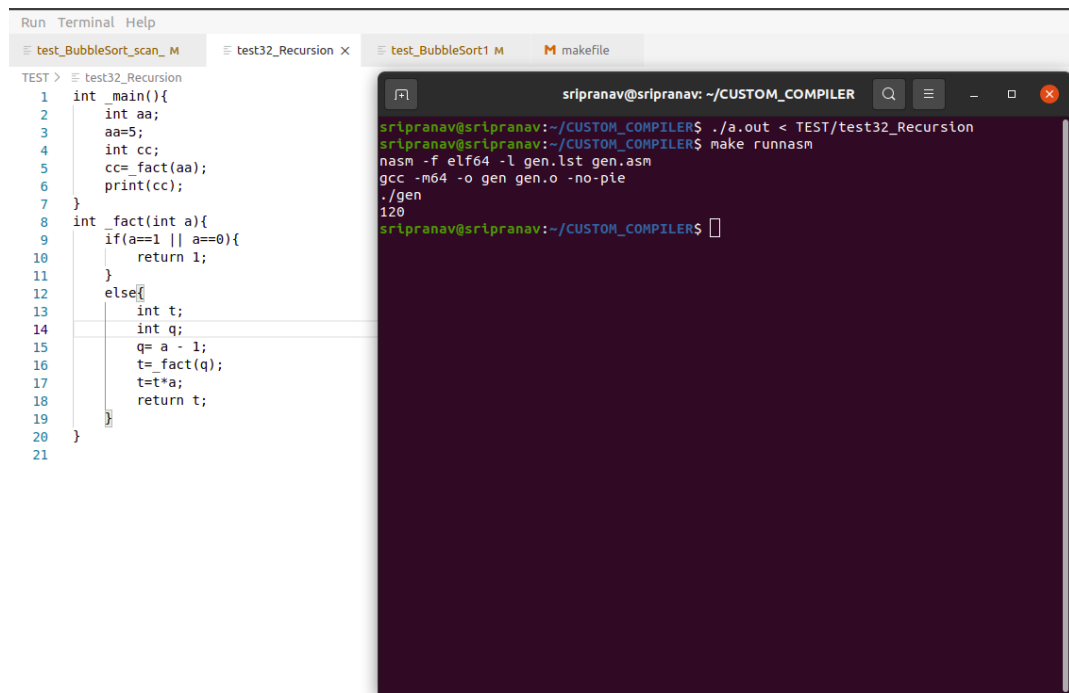
The screenshot shows a code editor with a C program for bubble sort and a terminal window showing the compilation and execution process.

```
1 # Lists - Sorting (1D Arrays of Integers)
2 int _main(){
3     int siz;
4     int m;
5     int n;
6     int o;
7     list mylis[10];
8     siz=@mylis;
9     int i;
10    int j;
11    int t;
12    for(i=0;i<siz;i++){
13        scan(j);
14        mylis[i]=j;
15    }
16    print(mylis);
17    siz--;
18    for(i=0;i<siz;i++){
19        t=siz-i;
20        for(j=0;j<t;j++){
21            m=mylis[j];
22            o=j+1;
23            n=mylis[o];
24            if(m>n){
25                m=mylis[j];
26                n=mylis[o];
27                mylis[j]=n;
28                mylis[o]=m;
29            }
30        }
31    }
32    print(mylis);
33 }
34
```

Terminal Output:

```
sripranav@sripranav: ~/CUSTOM_COMPILER
sripranav@sripranav:~/CUSTOM_COMPILER$ make
yacc -Wno -v -d -g yacc.y -o y.tab.cpp
lex lex.l
g++ -w lex.yy.c y.tab.cpp main.cpp registers.cpp
Each Test File has a naming format
Please Look TEST folder to check the test files names
Run the Test Codes as : ./a.out < [TEST/testx]
And then, to run asm code, do : make runnasm
To run all the test cases at once , do : make autorun
sripranav@sripranav:~/CUSTOM_COMPILER$ ./a.out < TEST/test_BubbleSort_scan_
sripranav@sripranav:~/CUSTOM_COMPILER$ make runnasm
nasm -f elf64 -l gen.lst gen.asm
gcc -m64 -o gen gen.o -no-pie
./gen
14
14
45
11
7
5
21
19
36
55
1
[14 45 11 7 5 21 19 36 55 1]
[1 5 7 11 14 19 21 36 45 55]
sripranav@sripranav:~/CUSTOM_COMPILER$
```

