CD Compiler Project Report

Jawahar Sai Nathani - CS18B023
Sagar Reddy P - CS18B025
Abhilash - CS18B039
Likith Kumar - CS18B019

1. Introduction

As part of the Compiler Design Lab course project we, Team 9 of the 3rd year CSE branch have designed a compiler for custom programming language all the way from syntax analysis to assembly code generation using lex and yacc tools. In our compiler tokenization of input stream is implemented using lex tool, syntax and semantic analysis is implemented using yacc tool, Intermediate code and Assembly code are manually generated after semantic analysis.

This compiler supports Data types like integers, characters, strings, floats. It supports Functions and Loops - For, While, do While. It also supports Conditional Statements (If Else), Simple and Complex Arithmetic Operations, Relational Operations. This report explains major parts of the compiler like tools used, design of compiler, how it works, limitations of the compiler.

2. Language and Tool Choices

a. Tools

i. **LEX** - *LEX* tool is used to divide the input stream into tokens and send those tokents to the parser for syntax and semantic analysis. Basically, the *LEX* tool is used to implement the *scanner* of the compiler.

ii. **Yacc -** Yacc file is used to check syntax of the input code and construct a parse tree using the tokens from the scanner.

b. Language Choice - *C*

i. C language is used to implement all the other major components of the compiler like maintaining symbol table and constant table, semantic analysis, generating three address code and generating assembly code

3. Design

a. Scanner

i. Scanner is used to convert the input stream into tokens and identify them as identifiers, operators, keywords, constants and these identified tokens will be used for syntax analysis by parser.
ii. It is implemented using the lex tool and C language.
iii. Regex are defined to tokenize the input stream.

b. Parser

i. Parser is implemented with the help of Yacc tool and C language.
ii. It is used to check the syntax of the input stream using the grammar rules and the tokens from the scanner.
iii. YACC tool is designed to compile LALR(1) grammar.

c. Symbol table

i. Symbol table is used to maintain and store all the identifiers and their type,value, line numbers, scope information.
ii. Symbol table is implemented using the most efficient Hashing mechanism to decrease the compilation time.
iii. Symbol table is also used for some semantic analysis tasks like finding undeclared variables, redeclared variables, to check the type of identifiers etc..
iv. Symbol table stores parameters count and all the parameters for function identifiers.
v. Constant table is used to store all the constant values like integers, floating point numbers, strings and characters.
vi. Constant table is implemented using the Hashing mechanism.

d. Semantic analyzer

i. Semantic analyzer is developed using C functions and predefined C libraries.
ii. After syntax analysis, identifiers will be sent to the semantic analyzer. Semantic analyzer verifies variable declarations, sizes of arrays and other semantic elements. Error Message will be displayed in case of any errors.

iii. After semantic analysis, the semantically correct identifiers, their values and their operators are stored in a stack and this stack is used to generate Intermediate code using the functions that are called after semantic analysis.

e. Type checker

i. As part of semantic analysis, the LHS and RHS of every expression are compared, if their types do not match a specific Error Message will be displayed.
ii. Type checker is implemented with the help of symbol table and few C libraries.

f. Intermediate code generator

i. Choice of Intermediate code for our compiler is *Three Address code*.
ii. Intermediate code generation is performed after semantic analysis.
iii. All the compound and complex expressions are divided into simple expressions which will be converted into Intermediate code.
iv. For every simple expression a new temporary variable is created to store the value of the simple expression, and this new temporary variable which plays the role of simple expression in the complex expression, is pushed into a stack that stores all the identifiers,operators and temporary variables.
v. After storing the identifiers, temporary variables and the connecting operator for identifier and temporary variable, three address code is generated using some C functions based on the connecting operator.

g. Code generator

i. Choice of assembly language is X86 Nasm.
ii. Code generation is performed after Intermediate code generation.
iii. The Intermediate code generated is manually converted into x86 assembly code using C language and few predefined C libraries.
iv. The whole x86 assembly language for the input stream is manually converted from intermediate code.
v. Few bytes are reserved in the memory to store all the values of the identifiers. And the addresses for these identifiers are stored in a struct that maintains the name of the identifier and the address of the identifier in the main memory.

