Report on

"Mini Compiler for R in C with Constructs: FOR, WHILE, IF and IF-ELSE statements"

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Kirthika Gurumurthy	**PES1201700230**
Richa	**PES1201700688**
Akanksha	**PES1201701799**

Under the guidance of

Prof. Kiran P

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

INTRODUCTION

This mini compiler has been built for the language R in C and handles the constructs: for, while, if and if-else.

Sample input :

```
# b = "hello"
# c <- 2.3
#if (a == 2) a = 3 else a = 4
# # egd <- 567
# print(a)
# a <- 3.5
#for (i in 1:4) {

#     print(i)

#}
#v = a:b:c
b = 2
c = 1
#d = 9
a = b*c+d
e = d-b*c
#d = a+d/b*2
#m = "okay"
if(a < 5) {
for (i in 2:3)
{
print("mom")
}
print(a)
a=5;}
else {
while(i>3){
a=5
}
print("end of program")
}
#b <- 3

#for (i in 2:3)
#{
#print(a)
```

Sample output :

Symbol table :

```
./a.out < test.1
--------------------------------This program is valid--------------------------------

---------------------------------SYMBOL TABLE-----------------------------------------
=============================================================================================|
Symbol                    Value          Data Type                 Line number
=============================================================================================|
b                         2              num                       15
 --------------------------------------------------------------------------------------------|
c                         1              num                       16
 --------------------------------------------------------------------------------------------|
a                         5              num                       18
 --------------------------------------------------------------------------------------------|
d                                                                   18
 --------------------------------------------------------------------------------------------|
e                         d              num                       19
 --------------------------------------------------------------------------------------------|
i                                                                   23
 --------------------------------------------------------------------------------------------|
```

AST :

```
TYPE: SEQ              DATA:
TYPE: SEQ              DATA:
TYPE: SEQ              DATA:
TYPE: SEQ              DATA:
TYPE: SEQ              DATA:
TYPE: =                DATA:
TYPE: symbol           DATA: b              ADDRESS:0x6081c0
TYPE: number_constant DATA: 2
TYPE: =                DATA:
TYPE: symbol           DATA: c              ADDRESS:0x608388
TYPE: number_constant DATA: 1
TYPE: =                DATA:
TYPE: symbol           DATA: a              ADDRESS:0x608550
TYPE: +                DATA:
TYPE: /                DATA:
TYPE: symbol           DATA: b              ADDRESS:0x6081c0
TYPE: symbol           DATA: c              ADDRESS:0x608388
TYPE: symbol           DATA: d              ADDRESS:0x608718
TYPE: =                DATA:
TYPE: symbol           DATA: e              ADDRESS:0x6088e0
TYPE: -                DATA:
TYPE: symbol           DATA: d              ADDRESS:0x608718
TYPE: /                DATA:
TYPE: symbol           DATA: b              ADDRESS:0x6081c0
TYPE: symbol           DATA: c              ADDRESS:0x608388
TYPE: IF               DATA:
TYPE: <                DATA:
TYPE: symbol           DATA: a              ADDRESS:0x608550
TYPE: number_constant DATA: 5
TYPE: SEQ              DATA:
TYPE: SEQ              DATA:
TYPE: SEQ              DATA:
TYPE: for              DATA:
TYPE: for_condition    DATA:
TYPE: symbol           DATA: i              ADDRESS:0x608aa8
TYPE: :                DATA:
TYPE: number_constant DATA: 2
TYPE: number_constant DATA: 3
TYPE: print            DATA:
```

```
TYPE: symbol          DATA: a               ADDRESS:0x608550
TYPE: =               DATA:
TYPE: symbol          DATA: a               ADDRESS:0x608550
TYPE: number_constant DATA: 5
TYPE: SEQ             DATA:
TYPE: SEQ             DATA:
TYPE: =               DATA:
TYPE: symbol          DATA: a               ADDRESS:0x608550
TYPE: number_constant DATA: 5
TYPE: while           DATA:
TYPE: >               DATA:
TYPE: symbol          DATA: i               ADDRESS:0x608aa8
TYPE: number_constant DATA: 3
TYPE: SEQ             DATA:
TYPE: =               DATA:
TYPE: symbol          DATA: a               ADDRESS:0x608550
TYPE: number_constant DATA: 5
TYPE: print           DATA:
TYPE: string_constant DATA: "end of program"
```

Intermediate Code Generation:

```
#################### ICG ####################
b = 2
c = 1
T6 = T4 + d
T4 = b * c
a = T6
T11 = d - T10
T10 = b * c
e = T11
T14 = a < 5
IF FALSE T14 GOTO L0
L1:
T15 = T16 && T17
T16 = i >= 2
T17 = i <= 3
IF FALSE T15 GOTO L2
print "mom"
GOTO L1
L2:
print a
a = 5
L0:
L3:
T27 = i > 3
IF FALSE T27 GOTO L4
a = 5
GOTO L3
L4:
print "end"
```

Code optimisation:

```
######## Constant Propagation and Constant Folding done 5 times ########
b = 2
c = 1
T6 = 2 + d
a = T6
T11 = d - 2
e = T11
T14 = a < 5
IF FALSE T14 GOTO L0
L1:
T15 = T16 && T17
T16 = i >= 2
T17 = i <= 3
IF FALSE T15 GOTO L2
print "mom"
GOTO L1
L2:
print a
a = 5
L0:
L3:
T27 = i > 3
IF FALSE T27 GOTO L4
a = 5
GOTO L3
L4:
print "end"
```

```
######## Optimised ICG after Dead Code Elimination ########
T6 = 2 + d
a = T6
T11 = d - 2
T14 = a < 5
IF FALSE T14 GOTO L0
L1:
T15 = T16 && T17
T16 = i >= 2
T17 = i <= 3
IF FALSE T15 GOTO L2
print "mom"
GOTO L1
L2:
print a
a = 5
L0:
L3:
T27 = i > 3
IF FALSE T27 GOTO L4
a = 5
GOTO L3
L4:
print "end"
```