2)Computing Factorial Using Recursion:



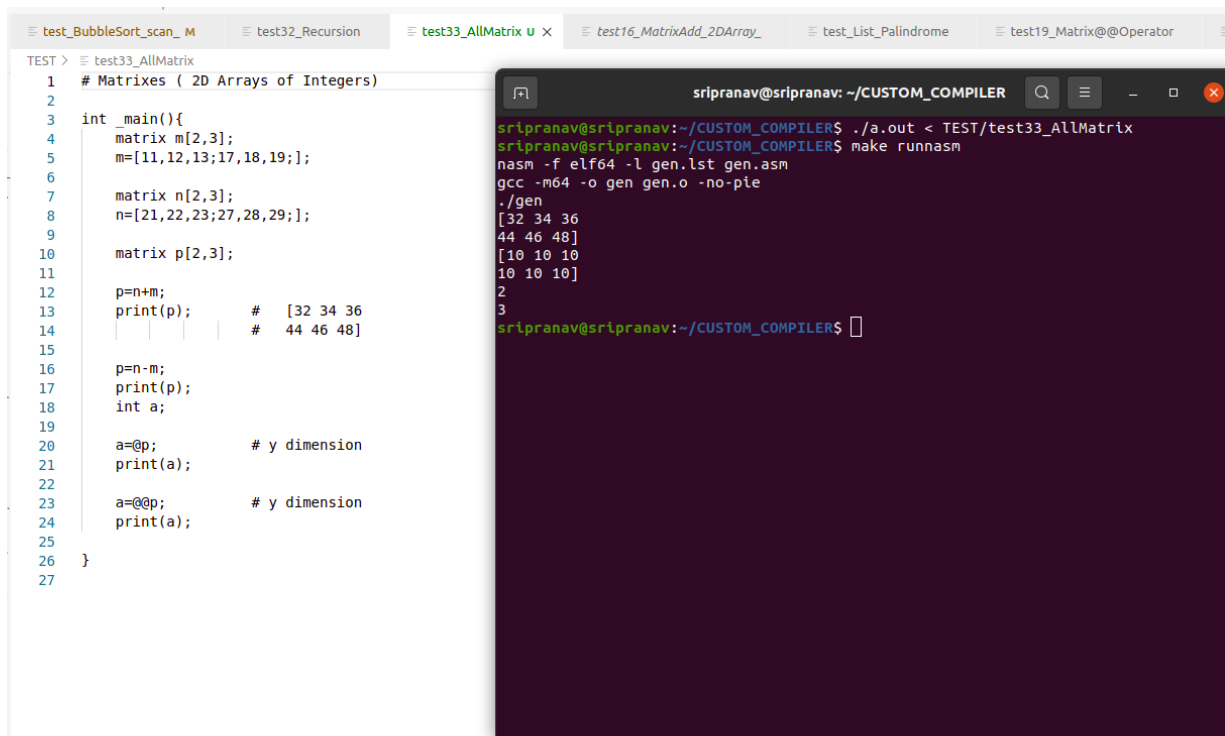
The screenshot shows a code editor with a C program for computing factorial using recursion and a terminal window showing the compilation and execution process.

```
1 int _main(){
2     int aa;
3     aa=5;
4     int cc;
5     cc=_fact(aa);
6     print(cc);
7 }
8 int _fact(int a){
9     if(a==1 || a==0){
10         return 1;
11     }
12     else{
13         int t;
14         int q;
15         q= a - 1;
16         t= _fact(q);
17         t=t*a;
18         return t;
19     }
20 }
21
```

