



Report on

Mini R compiler

*Submitted in partial fulfilment of the requirements for **Sem VI***

Compiler Design Laboratory

**Bachelor of Technology
in
Computer Science & Engineering**

Submitted by:

Ganesha K S

PES1201701731

Under the guidance of

Preet Kanwal

Assistant Professor, Dept. of CSE
PES University, Bengaluru

Ashwini M Joshi

Assistant Professor, Dept. of CSE
PES University, Bengaluru

January – May 2020

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY**

(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

TABLE OF CONTENTS

Chapter No.	Title	Page No.
1.	INTRODUCTION (Mini-Compiler is built for which language. Provide sample input and output of your project)	01
2.	ARCHITECTURE OF LANGUAGE: <ul style="list-style-type: none"> What all have you handled in terms of syntax and semantics for the chosen language. 	02
3.	LITERATURE SURVEY (if any paper referred or link used)	03
4.	CONTEXT FREE GRAMMAR (which you used to implement your project)	3-4
5.	DESIGN STRATEGY (used to implement the following) <ul style="list-style-type: none"> SYMBOL TABLE CREATION ABSTRACT SYNTAX TREE INTERMEDIATE CODE GENERATION CODE OPTIMIZATION ERROR HANDLING - strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyser, and code generator). TARGET CODE GENERATION 	4-10
6.	IMPLEMENTATION DETAILS (TOOL AND DATA STRUCTURES USED in order to implement the following): <ul style="list-style-type: none"> SYMBOL TABLE CREATION ABSTRACT SYNTAX TREE (internal representation) INTERMEDIATE CODE GENERATION CODE OPTIMIZATION ASSEMBLY CODE GENERATION ERROR HANDLING - strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyser, and code generator). Provide instructions on how to build and run your program. 	10-17
7.	RESULTS AND possible shortcomings of your Mini-Compiler	17
8.	SNAPSHOTS (of different outputs)	18-21
9.	CONCLUSIONS	22
10.	FURTHER ENHANCEMENTS	22
REFERENCES/BIBLIOGRAPHY		22

Introduction

The compiler that is built helps to convert and analyse R language code. The compiler takes the R code as the input and goes through various stages of compilations. The Intermediate code is generated and then optimised using this compiler which is then fed as the input to the Target code generator. The compiler finally generates the R code in RISC architecture on screen. The main aim is to correctly handle all the intermediate stages while not losing any data. The compiler handles all arithmetic operations at varying stages of complexity and generates the output. We use the tools lex and yacc to build the compiler as it works on top of C language.

Here are some sample inputs and outputs that show the basic idea:
Input is given in R code and output is generated accordingly

1. Input : `a <- 10*20`

Output : `MOV R1, #200`
`ST a, R1`

2. Input :

`a<-100`

`b<-20`

`c<-a/b`

`c->d`

Output: `MOV R1, #100`

`ST a, R1`

`MOV R2, #20`

`ST b, R2`

`MOV R3, #5`

`ST c, R3`

`ST d, R3`

Architecture of the Language

We mainly handle the following points in this compiler for R language.

1. Arithmetic operations.
2. Conditional operations within constructs.
3. Error Handling for all such operations as present in R.
4. The if, if else and ifelse function in terms of syntax acceptance.
5. All the basic semantics present in R language in regards to scope and assignment of values to variables.
6. Optimisation of code before giving it as an input to be converted to machine language.

The R language is used mainly for operations in Data Analytics and is an interpreted language that mainly operates on command line interpreter. This means that the compiler executes the program directly translating each statement to one or more subroutines and then to another language which is usually machine code. This means that a class of programming errors related to type checking is eliminated in this language. This is both an advantage and a disadvantage.

We can also write the full-fledged R code under the filename extension “.R” . To run R code, we must have R package previously installed. To understand the working of R better, this compiler takes the main functionalities of R and attempts to decode it while we go through its various stages.

In this project we have attempted to replicate more or less all the things mentioned above and run code not through command lines but through a series of R code written in a file. We send this file as the input to the compiler and it does all the above-mentioned processes in various stages of its execution.

Literature survey (Papers and links)

The following links were referred to while building this compiler.

1. For understanding how to write Grammar :

<http://marvin.cs.uidaho.edu/Teaching/CS445/c-Grammar.pdf>

2. For understanding how to build a symbol table :

<https://www.geeksforgeeks.org/symbol-table-compiler/>

3. For understanding basic R : <https://www.r-project.org/>

4. For AST: <https://ruslanspivak.com/lsbasi-part7/>

5. For ICG: <https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/>

The Context Free Grammar

CFG:

Problem -> HEADER | S

HEADER -> library(STR)\nIMPORT | library("STR")\nIMPORT | S | λ

S -> A \nS | Exp \nS | AssignExp1 \nS | AssignExp2 \nS | print(P) \nS | paste0(B)\nS | break\nS | D \nS | λ