Target code generation:

```
###### Assembly Code Generated ######
LD R0 , d
ADD R1 , #2 , R0
ST i , R4
MOV R4 , R1
SUB R2 , R0 , #2
LD R3 , a
SUB R4 , R3 , #5
BGEZ R4 , L0

L1:
LD R4 , i
SUB R5 , R4 , #2
BLZ R5 , L2
SUB R5 , R4 , #3
BGZ R5 , L2
print "mom"
BR L1

L2:
print a
MOV R4 , #5

L0:

L3:
SUB R5 , R4 , #3
BLEZ R5 , L4
MOV R4 , #5
BR L3

L4:
print "end"
ST T6 , R1
ST a , R4
ST d , R0
ST T11 , R2
```

ARCHITECTURE OF THE LANGUAGE

There is a sequence of phases involved in the compilation process, each phase takes the input from the previous phase, represents the program in its own way and feeds the output to the next phase. The phases implemented include - lexical analysis, syntax analysis, semantic analysis, intermediate code generation, intermediate code optimization, and target code generation. We have tried to incorporate as much of the basic syntax and functionality of R as we could. In terms of syntax, the following has been taken care of :

- for constructs
- while constructs
- if and if-else statements
- print statements
- newline
- expressions or assignment (can be “=” or “<-”) including multiple assignments (ex: a <- b <- 1)
- relational operators (“>”,”<”,”!=” etc)
- arithmetic operators(“+”,”-” etc)
- semicolon (R by default just needs a newline, a semicolon is not necessary but we have taken care of this as well)
- print statements
- numbers
- strings
- single-line comments

In terms of semantics we have ensured the following:

- Test expressions used in if, while, and for constructs evaluate to a boolean value.
- Variables used in the RHS have been defined previously.
- Uninitialized variables are added to the symbol table.
- Expressions are not on the left side of an assignment statement.
- Misunderstanding of operation precedence does not occur.

LITERATURE SURVEY

1. https://norasandler.com/2017/11/29/Write-a-Compiler.html
2. https://stackoverflow.com/
3. https://cran.r-project.org/doc/manuals/r-devel/R-lang.html
4. https://ruslanspivak.com/lsbasi-part7/
5. https://www.gatevidyalay.com/code-optimization-techniques/
6. https://www.javatpoint.com/code-generation
7. https://web.cs.ucdavis.edu/~pandey/Teaching/ECS142/Lects/final.codegen.pdf

CONTEXT FREE GRAMMAR

Data types:

<identifier> → < “.” ><letter><identifier> | (<letter> | <digit>| “_” | “.”) <identifier>) | λ
<numeric> → <integer> | <double>
<list> → <string_list> | <double_list> | <integer_list>
<integer> → (“+” | “-”) <digit> ‘L’
<double> → (“+” | “-”) <digit> “.” <digit> | (“+” | “-”) <digit> “.”
<string> → “ ‘ “ ([^\\] | <escapesequence>) <string> “ ‘ “ | ‘ “ ‘ ([^\\] | <escapesequence>) <string>) ‘ “ ‘ | λ
<escapesequence> → [\\](.)
<string_list> → “c(” <string_item> “)”
<string_item> → <string> | <string_item> “,” <string_item> | <double_item> | <list>
<integer_list> → “c(” <integer_item> “)”
<integer_item> → <integer> | <integer_item> “,” <integer_item>
<double_list> → “c(” <double_item> “)”
<double_item> → <double> | <double_item> “,” <double_item> | <integer_item>
<letter> → <lower> | <upper>
<lower> → “a” | “b” | “c” | “d” |.....| “y” | “z”
<upper> → “A” | “B” | “C” | “D” |.....| “Y” | “Z”
<digit> → [0-9] <digit> | [0-9]

Constructs:

<constructs> → <for_loop> | <while_loop> | <if_statement>
<for_loop> → “for(“<identifier> “in” <iterable> “) {“ <statements> “}”
<while_loop> → “while” <condition> “{” <statements> “}”
<if_statement> → “if” <condition> “{“ <statements> “}”

<iterable> → <range> | <list>
<range> → <integer> “:” <integer> | “seq(“<integer> “,” <integer> ”)” | “seq(“<integer> “,” <integer> “,” <numeric> ”)”
<condition> → <and_cond> | <or_cond>
<statements> → <assign_expr> <statements> | <print_stat> <statements> | <arithmetic_expr> <statements> | λ

<comp_ops> → "<" | ">" | "==" | ">=" | "<=" | "!="
<u_op> → “+=” | “-=” | “*=” | “*=” | “/=”
<and_cond> → <expr> <comp_ops> <expr> | (“&&” (<and_cond>|<or_cond>))
<or_cond> → <expr> <comp_ops> <expr> | (“||” (<and_cond>|<or_cond>))
<expr> → <identifier> | <character> | <numeric> |<list>

<assign_ops> → “=” | “<-”
<assign_expr> → <identifier> (<assign_ops> | <u_op>) <expr>
<print_stat> → “print(” <identifier> “)”
<import_stmt> → “import ::from(“ <library> “,” <func> “)” | <func> “<-” <library> “::” <func>
<arithmetic_ops1> → “+” | “-”
<arithmetic_ops2> → “*” | “/”
<arithmetic_expr> → <id2> | <id1> <arithmetic_ops1> <id2>
<id2> → <id3> | <id2> <arithmetic_ops2> <id3>
<id3> → <numeric> | <identifier>
<library> → “dplyr” | “ggplot2” | “knitr” | “lubridate” | ...
<func> → <<< function of <library> >>> | <func> “,” <func>

Tokens:

1. Keyword

<keyword, lexeme, line#>

2. Identifier

<id, lexeme, value, type, line#>

3. Operators

<op, lexeme, type = [<comp_ops>, <u_op>, <arithmetic>, <assign_ops>, <arithmetic_ops1>, <arithmetic_ops2>], line#>

4. Punctuators

<punc, lexeme, type = [COLON, ESCAPE CHARACTERS, PARENTHESES, QUOTES], line#>

5. Literal

<literal, lexeme, type, line#>

DESIGN STRATEGY

SYMBOL TABLE :

The symbol table stores the names of all the entities in a structured format. For each name it maintains an entry of the :

- ➢ Symbol: specifies the name of the symbol (identifier).
- ➢ Type: specifies the data type of the variable
- ➢ Value: represents the value of the variable if initialized/defined.
- ➢ Line number: specifies the line number at which the variable is present.

We define a function to lookup (search) the symbol table to check if a name exists. If the name does not exist then an installID function is called which inserts the corresponding entry (with the Symbol, Type, Value, Line number) to the symbol table. If the symbol already exists, a modifyID function modifies the corresponding entry and updates the table.

The symbol table is used in various phases of the compiler - in lexical analysis, new table entries are created, in the syntax analysis phase information is added (like type, line number, etc), the semantic analysis phase uses the information in the symbol table to check for semantics, each ID node in the AST is linked to the corresponding symbol table entry, the code optimization phase uses information present in the symbol table for machine-dependent optimization.

ABSTRACT SYNTAX TREE:

The AST is a tree representation of the abstract syntactic structure of the source code. As you can see in our snapshots, we display the AST by representing the nodes by their node type,and data (value in case of constant names and symbol names in case of variables).We also print the pointer address of symbols.The AST uses operators/operations as root and interior nodes and operands as its children.

The abstract syntax tree is printed in an inorder way. Semicolons, newlines are donated by node type “SEQ”.The number of children present in each node depends on the construct used. For example, for a while construct, the AST is :

while

condition body

There are different levels of the AST printed one below the other as depicted in the snapshots.

INTERMEDIATE CODE GENERATION:

Intermediate code is used to translate the source code into machine code. We generate a linear sequence of simple statements by recursively stepping through the AST. It is represented as a three address code which has at most one operation in the RHS, temporary variables are introduced as and when required. The while, for, if and if-else control structures are broken down into conditional branch statements. The statements are stored in a buffer and displayed as shown in the ICG snapshot. We have displayed the intermediate code for all types of evaluations and constructs (including constructs within other constructs such as a for within a while). Essentially for assignments

and expressions, every variable declared and the result of the expressions is stored in a temporary variable. Iterative and conditional statements are handled by goto statements and labels. The operators' precedence is specified and the computation is done accordingly.

CODE OPTIMIZATION

The code optimization phase tries to improve the intermediate code by making it consume fewer resources(memory, CPU, etc) so that it results in a faster running machine code. We have performed three kinds of code optimizations - Constant Propagation, Constant Folding, and Dead Code Elimination. For constant propagation, if a variable is known to contain a particular constant value at a particular point in the program, we replace the references to that point with that constant value.
For example, if we take the code :

```
x  <- 10
y  <- 3 + x/5
z  <-  y * (20/x +1 ) - x
```

Here x is assigned a constant and thus can be propagated 3 times. After propagation:

```
x  <- 10
y  <- 3 + 10/5
z  <-  y * (20/10 +1 ) - 10
```

This is faster than looking up and copying the value of the variable and also saves time by eliminating assigning a value to a variable that is itself further used only to propagate that value throughout the code.
In constant folding, we evaluate an expression involving constant values and simplify it to a constant resultant value. This is particularly useful when constant propagation is performed. For example, if we take the code :

```
a <- 4 * 5 * 10
```

We can simply replace the code with :

```
a <- 200
```

In dead code elimination, we have removed all the expressions or statements whose values or effects are unused.
We have defined functions such as isnum, iskey, istemp etc to help carry out these optimizations.

ERROR HANDLING

Every time the parser encounters an error,we display what the error is such as "unexpected symbol" along with the line number. Comments are removed from the code before parsing.

TARGET CODE GENERATION

In our compiler , the design for target code generation uses an algorithm wherein we first take a sequence of three-address statements as input and perform the following steps as described further to obtain assembly instructions.As a part of register allocation , temporaries and registers are used to hold intermediate results.We keep results and data value in registers if they are going to be used again as opposed to a simple load-use-store scheme which can be highly inefficient as instructions involving memory are slower than those involving registers.For this we need to keep track of the

registers which are currently in use and what they hold, where the current value of a variable can be found ,the next use of variables (keep track of if and where the variable is going to be used again),which variables whose values are currently in registers need to be stored in memory before performing next steps such as jumps etc.
How we do this is by keeping track of the next use of the variables by going through one pass of the code.Then on the second pass we start the conversion.If an instruction involves a variable that has a next use we keep it in the register without storing it back.Also to decrease computation time we perform loading only during arithmetic operations and the assignment of one variable to another.In all other cases we simply use the register with the values directly as the value of each variable might change at each point of time.We then look at each IR instruction and find corresponding set of machine instructions (in our case we've used a combination of ARM and MIPS architecture) .We check if the the variables in the instruction are already in a register.If they are we can use them ,if not we load from memory (as described before).We hold the values of variables in the registers as long as possible and store them back into location when running out of registers or jumping to a labelled statement.While jumping to a labelled statement if registers are not available we simply make the register being used to jump as available as it only consists of the label.But if we run out of registers in we check the register which uses a variable which has the least probability of being used next ,store it back and make it available for the new variable. We perform most of the stores at the end except for the cases mentioned.