Terminal Output:

```
sripranav@sripranav: ~/CUSTOM_COMPILER
sripranav@sripranav:~/CUSTOM_COMPILER$ ./a.out < TEST/test32_Recursion
sripranav@sripranav:~/CUSTOM_COMPILER$ make runnasm
nasm -f elf64 -l gen.lst gen.asm
gcc -m64 -o gen gen.o -no-pie
./gen
120
sripranav@sripranav:~/CUSTOM_COMPILER$
```

3) Showing Matrix (2d) Operations (Along With @, @@ Operators)

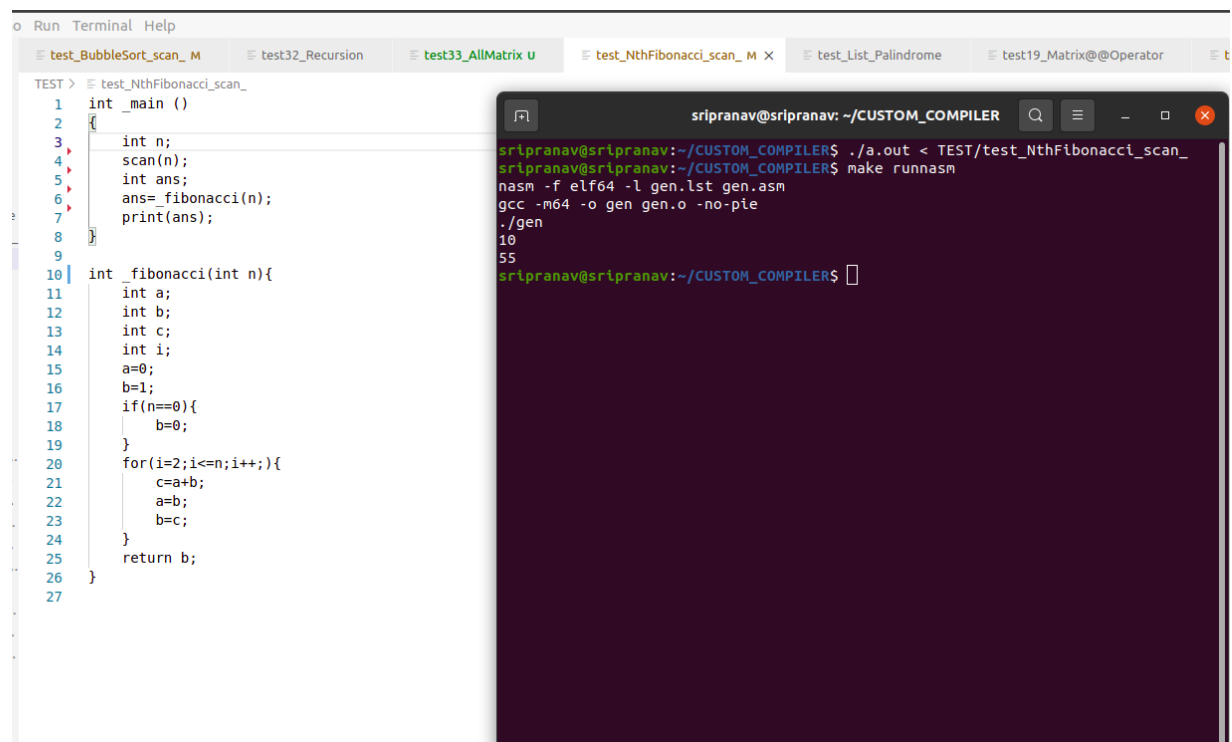


The screenshot shows an IDE with a C program named `test33_AllMatrix.c` and a terminal window. The C code defines two 2D arrays, `m` and `n`, and performs operations using the `@` and `@@` operators. The terminal window shows the compilation and execution of the program, displaying the output of the matrix operations.

```
1 # Matrices ( 2D Arrays of Integers)
2
3 int _main(){
4     matrix m[2,3];
5     m=[11,12,13;17,18,19;];
6
7     matrix n[2,3];
8     n=[21,22,23;27,28,29;];
9
10    matrix p[2,3];
11
12    p=n+m;
13    print(p);      # [32 34 36
14                    # 44 46 48]
15
16    p=n-m;
17    print(p);
18    int a;
19
20    a=@p;           # y dimension
21    print(a);
22
23    a=@@p;          # y dimension
24    print(a);
25
26 }
27
```

```
sripranav@sripranav: ~/CUSTOM_COMPILER
sripranav@sripranav:~/CUSTOM_COMPILER$ ./a.out < TEST/test33_AllMatrix
sripranav@sripranav:~/CUSTOM_COMPILER$ make runnasm
nasm -f elf64 -l gen.lst gen.asm
gcc -m64 -o gen gen.o -no-pie
./gen
[32 34 36
44 46 48]
[10 10 10
10 10 10]
2
3
sripranav@sripranav:~/CUSTOM_COMPILER$
```

4) Computing Nth Fibonacci Number By Taking Input From User.