P -> STR | ID | RANGE

RANGE -> NUM:NUM | V | seq(NUM , NUM EXTRA)

EXTRA -> , by=NUM | , length.out=NUM | λ

B -> "STR",B | ID , B | "STR" | ID

A -> if (Exp){S\n } | if (Exp){S\n } C

C -> else if(Exp){S\n } C | else if(Exp){S\n } | else{S\n } | λ

D -> ifelse(Exp , "STR", "STR")

Exp -> Exp RELOP Exp | Exp LOGOP Exp | !Exp | NUM | ID | E

RELOP -> < | > | == | <= | >= | !=

LOGOP -> L | | LOGOP | L | LOGOP | L

L -> N&& L | N | N & L | N

N -> !O | O

O -> (Exp) | ID | BOOL | NUM

AssignExp1	-> VAR ASSIGNOP1 Exp VAR ASSIGNOP1 list(V)
AssignExp2	-> Exp ASSIGNOP2 VAR list(V) ASSIGNOP2 VAR
ASSIGNOP1	-> = <- <<-
ASSIGNOP2	-> = -> ->>

E	-> E + T E - T T (E)
T	-> T * F T / F F
F	-> F %% G F %/% G G
G	-> ID NUM ID[NUM] V
V	-> c(vectornum)
Vectornum	-> ITEM, vectornum ITEM
ITEM	-> NUM BOOL B
VAR	-> ID ID[NUM]
ID	-> [A-Z a-z]H .[A-Z_.]H_
H	-> [A-Z a-z _ 0-9.]H λ
BOOL	-> TRUE FALSE
STR	-> [a-z A-Z 0-9.!\$] STR λ
NUM	-> +DIGIT -DIGIT DIGIT
DIGIT	-> [0-9]DIGIT [0-9]

Design Strategy

Symbol Table:

The symbol table is the data structure that is created to serve the following purposes

1. To store the names of all the variables in the code
2. To implement whether all the assignments in the code are done correctly
3. To determine the scope of the variable used within the code
4. Used to store line number for better handling of errors.

The symbol table is implemented keeping the above requirements in mind using a **linked list data structure**. The idea is to push all the variables when they are encountered and their values before or after analysing the arithmetic operations.

In the initial stages we keep track of only direct assignment expressions and leave handling the operations for later stages of the compiler since this stage does not handle operations. We give the scope for each variable starting with one. If the variable is used with a different value again within a loop or a construct the scope is increased by one. This helps us to keep track of the correct value stored in a particular variable if it is re-used in different sections of the code

Here is an example to understand what the design must be.

Input

```
5 b=10
6 print(a)
7 if(a)
8 {
9 b=30
10 print(a)
11 }
```

Output

SYMBOL TABLE							
	LINE		ID		VALUE		SCOPE
	8		b		30		2
	4		b		10		1

In the above example we see the different values of scope for b as it is used in different levels of code.

As mentioned above the linked list data structure is used. The idea is to write a separate code for this using C language and then use its functions whenever we need to update the symbol table.

Abstract Syntax Tree:

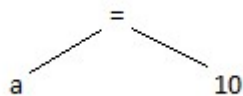
The abstract syntax tree is designed so that the tree is printed when we encounter the assignment part or the operations part of the code. The design strategy **used is a binary tree** with each node being the different variables or numerical values.

The following example is used to highlight the design of the AST used:

Input:

a <- 10

Output:



In the above example the left and right elements of the R assignment expression <- act as the left and the right nodes of the tree.

We use this binary tree that is continuously updated as we encounter the assignment operations.

Intermediate Code Generation

The design strategy to generate intermediate code is to convert all the arithmetic and logical and conditional operations to three address code. This is **done using a stack**. We push elements to a stack and replace the variables with temporaries t0, t1 and so on whenever the code is not in three address format. Otherwise we print the code as it is. Arithmetic expressions are not yet evaluated fully in this stage. The evaluation of arithmetic expression along is partial.

The following example shows the expected input and output.

Input

```
1 a=7+2*4-1*3
2 b=a+2
3 5->c
```

Output

```
t0 = 2 * 4
t1 = 7 + t0
t2 = 1 * 3
t3 = t1 - t2
a = t3
t4 = a + 2
b = t4
c = 5
```

In the above example we see that whenever the code isn't in three address format it is converted using stack to 3AC and when it is already in 3AC it is printed without any changes

Code Optimisation

The two main strategies applied here are

- 1.Constant folding
2. constant propagation

Constant folding is evaluation of all expressions. We do all the arithmetic operations that were pending here in this stage. The idea is that finally only the value and the variable must remain. Now the method to implement this would be to evaluate the expression in an else statement **before pushing it to the stack**. So every time we have only the evaluated value and the variable inside the stack.