IMPLEMENTATION DETAILS

SYMBOL TABLE :

For creating the symbol table we define the structure :

```
typedef struct symbol_table
{
        int  ln_no;
        char symbol_name[50];
        char data_type[200];
        char val[200];

} sym_tab;
```

Which as shown above and specified in the design strategy matches the structure of each entry in the symbol table - name, type, value, and line number.

An int lsym_last variable is maintained which is modified as and when the entries to the symbol table are added. These functions are defined in relation to the symbol table :

- ➢ int lookup_symtab(char *symbol) : which checks whether the token name is already present in the symbol table
- ➢ void print_symtab(sym_tab *tab, int c) : which displays the symbol table
- ➢ void installID(char *symbol_name, int ln_no): which inserts a new entry into the symbol table if it doesn't exist.
- ➢ void modify_symtab(char *symbol_name, char *data_type, char *val): which modifies the symbol table when there is a change in the value, type, or line number for a particular entry.
- ➢ sym_tab *fetch_sym(char *symbol): which gets the corresponding symbol table entry for constructing the AST.

ABSTRACT SYNTAX TREE:

For the abstract syntax tree generation the following data structures are used:

```
typedef union val
{
        sym_tab *p;
        char const_str[30];
        int const_num;

} val;
```

This contains the pointer to the particular entry in the symbol table along with the corresponding value of the string or number constant.

```
typedef struct node
{
```

```
        union val value;
        char data_type[30];
        int nodes_n;
        struct node* plist[15];
} node;
```

This contains the type of the node, the value as defined in the union data structure, the number of nodes present, and the subtree of the node as denoted by Node* ptrList[10].
The functions defined are :

- node* createNode(char *data_type, val value, node* *plist, int length): which creates the corresponding node of the AST.
- void printAST(node *m): which displays the subtree from that particular node as the parent.

INTERMEDIATE CODE GENERATION:

For intermediate code generation as specified in the design strategy we use temporary variables to Store the results of assignments and expressions. The ICG statement is stored in a buffer and displayed. The following variables are defined :
int valid: flag to check for the validity of the code
int label_count: used for jumping to labels for handling iterative and conditional statements.
int temp_count : count of the temporary variables used
char buffer[300] : used to display the ICG statements
int step = 0: used to increment or jump to statements.

CODE OPTIMIZATION:

For code optimisation we define the structures :

```
typedef struct list{
   char item[100];
} list_;

typedef struct dict{
   char key[100];
   char value[100];
} dict_;
```

list_ is used to store the list of tokens and a dict structure is defined which stores the name(key) and value of the token.
We define lists :
char arith[] = {'+','-','*','/'}
char *logic[] = {"<",">","<=",">=","==","!="}
char *rel[] = {"&&","||"}
char *keywords[] = {"IF","FALSE","GOTO","print"}

and functions such as isid() isnum,istemp(),islevel(),isnum() etc to check and perform the code optimisation. We have a int eval(int a, int b, char *c) which evaluates the expression , void constant_folding(FILE *fp,FILE *fp1) which performs constant folding , void constant_propagation(FILE *fp,FILE *fp1,char *s) which performs constant propagation and void dead_code_elim(FILE *fp, FILE *fp1) which performs dead code elimination.We loop through the code till no more code can be eliminated in dead code elimination.

ERROR HANDLING

We return the line number and the error when the parser encounters it.

ASSEMBLY CODE GENERATION

To implement the design strategy as explained in the previous section we've used lists to store the arithmetic,relational and logical operators ,as well as the keywords :

```
arith = ['+','-','*','/']
relop = ['<','>','<=','>=','==','!=']
logic = ['&&','||']
keywords = ['IF','FALSE','GOTO','print']
```

We've defined dictionaries for arithmetic and relational operators

```
arith_ass = {'+':"ADD",'-':"SUB",'*':"MUL",'/':"DIV"}
```

Value is the assembly instruction for the symbol

```
relop_ass = {'<':'GEZ','>':'LEZ','<=':'GZ','>=':'LZ','==':'NEZ','!=':'EZ'}
```

If the relational operator to be checked is lesser than ,then we subtract the value in the variable from the value it is compared with and check if it is greater than or equal to zero ,and based on whether this is true or false we jump to the corresponding label.SImilarly this is done for other relational operators as well.

We have an def addLine(list_of_toks) which takes each line separated into tokens as input and returns the assembly code for each line and a def printAssCode(list_of_lines) function which prints the assembly code by combining all the lines.We've also defined a few lambda functions :

- ➢ islevel = lambda s : bool(re.match(r"^L[0-9]*$", s)) - to check if it is a label token
- ➢ istemp = lambda s : bool(re.match(r"^T[0-9]*$", s))- to check if it is a temporary variable
- ➢ isid = lambda s : bool(re.match(r"^[A-Za-z][A-Za-z0-9_]*$", s)) - to check if it is an identifier
- ➢ isnum = lambda s: bool(s.isdigit()) - to check if it is a constant

We have a fixed number of 8 registers that we keep track of in an array , a dictionary with tokens as key and assembly code as value for arithmetic and relational expressions, #dictionary with tokens as key and assembly code as value for arithmetic and relational expressions and a list of labels for generating the assembly code.We generate it using these data structures with the logic :

1) Arithmetic Expression:

BEFORE

```
b = 5 + c
```

AFTER

```
LD R0 , c
ADD R0 , #5 , R0
```

ST b , R0 // this is just an example ,the values are held in the registers as ling as possible as described in the design strategy.
Similarly for SUB, MUL and DIV
2) Logical Expression:
BEFORE
T0 = a < 5
IF FALSE T0 GOTO L0
AFTER
LD R0 , a
SUB R0 , R0 , #5
BGEZ R0 , L0
Similarly for other logical operators
3) Relational Expression:
BEFORE
T17 = i >= 2
T18 = i <= 3
T16 = T17 && T18
IF FALSE T16 GOTO L2
AFTER
LD R0 , i
SUB R0 , R0 , #2
BLZ R0 , L2
LD R0 , i
SUB R0 , R0 , #3
BGZ R0 , L2
Simlarily for or(||) operator
4)Unconditional Break
BEFORE
GOTO L2
AFTER
BR L2
5)Assignment Operation with identifier or immediate value
BEFORE
a = 5
b = a
AFTER
ST a , #5
LD R0 , a
ST b , R0
We have a defined a reg_use dictionary and a var_use dictionary which contain the variables as keys and the registers containing those variables, the next use of the variables as a list as values respectively.We also have a releaseReg function which takes the variable and register as input and

releases the register which contains the variable which has the least probability of next use .Apart from this we also have a reg_avail list which keeps track of the available registers.