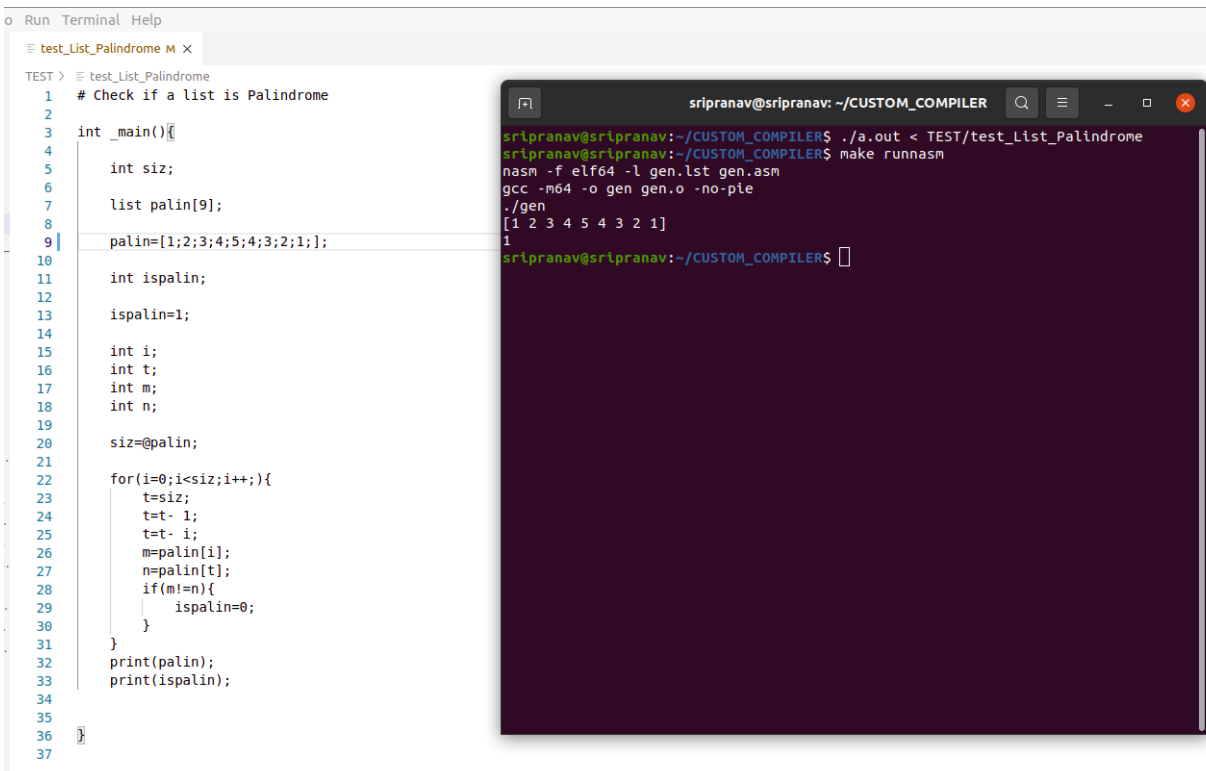


The screenshot shows an IDE with a C program named `test_NthFibonacci_scan_M.c` and a terminal window. The C code takes an input `n` and computes the `n`th Fibonacci number using a loop. The terminal window shows the compilation and execution of the program, displaying the output of the Fibonacci number calculation.

```
1 int _main ()
2 {
3     int n;
4     scan(n);
5     int ans;
6     ans= fibonacci(n);
7     print(ans);
8 }
9
10 int fibonacci(int n){
11     int a;
12     int b;
13     int c;
14     int i;
15     a=0;
16     b=1;
17     if(n==0){
18         b=0;
19     }
20     for(i=2;i<=n;i++){
21         c=a+b;
22         a=b;
23         b=c;
24     }
25     return b;
26 }
27
```

```
sripranav@sripranav:~/CUSTOM_COMPILER
sripranav@sripranav:~/CUSTOM_COMPILER$ ./a.out < TEST/test_NthFibonacci_scan_
sripranav@sripranav:~/CUSTOM_COMPILER$ make runnasm
nasm -f elf64 -l gen.lst gen.asm
gcc -m64 -o gen gen.o -no-pie
./gen
10
55
sripranav@sripranav:~/CUSTOM_COMPILER$
```

5) Checking if a list is palindrome or not.