The idea in Constant propagation is to use the previous value of the variable if it is already present in the code. For example is “a” is assigned with a value 10 and we write `b<- a` in the input then the code generated in the output must be `b = 10`. Here we “propagate” the value of the variables throughout the code

The example below shows the code before and after optimisation

Before

```
t0 = 2 * 4
t1 = 7 + t0
t2 = 1 * 3
t3 = t1 - t2
a = t3
t4 = a + 2
b = t4
c = 5
```

After

```
a = 12
b = 14
c = 5
```

In the examples we also see that we eliminate all the temporary variables assigned, in this stage of the compiler.

Target code generation:

The Target code is generated by using a python library to scan through the output obtained after the optimization phase of the compiler. The “re” library and several of its functions are used in this stage. The idea is to scan through the input and generate the output in machine language. We have implemented RISC architecture. The

python file scans through the list of inputs and generates the code in machine language.

For the above explained stages now we will go through the tools and Data structures used

Implementation

Symbol table:

The main data structure used is **linked list**. Whenever we encounter a variable assigned to a value we push it to the linked list.

The structure looks like

```
struct symtable
{
    int line;
    char name[30];
    char value[50];
    int scope;
    struct symtable* next;
};
```

The following functions are used:

1. `int checkTable()` : To check whether the variable and its appropriate value are already present in the Symbol table. We pass all the elements of the structure above as parameters along with the head pointer.
2. `Node* pushIntoTable()` : This is to make new entries to the symbol table and called whenever assignments are made within the input code. We use this functions in the assignment stage within the yacc file along with the grammar. The parameters to this function is a head

pointer along with all the elements of the structure written above.

3. Void printTable(): This is used to print the Symbol table. A head pointer is passed as a parameter to begin printing the table. If it points to nothing then the function return nothing.

This covers the implementation of the Symbol table. The functions are called at various stages inside the yacc file. One such example is shown below.

```
T:T MUL F|T DIV F|F;  
F:r{head=pushIntoTable(head,line,(char*)ident,(char*)val,scope);strcpy(val,"-");strcpy(ident,"-");}  
|OB E CB;
```

Abstract Syntax Tree:

The abstract syntax tree is basically a **binary tree**.

The internal structure is as follows.

struct BTree

```
{  
    struct BTree* left;  
    struct BTree* cur;  
    int what;  
    union  
    {  
        int op;  
        char *str;  
    }value;  
    struct BTree* right;  
};
```

We can explain this using an example. If the input is $a = 10$ then the binary tree generated will have “a” as the left node and “10” as the right node. The root will be the assignment operator “=”.

The abstract syntax tree is printed in preorder format during execution and is generated by calling the following functions at various stages in the yacc file.

1. void printAST() : This function is to print the final generated syntax tree. The parameters to this function are the root pointer and the elements of the structure shown above. This goes through a list of operators(+,=,-,*,/ and so on) and selects the appropriate root to be printed.
2. void createTree() : This function is for creating the syntax tree. The parameter is the root pointer and the elements of the structure
3. void preorder() : This function is used to print the tree in preorder format and display it on the terminal. Only the root pointer is the parameter.
4. void freeTree(): After generating the syntax tree we have to free all the left and right node pointers. This function is used to do so.

This covers the implementation of the Abstract Syntax Tree. The functions are called at various stages inside the yacc file.

Intermediate code generation

The main structure that we use here is stack. The stack input and output helps us to evaluate expressions in the right order and precedence. The basic variable used along with the stack is shown below.

```
--  
23 char stack[STACKSIZE][15];  
24 int temp_top = -1;  
25 char temp_var[2];  
26 char temp_count[2]="0";  
27 int label[20],label_num=0,ltop=-1;
```

There are some functions for various stages of the code generation. Basic stack functions like pushtoStack() and popfromStack() are used.

The following two play an important role other than those mentioned above

1. `codegen()`: This function is used to pop elements from the stack after verifying that the line of code that is given as input is already in the form of 3AC.

2. `codegen-assign()`: This function is used to pop elements from the stack when after verifying that the temporary variable is assigned to it appropriately after expression evaluation

We use these functions at different points in the yacc file. One such implementation is shown below.

```
pushToStack("=");  
codegen_assign();
```

Code Optimisation:

The implementation of code optimisation is done by using the stack operations wisely at different points in the code.

The Following two strategies are used

1. Constant folding
2. Constant propagation

The idea for code optimisation is clearly described in the Design Strategy part of the report above.

Implementation of the optimisation stage includes manipulation of stack at various stages of the grammar. We can push to the stack once all evaluation is done instead of directly pushing to stack as soon as we encounter the variable. One such example is shown below.

```

char val[30];
//pushToStack($1.value.str); // comment this for optimisation
strcpy(val,getval(head,$1.value.str,scope));
if(strcmp(val,"-1")==0)
{
    yyerror("No value assigned to variable");
}
else
{
    strcpy($$.value.str,val);
    pushToStack($1.value.str); //remove comment for optimisation
}

```

The above piece of code is already optimised as can be seen from the explanations in the comments.