We've implemented the assembly code generation in Python.

BUILD AND RUN OUR PROGRAM:

We have specified makefiles for each phase, so to see the output simply go into the corresponding phase directory and run :

- make clean
- make

RESULTS AND SHORTCOMINGS

This mini compiler for R scans the source code as a stream of characters,converts it into meaningful lexemes, parses the grammar according to the R syntax, and finally generates a target code (after performing optimizations on the intermediate code). Possible shortcomings of the compiler are :

- It does not perform error recovery.
- It only takes care of basic R syntax specified in the report.
- It doesn't handle conditions such as dangling else.
- It doesn't handle infinite loops.
- In case of print statements,it can't handle strings with multiple words.

SNAPSHOTS

1. Symbol table creation

Input:

```
# this is a comment
a = 3
c=10
b = "hello"
if (a == 2) { a = 3 }
b <- a <- 5
print(a)

for (i in 1:4) { print(i) }
```

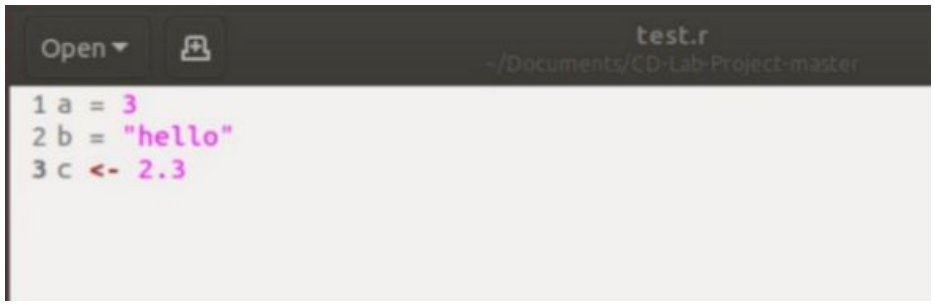
Output:

```
---------------------------------This program is valid---------------------------------

---------------------------------SYMBOL TABLE-----------------------------------------
=============================================================================================|
Symbol                  Value           Data Type               Line number
=============================================================================================|
a                       5               num                     2
 --------------------------------------------------------------------------------------------|
c                       10              num                     3
 --------------------------------------------------------------------------------------------|
b                       a               num                     4
 --------------------------------------------------------------------------------------------|
i                                                               9
 --------------------------------------------------------------------------------------------|
guru@guru-HP-Notebook:~/Desktop/CD Project/Phase3 AST$
```

AST AND ICG

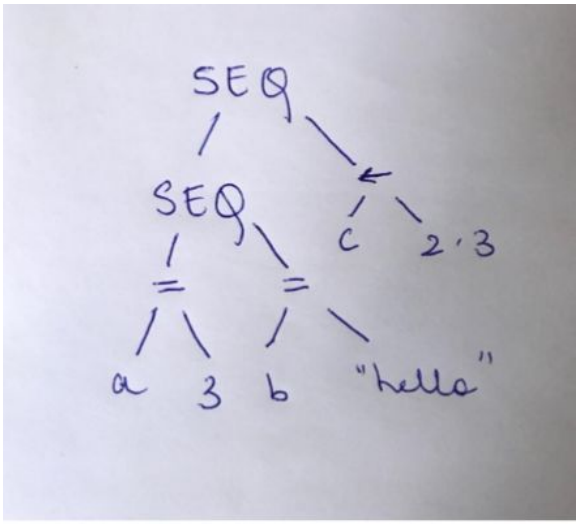
Example 1:



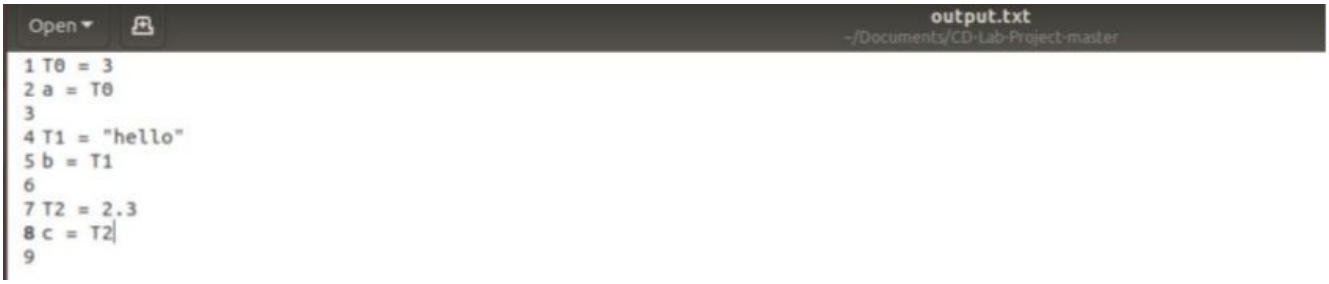
AST :

```
./a.out < t2.r
TYPE: SEQ              DATA:
TYPE: SEQ              DATA:
TYPE: =                DATA:
TYPE: symbol           DATA: a              ADDRESS:0x6081c0
TYPE: number_constant  DATA: 3
TYPE: =                DATA:
TYPE: symbol           DATA: b              ADDRESS:0x608388
TYPE: string_constant  DATA: "hello"
TYPE: <-               DATA:
TYPE: symbol           DATA: c              ADDRESS:0x608550
TYPE: number_constant  DATA: 2
```