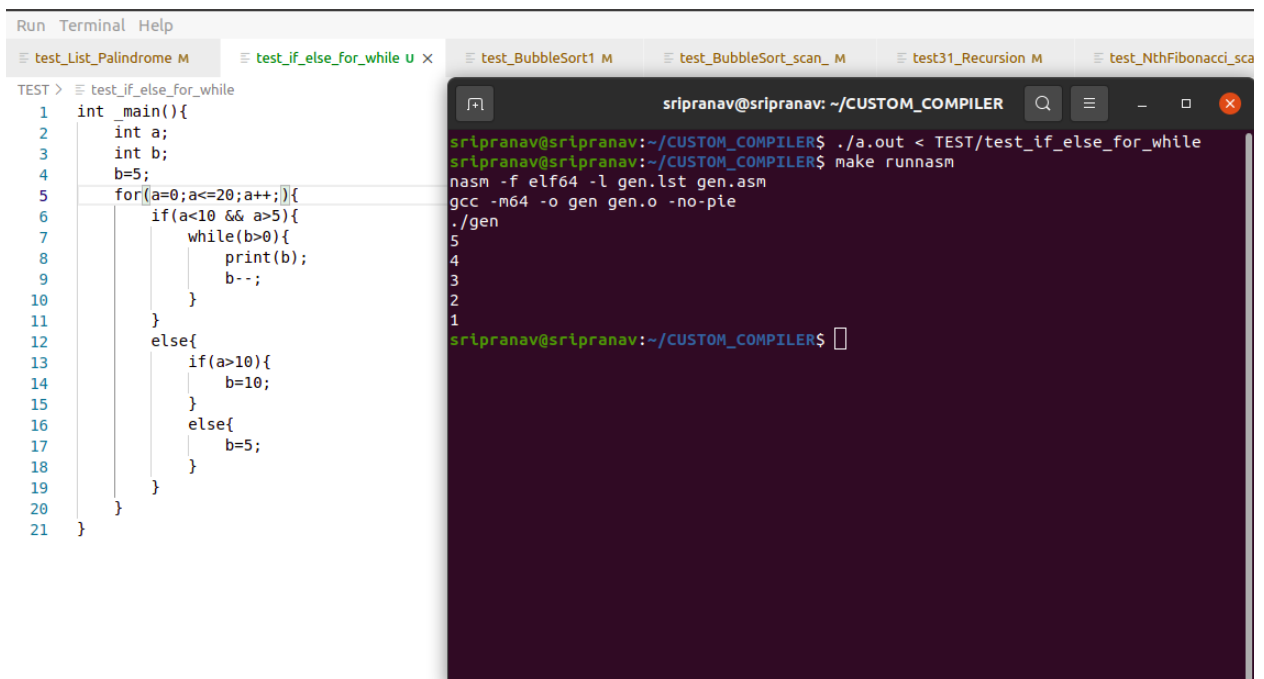
The screenshot shows a code editor with a C program to check if a list is a palindrome. The program is named `test_List_Palindrome`. It defines a `main` function that initializes an array `palin` with the values `[1, 2, 3, 4, 5, 4, 3, 2, 1]`. It then calls a function `ispalin` to check if the list is a palindrome. The `ispalin` function uses a two-pointer approach, comparing elements from both ends of the array towards the center. The output of the program is printed to the console.

```
1 # Check if a list is Palindrome
2
3 int _main(){
4     int siz;
5     list palin[9];
6
7     palin=[1;2;3;4;5;4;3;2;1;];
8
9     int ispalin;
10
11     ispalin=1;
12
13     int i;
14     int t;
15     int m;
16     int n;
17
18     siz=@palin;
19
20     for(i=0;i<siz;i++){
21         t=siz;
22         t=t- 1;
23         t=t- i;
24         m=palin[i];
25         n=palin[t];
26         if(m!=n){
27             ispalin=0;
28         }
29     }
30
31     print(palin);
32     print(ispalin);
33
34 }
35
36
37
```