Target code Generation

The main implementation method for generating the code in machine language is to use regex in python. Scanning through the input files lines if we encounter the different symbols or the different branch conditions then we generate the appropriate code in that area.

These are some of the functions used in it.

1. build() : To generate the registers R0,R1, R2 and so on.
2. verify() : This is for a particular case while scanning through the code. It checks whether the left and the right variables/numerics have registers previously or not. If not present then it calls the build() function to generate a new register for each.
3. movld() : This function is used to generate the MOV or LD command as and when required. It is called when needed.
4. store() : To generate the ST command in the final code generation.

Error Handling

In the Symbol table generation stage, we essentially run the entire code even though we found any syntax error at any stage. The code points to the line number of the error clearly and the user can see what must be changed based on this line number.

We also generate an output file with line numbers and comments and whitespaces removed to further benefit debugging

As we go to further stages of the compiler, we must implement the R language more and more clearly. To this effect we handle errors just like R does. We stop updating the symbol table at the point where we encounter the error and stop executing the code. Since R is an interpreter language this is how errors are handled in R. Also, there will not be any type errors since R handles type within the language itself dynamically. This means that a variable can store a numerical value at one point and can have a string value next.

All numerical errors are handled. For example, division by zero error. This is a commonly encountered error independent of the type of the variables and only depends on the numerical values.

Input:

```
4 f<-19+28*6/3*24
5 f->>g
6 v=10/0
```

Output:

```
f = 1363
g = 1363Divided by zero error at line : 5
SYMBOL TABLE
```


There is also one warning in the case of empty variables
For example, if we have a single line of code like so at line 11

Input:

a

Output:

```
No value assigned to variable at line : 11  
SYMBOL TABLE
```

Then the above warning is returned and there is no update made in the symbol table for this error.

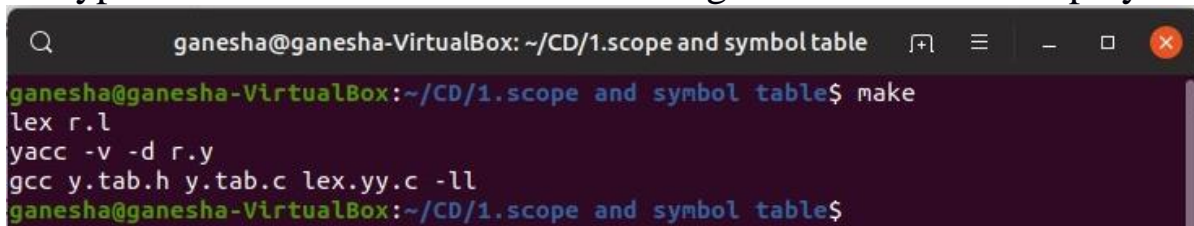
These are the errors handled in the compiler.

Instructions on Running the Compiler:

There are many stages in the compiler but the basic instruction remains the same for all stages except target code generation.

Follow these instructions for all stages:

1. Open the terminal inside the folder of the stage that is to be built.
2. Type make and hit enter. The following result should be displayed.

A screenshot of a terminal window titled 'ganesha@ganesha-VirtualBox: ~/CD/1.scope and symbol table'. The terminal shows the command 'make' being executed, which results in the following output: 'lex r.l', 'yacc -v -d r.y', and 'gcc y.tab.h y.tab.c lex.yy.c -ll'. The prompt returns to 'ganesha@ganesha-VirtualBox:~/CD/1.scope and symbol table\$'.

3. Then type the command “./a.out “filename” ”. Write the name of the file in place of filename without the quotes and hit enter. The output of the following stage will be displayed. Below is the putput for the above stage.

```
ganesha@ganesha-VirtualBox: ~/CD/1.scope and symbol table
ganesha@ganesha-VirtualBox:~/CD/1.scope and symbol table$ make
lex r.l
yacc -v -d r.y
gcc y.tab.h y.tab.c lex.yy.c -ll
ganesha@ganesha-VirtualBox:~/CD/1.scope and symbol table$ ./a.out in
Success
SYMBOL TABLE
      LINE      ID      VALUE      SCOPE
|      8      |      b      |      30      |      2      |
|      4      |      b      |      10      |      1      |
|      2      |      a      |      11      |      1      |
ganesha@ganesha-VirtualBox:~/CD/1.scope and symbol table$
```

Results:

We have obtained a robust and simple R compiler that can be used for understanding the workings of constructs as mentioned above and can handle varying degrees of complex arithmetic operations.

One notable fact is that the compiler can handle nested if else operations with statements within it. It can generate the symbol table and the Intermediate code which is in 3AC format while also handling the expressions and the scope.