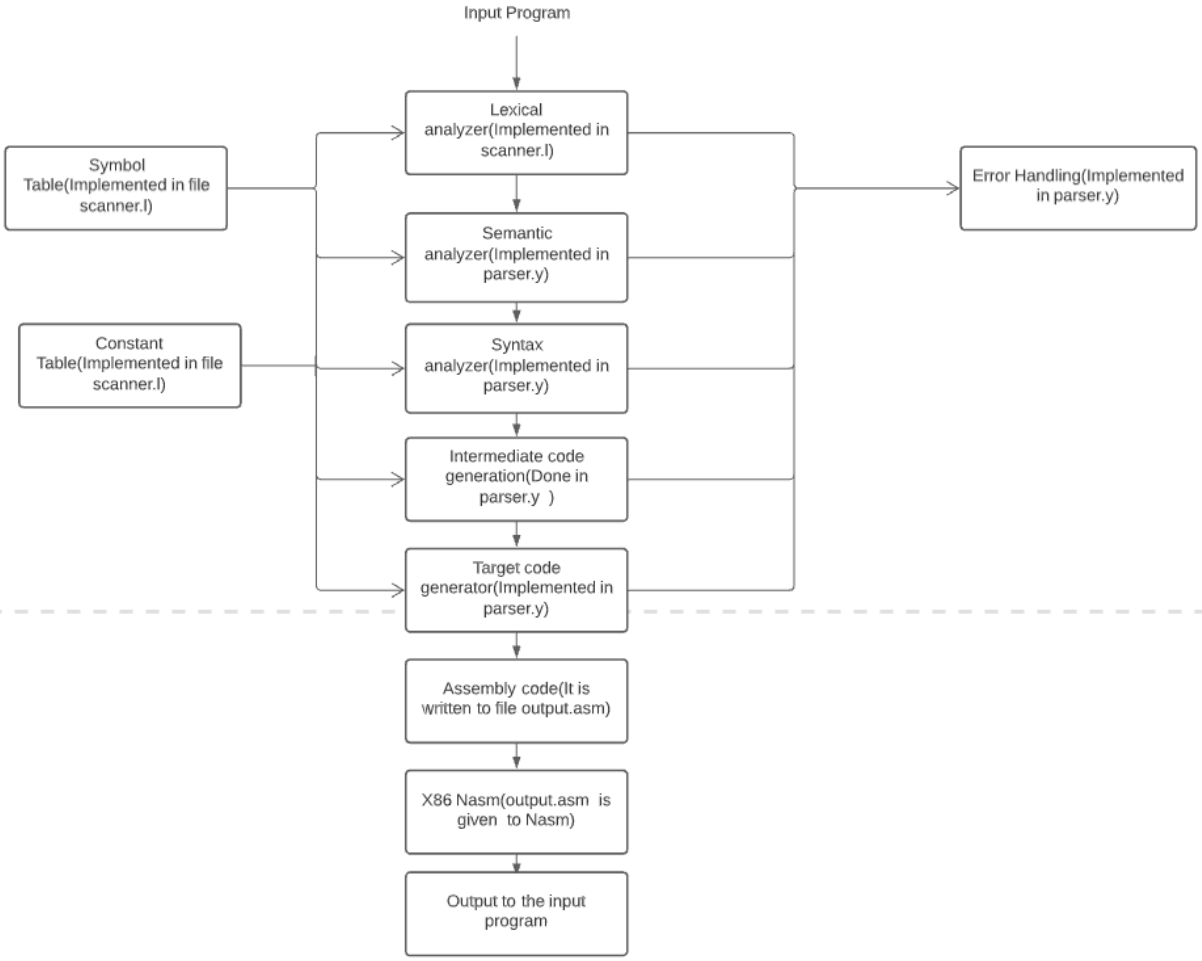
vi. Whenever an identifier is used in any operation, the value of that identifier is moved into a register using the address stored in the struct and the value of the register is used for the respective operation.

vii. As the string and character identifiers only support declaration and print statements, the value of the string identifier is directly displayed using the address of the identifier from the main memory.

viii. Similar to identifiers, few bytes are reserved in the memory to store arrays. And to store the address for these arrays is stored a new struct is maintained.

ix. Finally, the completely generated X86 assembly language is directed into the "*output.asm*" file.

4. Flow/interactions between different components



5. Source Code Organization

a. Source Files : *"scanner.l"* , *"parser.y" and "Makefile"* are the source files required.
b. Header Files : *stdio.h* , *string.h* and *stdlib.h*
c. *Flex* and *Yacc* are the prerequisites required to run the compiler.

d. Usage

i. Test Cases

1. *>> make clean*
2. *>> make compiler*
3. Run *>> make <test case>* to test already defined test cases.
4. *>> make assembly*

ii. New Test Case

1. *>> make clean*
2. *>> make compiler*
3. *>> ./compiler <name of input file>*
4. *>> make assembly*

6. Testing

a. This compiler has been tested on a few test cases that cover all the above mentioned functionalities like functions, arrays, scan statements, print statements, arithmetic operations, relational operators, complex operations etc..
b. It is also tested on a few invalid input streams for implementing basic error handling techniques.