The terminal output shows the execution of the program:

```
sripranav@sripranav: ~/CUSTOM_COMPILER
sripranav@sripranav:~/CUSTOM_COMPILER$ ./a.out < TEST/test_List_Palindrome
sripranav@sripranav:~/CUSTOM_COMPILER$ make runnasm
nasm -f elf64 -l gen.lst gen.asm
gcc -m64 -o gen gen.o -no-pie
./gen
[1 2 3 4 5 4 3 2 1]
1
sripranav@sripranav:~/CUSTOM_COMPILER$
```

6) Checking conditional, iterative statements with nesting.



The screenshot shows a code editor with a C program named `test_if_else_for_while`. The program defines a `main` function that initializes variables `a` and `b`. It then enters a `for` loop that iterates from `a=0` to `a=20`. Inside the loop, it uses nested `if` and `while` statements to check conditions on `a` and `b`. The output of the program is printed to the console.

```
1 int _main(){
2     int a;
3     int b;
4     b=5;
5     for(a=0;a<=20;a++){
6         if(a<10 && a>5){
7             while(b>0){
8                 print(b);
9                 b--;
10            }
11        }
12        else{
13            if(a>10){
14                b=10;
15            }
16            else{
17                b=5;
18            }
19        }
20    }
21 }
```

The terminal output shows the execution of the program:

```
sripranav@sripranav: ~/CUSTOM_COMPILER
sripranav@sripranav:~/CUSTOM_COMPILER$ ./a.out < TEST/test_if_else_for_while
sripranav@sripranav:~/CUSTOM_COMPILER$ make runnasm
nasm -f elf64 -l gen.lst gen.asm
gcc -m64 -o gen gen.o -no-pie
./gen
5
4
3
2
1
sripranav@sripranav:~/CUSTOM_COMPILER$
```


TEST PLAN :

Test Items:

We have all the test programs in the "TEST" folder. We have 33 programs + 7 general programs.

Features To Be Tested :

- Integer declaration
- scanning
- printing
- arithmetic and bitwise operations on integers
- float declaration
- arithmetic operations on floats
- character declaration
- character printing
- if
- if-else
- while loops
- for loops
- list (1D array) declarations
- list element accessing and modification
- list printing
- list arithmetics
- list size operator
- matrix (2D array) declarations
- matrix printing
- matrix arithmetics
- functions with integer return values with any number of arguments
- Least recently used register policy to reduce memory calls.
- Function Recursion.