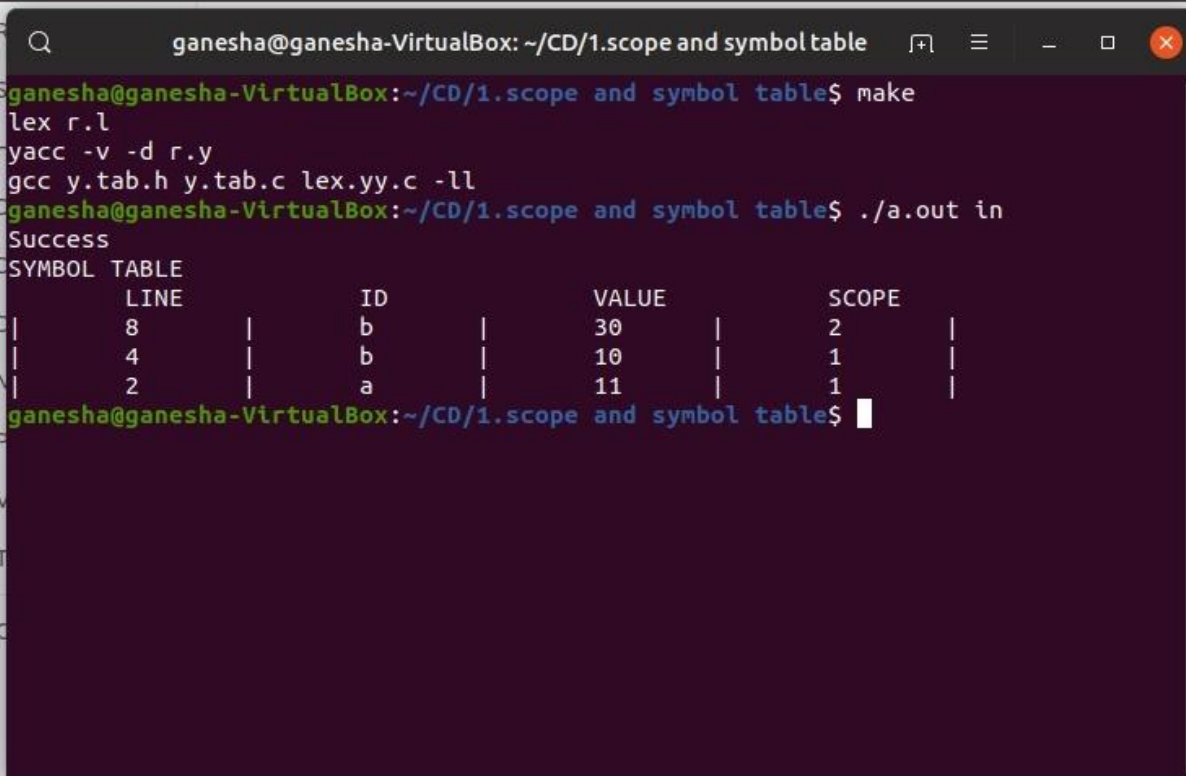
The compiler generated also does all the error handling for the grammar and also finally converts the R code to machine level code in RISC architecture.

Shortcomings:

R code without the necessary braces and just indentation will throw a syntax error. Operations of the arithmetic type executed within the conditional statements are not perfectly handled as the real R compilers. More functions are present in R than that implemented in this compiler. The Target Code generated is specifically RISC architecture and thus comes with all the disadvantages of the same architecture

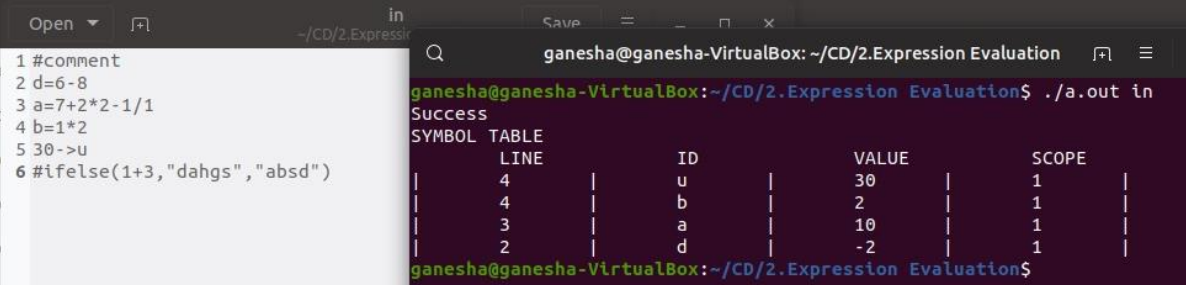
Snapshots:

Symbol table along with scope



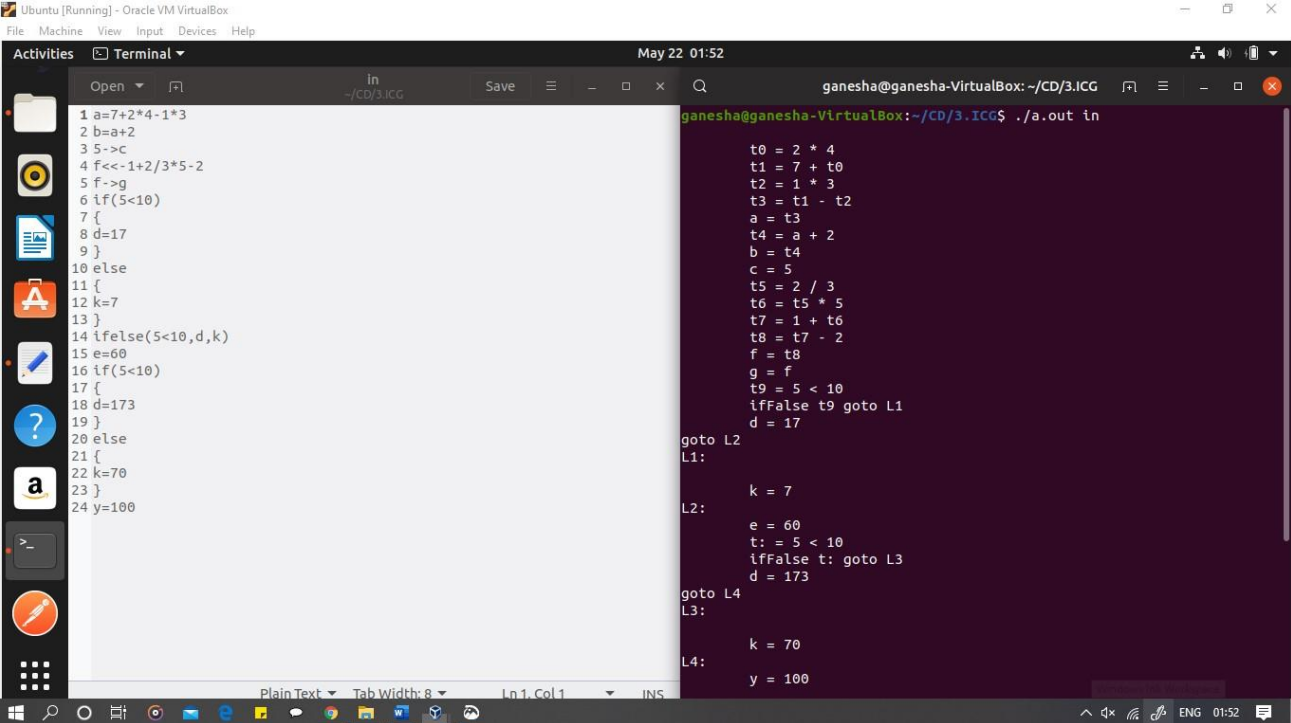
```
ganesha@ganesha-VirtualBox: ~/CD/1.scope and symbol table$ make
lex r.l
yacc -v -d r.y
gcc y.tab.h y.tab.c lex.yy.c -ll
ganesha@ganesha-VirtualBox:~/CD/1.scope and symbol table$ ./a.out in
Success
SYMBOL TABLE
      LINE      ID      VALUE      SCOPE
|-----|-----|-----|
|      8      |      b      |      30      |      2      |
|      4      |      b      |      10      |      1      |
|      2      |      a      |      11      |      1      |
ganesha@ganesha-VirtualBox:~/CD/1.scope and symbol table$
```

Expression Evaluation along with input:



```
ganesha@ganesha-VirtualBox: ~/CD/2.Expression Evaluation$ ./a.out in
Success
SYMBOL TABLE
      LINE      ID      VALUE      SCOPE
|-----|-----|-----|
|      4      |      u      |      30      |      1      |
|      4      |      b      |      2       |      1      |
|      3      |      a      |      10      |      1      |
|      2      |      d      |      -2      |      1      |
ganesha@ganesha-VirtualBox:~/CD/2.Expression Evaluation$
```

Intermediate code Generation



The screenshot displays an Oracle VM VirtualBox window titled "Ubuntu [Running] - Oracle VM VirtualBox". Inside the VM, the Ubuntu desktop environment is visible. A text editor window titled "in" shows a C program with the following code:

```
1 a=7+2*4-1*3
2 b=a+2
3 5->c
4 f<-1+2/3*5-2
5 f->g
6 if(5<10)
7 {
8 d=17
9 }
10 else
11 {
12 k=7
13 }
14 ifelse(5<10,d,k)
15 e=60
16 if(5<10)
17 {
18 d=173
19 }
20 else
21 {
22 k=70
23 }
24 y=100
```

The terminal window, titled "ganesha@ganesha-VirtualBox: ~/CD/3.1CG", shows the execution of the program. The command `./a.out in` has been run, resulting in the following assembly code output:

```
t0 = 2 * 4
t1 = 7 + t0
t2 = 1 * 3
t3 = t1 - t2
a = t3
t4 = a + 2
b = t4
c = 5
t5 = 2 / 3
t6 = t5 * 5
t7 = 1 + t6
t8 = t7 - 2
f = t8
g = f
t9 = 5 < 10
ifFalse t9 goto L1
d = 17
goto L2
L1:
k = 7
L2:
e = 60
t: = 5 < 10
ifFalse t: goto L3
d = 173
goto L4
L3:
k = 70
L4:
y = 100
```