Reference diagram to the above output: Inorder traversal



ICG:



Code snippet to generate AST and ICG:

```
equal_assign:
        SYMBOL EQ_ASSIGN expr_or_assign {
                printf("eq_assign: %s %s, SYMBOL: %s\n", $3.type, $3.value, $1.value);
                modifyID($1.value, $3.type, $3.value);

                $1.nodeptr = make_node("SYMBOL", (data) getSymbol($1.value), (NodePtrList) {NULL}, 0);
                $$.nodeptr = make_node("=", (data) 0, (NodePtrList) {$1.nodeptr, $3.nodeptr}, 2);
                sprintf(buffer,"T%d = %s"
                                "\n%s = T%d\n" ,
                                temp_count,$3.value,$1.value,temp_count);
                ++temp_count;
                strcpy($$.value,buffer);
                        }
    ;

        |       SYMBOL LEFT_ASSIGN expr          {
                printf("left_assign: %s %s, SYMBOL: %s\n", $3.type, $3.value, $1.value);
                modifyID($1.value, $3.type, $3.value);
                $1.nodeptr = make_node("SYMBOL", (data) getSymbol($1.value), (NodePtrList) {NULL}, 0);
                $$.nodeptr = make_node("<-", (data) 0, (NodePtrList) {$1.nodeptr, $3.nodeptr}, 2);
                sprintf(buffer,"T%d = %s"
                                "\n%s = T%d\n" ,
                                temp_count,$3.value,$1.value,temp_count);
                ++temp_count;
                strcpy($$.value,buffer);
                        }
```

Example 2:

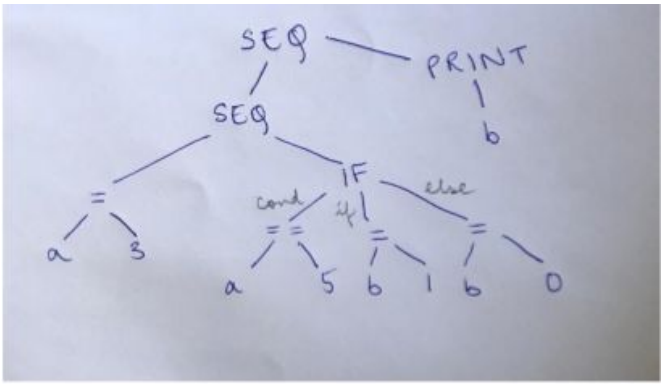
Open ▾ | test.r ~/Documents/CD-Lab-Project-master | Save

```
1 a = 3
2 if ( a == 5 ) b = 1 else b = 0
3 print(b)
4
```

AST:

```
./a.out < t3.r
TYPE: SEQ              DATA:
TYPE: SEQ              DATA:
TYPE: =                DATA:
TYPE: symbol           DATA: a              ADDRESS:0x6081c0
TYPE: number_constant  DATA: 3
TYPE: IF               DATA:
TYPE: ==               DATA:
TYPE: symbol           DATA: a              ADDRESS:0x6081c0
TYPE: number_constant  DATA: 5
TYPE: =                DATA:
TYPE: symbol           DATA: b              ADDRESS:0x608388
TYPE: number_constant  DATA: 1
TYPE: =                DATA:
TYPE: symbol           DATA: b              ADDRESS:0x608388
TYPE: number_constant  DATA: 0
TYPE: print            DATA:
TYPE: symbol           DATA: b              ADDRESS:0x608388
```