Approach :

- We wrote all the test cases in the TEST folder.
Run **"make clean "**
"Make"
"./a.out < TEST/ {file name} "
"Make run nasm"
- ({file name} is the name of the file that needs to be tested),to check the output of the particular test case.
- The generated assembly code is in the **"gen.asm"** file.
- We also wrote a python script to automate the testing process.
Run **" make clean"**
" make"
" python3 test_script.py" or "make autorun"
- This script iterates over all the test cases and automatically runs all the general test files.

Item Pass/Fail Criteria :

If the output matches the expected output for the program , then we can say that the test case is passed. (assuming the program follows our specified grammar. If not, an error is generated. Also , scope of our project is to be kept in mind.We discussed our scope in our language manual and report)

Testing Tasks:

We made the following test programs to test all the features of our compiler.

General Test Cases

In the TEST folder,

- 1) **test 1 to test 8** checks basic integer and float operations.
- 2) **test 9 to test 14** checks list operations.
- 3) **test 15 to test 19** checks Matrix operations.
- 4) **test 20 to test 23** checks loops.
- 5) **test 24 to test 26** checks functions.
- 6) **Test 28** checks character printing.
- 7) **Test 29 and test 30** checks list sorting in 1 dimension.
- 8) **Test 27** checks register allocation and with reduced memory calls.
- 9) Test cases also cover function recursion.

Real Life Test Cases

In the TEST folder,

- 1) **test_BubbleSort_scan_** : takes Array(1D) input from the user and outputs the sorted array and also the given input. We used the bubble sort algorithm to do this.
- 2) **test_Bubble Sort2**: checks the code bubble sort on a given array.
- 3) **test_NthFibonacci** : finds the Fibonacci of a given number which is scanned (user gives the input) and prints the nth Fibonacci .
- 4) **test_NFactorial** : finds the Factorial of given number which is scanned (user gives the input) and prints the factorial .
- 5) **test_Gcd_scan** : finds the GCD of two numbers which are scanned (user gives the input) and prints the GCD .

- 6) **test_List_Palindrome** :checks if list is palindrome or not.
- 7) **test_Fact_Recursion** : shows function recursion.
- 8) **test31_Recursion** : shows another recursive function.

Testing Process Automation :

We also have a python script "**test_script.py**" which iterates over all the general test cases and shows their output on shell.(One at a time with 3 seconds gap)

This script **automates the testing process.**

LIMITATIONS OF THE COMPILER

- We only implemented one dimensional and 2 dimensional arrays.
- No dynamic memory allocation.
- We don't have classes.
- We don't have dynamic arrays, maps etc.

CONCLUSION:

- We learnt a lot during the entire development of the project. We learnt various phases involved in the compilation process, various data structures implementation , deep understanding of how assembly language works and most importantly we got our question of "How a programming language is understood by a computer" answered. We also spent a lot of time discussing the project with the peers , understanding the flow of code and much more.

- We also got to implement the theories that we learnt in the compiler design theory course.
- Additionally, we also got an experience to work with big pieces of code and complex tools.
- We also got experience of design and development of large software systems.
- The techniques learned will be useful in many tasks.

CONTRIBUTIONS:

Sripranav Mannepalli - Lexical analysis, Syntax analysis, Semantic Analysis, code generation(x86 assembly), error handling, report , language manual, demo video.

S R V S Maheswara reddy - Lexical analysis, Syntax analysis, Semantic Analysis, code generation(x86 assembly), error handling, report , language manual, demo video.

Chirag Gupta - Lexical analysis, Syntax analysis, Semantic Analysis, code generation(x86 assembly), error handling.

A V S Hrudai - Lexical analysis, Syntax analysis, error handling, report , language manual.