a. Test Case 1

This Sample Program covers simple and complex arithmetic operations using integer values. Print Statements for integers, strings. Scan Statements for Integer values.

```
Int main()
{

    Int a = 20;
    Int b = 6;

    Int c = a % b;
    Print("Remainder: %I\n",c);

    c = a + b;
    Print("Add: %I\n",c);

    c = a - b;
    Print("Sub: %I\n",c);

    c = a * b;
    Print("Mul: %I\n",c);

    String s = "Enter values a and b: ";

    Print("%S\n",s);
    Scan(a,b);
    Print("Entered values: %I - %I\n",a,b);

    c = (((a+b)*(a-b))+a-b)/b;
    Print("Complex: %I\n",c);

    If(a>b)
    {
        Print("a is greater than b\n");
    }
```

```
    Else
    {
        Print("a is not greater than b\n");
    }

    If(a<b)
    {
        Print("b id greater than a\n");
    }
    Else
    {
        Print("b is not greater than a\n");
    }

}
```

Output :

```
gamemaster@gamemaster-VirtualBox:~/Desktop/CD_Final_Compiler$ make arithm
./compiler TestCases/Executable/arithm.cd
gamemaster@gamemaster-VirtualBox:~/Desktop/CD_Final_Compiler$ make assembly
nasm -f elf64 -o out.o output.asm
output.asm:222: warning: character constant too long [-w+other]
ld out.o -o out
./out
Remainder: 2
Add: 26
Sub: 14
Mul: 120
Enter values a and b:
12
4
Entered values: 12 - 4
Complex: 34
a is greater than b
b is not greater than a
gamemaster@gamemaster-VirtualBox:~/Desktop/CD_Final_Compiler$
```

b. Test Case 2

This Sample program covers declaration of array, For loop, arithmetic operations with array elements, printing array elements with Print statement.

```
Int main()
{
    Int a[5] = {2,15,1,3,4};
    Int i = 0;
    Int j = 0;
    For(i=0;i<5;i++)
    {
        j = a[i-1] + j;
    }
    Print("Sum of values of array a is: %I\n",j);

    a[0] = 10;
    a[1] = 12;
    a[2] = 0;

    Print("First 3 values of array a are: %I %I %I\n",a[0],a[1],a[2]);
}
```

Output :

```
gamemaster@gamemaster-VirtualBox:~/Desktop/CD_Final_Compiler$ make as
./compiler TestCases/Executable/arrays_sum.cd
gamemaster@gamemaster-VirtualBox:~/Desktop/CD_Final_Compiler$ make assembly
nasm -f elf64 -o out.o output.asm
ld out.o -o out
./out
Sum of values of array a is: 25
First 3 values of array a are: 10 12 0
gamemaster@gamemaster-VirtualBox:~/Desktop/CD_Final_Compiler$
```

c. Test Case 3.

This Sample code covers sorting technique for an array using Insertion sort.

```
Int main()
{
    Int a[9] = {2,9,7,3,10,54,100,1,0};
    Int len = 9;
    Int i = 0;


    Print("Initial Array => %I",a[0]);
```

```
    For(i=1;i<len;i++)
    {
        Int k = 0;
        k = a[i-1] + 0;
        Print(" - %I",k);
    }
    Print("\n");


    For(i=0;i<(len-1);i++)
    {
        Int j = 0;
        For(j=i;j<len;j++)
        {
            Int k = a[i-1] + 0;
            Int p = a[j-1] + 0;
            If(k>p)
            {
                a[j-1] = k;
                a[i-1] = p;
            }
        }
    }


    Print("Sorted Array  => %I",a[0]);
    For(i=1;i<len;i++)
    {
        Int k = 0;
        k = a[i-1] + 0;
        Print(" - %I",k);
    }
    Print("\n");

}
```

Output :

```
gamemaster@gamemaster-VirtualBox:~/Desktop/CD_Final_Compiler$ make sortingarray
./compiler TestCases/Executable/sorting_arrays.cd
gamemaster@gamemaster-VirtualBox:~/Desktop/CD_Final_Compiler$ make assembly
nasm -f elf64 -o out.o output.asm
ld out.o -o out
./out
Initial Array => 2 - 9 - 7 - 3 - 10 - 54 - 100 - 1 - 0 - 234
Sorted Array  => 0 - 1 - 2 - 3 - 7 - 9 - 10 - 54 - 100 - 234
gamemaster@gamemaster-VirtualBox:~/Desktop/CD_Final_Compiler$
```

d. Test Case 4

This Sample program covers declaration of functions, void function types, Int function types with return statements, recursive functions.

```
Void pr(Int d)
{
    String s = "Value received is:";
    Print("%S %I\n",s,d);
}

Void printfib(Int l, Int q, Int r, Int y)
{
    If(r<=y)
    {
        Int g = l + q;
        Print("%I ",g);
        r++;
        printfib(q,g,r,y);
    }

}

Int addtwonumbers(Int i, Int j)
{
    pr(i);
    pr(j);
    i = i + j;
    return i;
}

Int main()
{
    Int a = 1;
    Int b = 1;
    Int k = addtwonumbers(a,b);
    Print("Value of k is: %I\n",k);
    Int c = 15;
    Int d = 3;
    Print("Fibonacci series: %I %I ",a,b);
    printfib(a,b,d,c);
    Print("\n");
}
```