Reference diagram to the above output: Inorder traversal

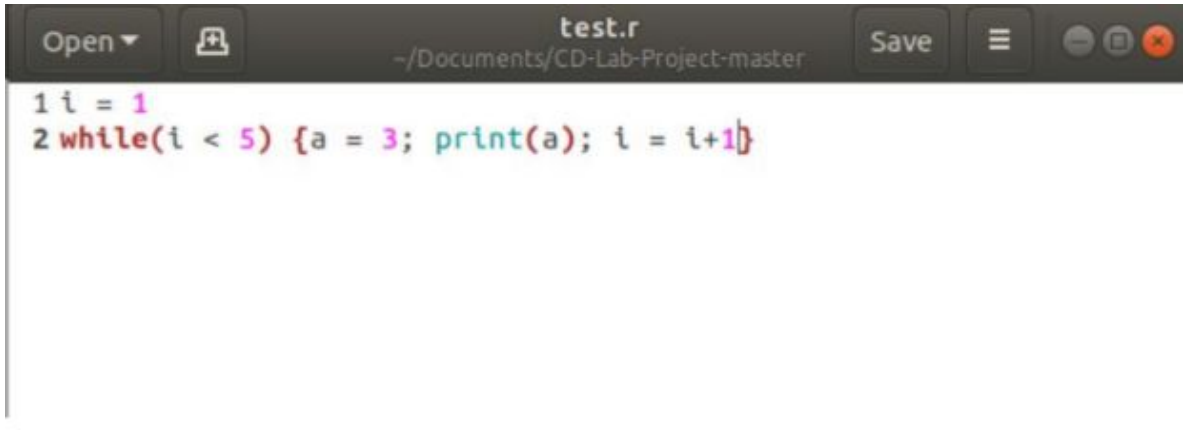


Code snippet to generate AST and ICG :

```
118         |       IF \fcond expr_or_assign ELSE expr_or_assign    {
119                                                                   $$.nodeptr = make_node("IF", (data) 0, (NodePtrList) ($2.nodeptr, $3.nodeptr, $5.nodeptr), 3);
120                                                                   sprintf(buffer,"T%d = %s"
121                                                                                  "IF FALSE T%d GOTO L%d"
122                                                                                  "\n%s "
123                                                                                  "\nGOTO L%d "
124                                                                                  "\nL%d:%s"
125                                                                                  "\nL%d:\n" ,
126                                                                                  temp_count,$2.value,temp_count,label_count,$3.value , (label_count+1) ,
127                                                                                  label_count,$5.value,(label_count+1));
128                                                                   strcpy($$.value,buffer);
129                                                                   ++label_count;
130                                                                   ++label_count;
131                                                                   ++temp_count;
```

```
168 print_statement:
169         PRINT_ expr RIGHT_PAREN                 {
170                                                                   $$.nodeptr = make_node("PRINT", (data) 0, (NodePtrList) ($2.nodeptr), 1);
171                                                                   sprintf(buffer,"T%d = %s"
172                                                                                  "\nprint T%d\n" ,
173                                                                   temp_count,$2.value,temp_count);
174                                                                   ++temp_count;
175                                                                   strcpy($$.value,buffer);
176                                                 }
```

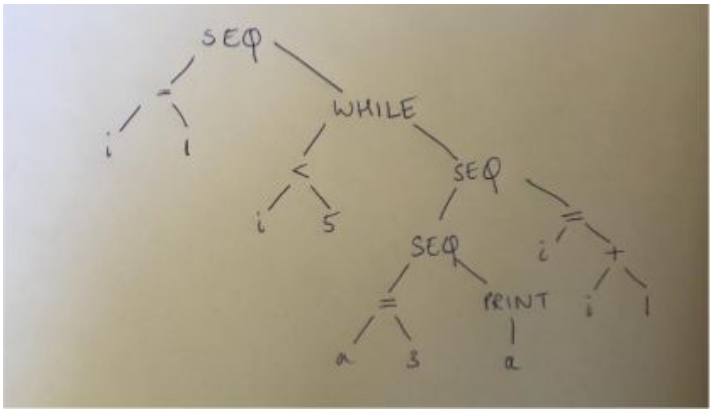
Example 3:



AST:

```
./a.out < t4.r
TYPE: SEQ              DATA:
TYPE: =                DATA:
TYPE: symbol           DATA: i              ADDRESS:0x6081c0
TYPE: number_constant DATA: 1
TYPE: while            DATA:
TYPE: <                DATA:
TYPE: symbol           DATA: i              ADDRESS:0x6081c0
TYPE: number_constant DATA: 5
TYPE: SEQ              DATA:
TYPE: SEQ              DATA:
TYPE: =                DATA:
TYPE: symbol           DATA: a              ADDRESS:0x608388
TYPE: number_constant DATA: 3
TYPE: print            DATA:
TYPE: symbol           DATA: a              ADDRESS:0x608388
TYPE: =                DATA:
TYPE: symbol           DATA: i              ADDRESS:0x6081c0
TYPE: +                DATA:
TYPE: symbol           DATA: i              ADDRESS:0x6081c0
TYPE: number_constant DATA: 1
```

Reference diagram to the above output: Inorder traversal



ICG:

Open ▾ output.txt

```
1 T0 = 1
2 i = T0
3
4 L0: T8 = T1 < T2
5 T1 = i
6 T2 = 5
7 IF FALSE T8 GOTO L1
8 T3 = 3
9 a = T3
10
11 T4 = a
12 print T4
13
14 T7 = T5 + T6
15 T5 = i
16 T6 = 1
17
18 i = T7
19
20 GOTO L0
21 L1:
```

Code snippet to generate AST and ICG:

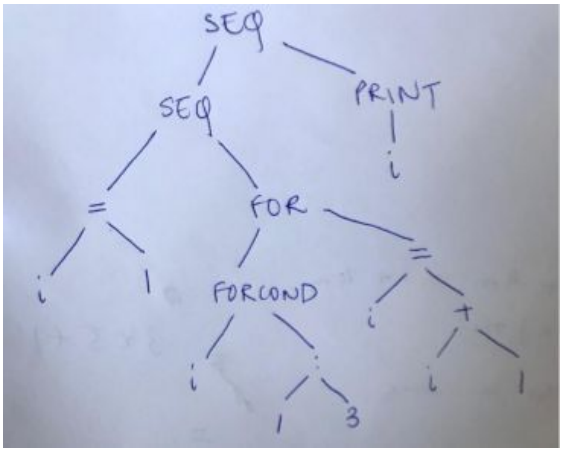
```
136       |       WHILE cond expr_or_assign                                {
137                                                                       $$.nodeptr = make_node("WHILE", (data) 0, (NodePtrList) {$2.nodeptr, $3.nodeptr}, 2);
138                                                                       sprintf(buffer,"L%d: T%d = %s"
139                                                                                      "IF FALSE T%d GOTO L%d"
140                                                                                      "\n%s "
141                                                                                      "\nGOTO L%d "
142                                                                                      "\nL%d:\n" ,
143                                                                                      label_count,temp_count,$2.value,temp_count,(label_count+1),$3.value,
144                                                                                      label_count,(label_count+1));
145                                                                       strcpy($$.value,buffer);
146                                                                       ++label_count;
147                                                                       ++label_count;
148                                                                       ++temp_count;
149                                                                                                              }
150
247       |       expr PLUS expr          {
248                                                                       $$.nodeptr = make_node("+", (data) 0, (NodePtrList) {$1.nodeptr, $3.nodeptr}, 2);
249                                                                       sprintf(buffer,"T%d + T%d\n"
250                                                                                      "T%d = %s"
251                                                                                      "\nT%d = %s\n",
252                                                                                      temp_count,(temp_count+1),
253                                                                                      temp_count,$1.value,(temp_count+1),$3.value);
254                                                                       ++temp_count;
255                                                                       ++temp_count;
256                                                                       strcpy($$.value,buffer);
257                                                                   }
292       |   expr LT expr            {
293                                                                         $$.nodeptr = make_node("<", (data) 0, (NodePtrList) {$1.nodeptr, $3.nodeptr}, 2);
294                                                                         sprintf(buffer,"T%d < T%d\n"
295                                                                                        "T%d = %s"
296                                                                                        "\nT%d = %s\n",
297                                                                                        temp_count,(temp_count+1),
298                                                                                        temp_count,$1.value,(temp_count+1),$3.value);
299                                                                         ++temp_count;
300                                                                         ++temp_count;
301                                                                         strcpy($$.value,buffer);
302                                                                     }
```