Code Optimisation

Ubuntu [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

May 22 01:55

ganeshha@ganeshha-VirtualBox: ~/CD/4.Optimised Code

ganeshha@ganeshha-VirtualBox: ~/CD/4.Optimised Code\$./a.out in

```
a = 12
b = 14
c = 5
f = 1363
g = 1363
h = 1363
t0 = 5 > 10
ifFalse t0 goto L1
d = 17
goto L2
L1:
    k = 7
L2:
    e = 60
```

Success

SYMBOL	LINE	ID	VALUE	SCOPE
	17	e	60	1
	13	k	7	1
	9	d	17	1
	6	h	1363	1
	4	g	1363	1
	4	f	1363	1
	2	c	5	1
	2	b	14	1
	1	a	12	1

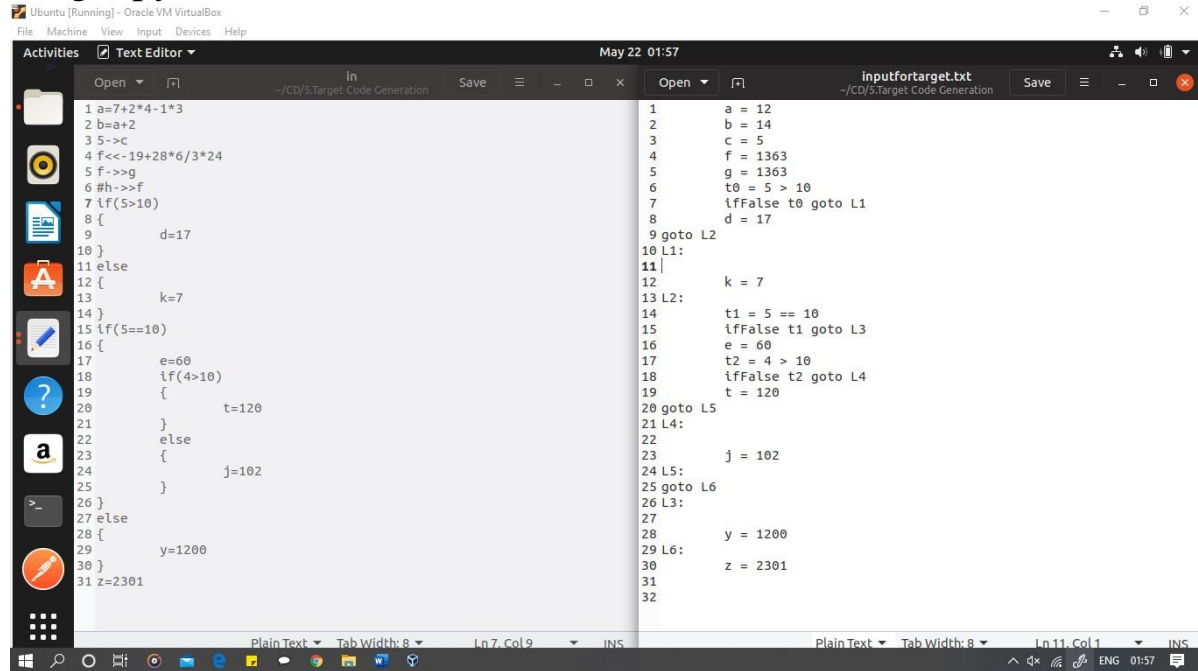
ganeshha@ganeshha-VirtualBox: ~/CD/4.Optimised Code\$

Plain Text Tab Width: 8 Ln 6, Col 5 INS

ENG 01:55

Target Code Generation

The R input file and the optimised code which is also the input to Target.py:



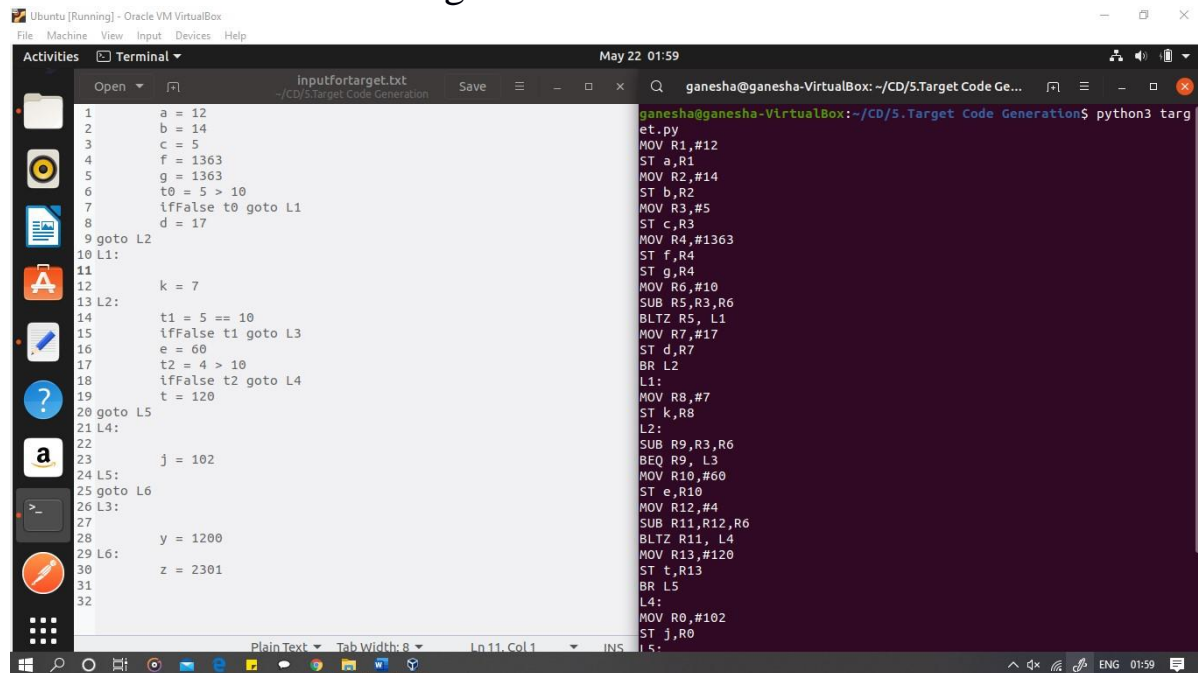
The screenshot shows a text editor window with two tabs. The left tab, named 'in', contains R code. The right tab, named 'inputfortarget.txt', contains the target code. The R code is as follows:

```
1 a=7+2*4-1*3
2 b=a+2
3 5->c
4 f<-19+28*6/3*24
5 f->g
6 #h->>f
7 if(5>10)
8 {
9     d=17
10 }
11 else
12 {
13     k=7
14 }
15 if(5==10)
16 {
17     e=60
18     if(4>10)
19     {
20         t=120
21     }
22     else
23     {
24         j=102
25     }
26 }
27 else
28 {
29     y=1200
30 }
31 z=2301
```

The target code is as follows:

```
1 a = 12
2 b = 14
3 c = 5
4 f = 1363
5 g = 1363
6 t0 = 5 > 10
7 ifFalse t0 goto L1
8 d = 17
9 goto L2
10 L1:
11 |
12 k = 7
13 L2:
14 t1 = 5 == 10
15 ifFalse t1 goto L3
16 e = 60
17 t2 = 4 > 10
18 ifFalse t2 goto L4
19 t = 120
20 goto L5
21 L4:
22 |
23 j = 102
24 L5:
25 goto L6
26 L3:
27 |
28 y = 1200
29 L6:
30 z = 2301
31
32
```

The RISC instruction set generated:



The screenshot shows a terminal window with the RISC instruction set generated from the target code. The instruction set is as follows:

```
MOV R1,#12
ST a,R1
MOV R2,#14
ST b,R2
MOV R3,#5
ST c,R3
MOV R4,#1363
ST f,R4
ST g,R4
MOV R6,#10
SUB R5,R3,R6
BLTZ R5, L1
MOV R7,#17
ST d,R7
BR L2
L1:
MOV R8,#7
ST k,R8
L2:
SUB R9,R3,R6
BEQ R9, L3
MOV R10,#60
ST e,R10
MOV R12,#4
SUB R11,R12,R6
BLTZ R11, L4
MOV R13,#120
ST t,R13
BR L5
L4:
MOV R0,#102
ST j,R0
L5:
```

Conclusions

By building this compiler practical insight into how the various stages of a compiler works was attained. May it be the error handling stage or the target code generation making compilers for any language requires discipline and order and clear understanding of the various stages of the compiler. This is a good exercise into the learning multiple languages and implementations thoroughly while also offering many interesting problems along the way

Further Enhancements

The following enhancements can be done while building this compiler again

1. Handling multidimensional vectors
2. Handling strings
3. Refining the error handling more so that it resembles that of an actual R compiler.
4. Building a more refined symbol table and a syntax tree with many additional features
5. Handling various looping constructs like the for or while equivalents in R language.

Bibliography

The following links were referred to while building this compiler.

1. For understanding how to write Grammar : <http://marvin.cs.uidaho.edu/Teaching/CS445/c-Grammar.pdf>
2. For understanding how to build a symbol table : <https://www.geeksforgeeks.org/symbol-table-compiler/>
3. For understanding basic R : <https://www.r-project.org/>
4. For AST: <https://ruslanspivak.com/lbasi-part7/>
5. For ICG: <https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/>