Output :

```
gamemaster@gamemaster-VirtualBox:~/Desktop/CD_Final_Compiler$ make function
./compiler TestCases/Executable/functions.cd
gamemaster@gamemaster-VirtualBox:~/Desktop/CD_Final_Compiler$ make assembly
nasm -f elf64 -o out.o output.asm
output.asm:21: warning: character constant too long [-w+other]
ld out.o -o out
./out
Value recieved is: 1
Value recieved is: 1
Value of k is: 2
Fibonacci series: 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
gamemaster@gamemaster-VirtualBox:~/Desktop/CD_Final_Compiler$
```

e. Test case 5

This Sample code covers while loop and do while loop.

```
Int main()
{

    Int a = 0;
    Int i = 0;

    While(i<10)
    {
        a= a + 2;
        i++;
    }
    Print("Value of a is: %I\n",a);

    do{
        a = a - 1;
    }While(a>20);

    Print("Value of a is: %I\n",a);

}
```

Output :

```
gamemaster@gamemaster-VirtualBox:~/Desktop/CD_Final_Compiler$ make dowhile
./compiler TestCases/Executable/dowhile.cd
gamemaster@gamemaster-VirtualBox:~/Desktop/CD_Final_Compiler$ make assembly
nasm -f elf64 -o out.o output.asm
ld out.o -o out
./out
Value of a is: 20
Value of a is: 19
gamemaster@gamemaster-VirtualBox:~/Desktop/CD_Final_Compiler$
```

f. Test case 6

This Sample program covers If Else-if and Else conditional statements and nested If Else Statements.

```
Int main()
{

    Int a = 12;
    Int b = 4;

    If(a>5 and b>5)
    {
        Print("1\n");
    }
    Else If(a>5 or b>5)
    {
        If(a<5)
        {
            Print("2.1\n");
        }
        Else If(b>5)
        {
            Print("2.2\n");
        }
        Else{
            Print("2.3\n");
        }
    }
    Else{
        Print("3\n");
    }

}
```

Output :

```
gamemaster@gamemaster-VirtualBox:~/Desktop/CD_Final_Compiler$ make ifelse
./compiler TestCases/Executable/ifelse.cd
gamemaster@gamemaster-VirtualBox:~/Desktop/CD_Final_Compiler$ make assembly
nasm -f elf64 -o out.o output.asm
ld out.o -o out
./out
2.3
gamemaster@gamemaster-VirtualBox:~/Desktop/CD_Final_Compiler$
```

7. Limitations

a. Function :
   i. Only supports *Int* and *Void* type.
   ii. Each function can return only one integer value.
   iii. Input variables can be only of type *Int*.
b. Float :
   i. Floating values only support declaration and print statement.
c. String and Char :
   i. *String* and *Char* identifiers only support declaration and print statement.
d. Arrays :
   i. This compiler only supports one dimensional arrays.
   ii. Arrays only support type *Int.*
e. Scan
   i. Scan statement only supports *Int* values.
f. Error Handling
   i. This compiler only displays basic errors. It can be extended to display errors more clearly and also support error recovery techniques.

8. Conclusion

This whole Compiler Design Lab Project is an exciting ride of learning all the phases involved in the compilation process. It exposed us to the very basic and very complex places of the compiler. It improved our level of understanding of scanner and parser and how the input stream is converted into tokens and how syntax analyser works, how semantics of the input stream is verified.From Intermediate code generation and Code generation sections we have learned the usage of registers and code optimisation techniques and many other concepts.

Apart from the course related topics we have improved our skills as part of the team. Our team discussions helped us improve our team player skills in communication, splitting the work, understanding each other's work, which we hope would help in our career. The Agile methodology that we followed helped us in completing the project in an effective and efficient manner.

9. Contributions

a. Jawahar sai :

i. Worked on Intermediate code generation(three address code) , conversion of Intermediate code to x86 assembly code, Demo video.

b. Sagar reddy :

i. Worked on Parser, Grammar rules, Semantic analysis, Intermediate Code generation.

c. Likith kumar raju :

i. Worked on scanner, symbol table, maintaining scope of identifiers and methods to check their scope in compound statements, Language specifications.

d. Abhilash :

i. Worked on scanner, symbol table, maintaining scope of identifiers and methods to check their scope in compound statements, Language specifications.

e. Everyone :

i. Worked on Error Handling techniques, issues in the code.

ii. Report, Language Reference manual.