Example 4:

AST:

```
./a.out < t5.r
TYPE: SEQ             DATA:
TYPE: SEQ             DATA:
TYPE: =               DATA:
TYPE: symbol          DATA: i              ADDRESS:0x6081c0
TYPE: number_constant DATA: 1
TYPE: for             DATA:
TYPE: for_condition   DATA:
TYPE: symbol          DATA: i              ADDRESS:0x6081c0
TYPE: :               DATA:
TYPE: number_constant DATA: 1
TYPE: number_constant DATA: 3
TYPE: =               DATA:
TYPE: symbol          DATA: i              ADDRESS:0x6081c0
TYPE: +               DATA:
TYPE: symbol          DATA: i              ADDRESS:0x6081c0
TYPE: number_constant DATA: 1
TYPE: print           DATA:
TYPE: symbol          DATA: i              ADDRESS:0x6081c0
```

Reference diagram to the above output: Inorder traversal



ICG:

output.txt
~/Documents/CD Lab Project master

```
T0 = 1
i = T0

L0: T10 = T5 && T6
T5 = T1 >= T2
T1 = i
T2 = 1

T6 = T3 < T4
T3 = i
T4 = 3

IF FALSE T10 GOTO L1
T9 = T7 + T8
T7 = i
T8 = 1

i = T9

GOTO L0
L1:

T11 = i
print T11

```

Code snippet to generate AST and ICG:

```
        |       FOR forcond expr_or_assign                                {
                                                                $$.nodeptr = make_node("FOR", (data) 0, (NodePtrList) {$2.nodeptr, $3.nodeptr}, 2);
                                                                sprintf(buffer,"L%d: T%d = %s"
                                                                               "IF FALSE T%d GOTO L%d"
                                                                               "\n%s "
                                                                               "\nGOTO L%d "
                                                                               "\nL%d:\n" ,
                                                                               label_count,temp_count,$2.value,temp_count,(label_count+1),$3.value,
                                                                               label_count,(label_count+1));
                                                                strcpy($$.value,buffer);
                                                                ++label_count;
                                                                ++label_count;
                                                                ++temp_count;
                                                                                                        }

forcond:        LEFT_PAREN SYMBOL IN expr RIGHT_PAREN           {
                                                        $2.nodeptr = make_node("SYMBOL", (data) getSymbol($2.value), (NodePtrList) {NULL}, 0);
                                                        $$.nodeptr = make_node("FORCOND", (data) 0, (NodePtrList) {$2.nodeptr, $4.nodeptr}, 2);

                                                                                                        }
        |       LEFT_PAREN SYMBOL IN expr RIGHT_PAREN NEWLINE   {
                                                        $2.nodeptr = make_node("SYMBOL", (data) getSymbol($2.value), (NodePtrList) {NULL}, 0);
                                                        $$.nodeptr = make_node("FORCOND", (data) 0, (NodePtrList) {$2.nodeptr, $4.nodeptr}, 2);
                                                                                                        }
    ;
```

2. Code optimization:

Example:

test.r

```
1 # this is a comment
2 # a = 3
3 # b = "hello"
4 # c <- 2.3
5 #if (a == 2) a = 3 else a = 4
6 # # egd <- 567
7 # print(a)
8 # a <- 3.5
9 #for (i in 1:4) {
10
11 #     print(i)
12
13 #}
14 #v = a:b:c
15
16 c <- 5
17 while(c>3){
18 if(a < 5) {
19 for (i in 2:3)
20 {
21 print("CD")
22 }
23 }
24 else {
25 print("end")
26 c = c + 1
27 }
28 }
```

ICG:

```
./a.out < test.r
Valid program
gcc opt.h code_opt.c -o top
./top
------------------------ICG-----------------------
c = 5
L0:
T3 = c > 3
IF FALSE T3 GOTO L1
T6 = a < 5
IF FALSE T6 GOTO L2
L3:
T7 = T8 && T9
T8 = i >= 2
T9 = i <= 3
IF FALSE T7 GOTO L4
print "CD"
GOTO L3
L4:
L2:
print "end"
T18 = c + 1
c = T18
L2:
GOTO L0
L1:
```

Code optimisation:

```
------------Constant Propagation and Constant Folding done 5 times--------------
c = 5
L0:
T3 = c > 3
IF FALSE T3 GOTO L1
T6 = a < 5
IF FALSE T6 GOTO L2
L3:
T7 = T8 && T9
T8 = i >= 2
T9 = i <= 3
IF FALSE T7 GOTO L4
print "CD"
GOTO L3
L4:
L2:
print "end"
c = 6
L2:
GOTO L0
L1:
------------Dead Code Elimination Done---------------
c = 5
L0:
T3 = c > 3
IF FALSE T3 GOTO L1
IF FALSE T6 GOTO L2
L3:
T7 = T8 && T9
T8 = i >= 2
T9 = i <= 3
IF FALSE T7 GOTO L4
print "CD"
GOTO L3
L4:
L2:
print "end"
c = 6
L2:
GOTO L0
L1:
```

3. Target code generation:

Example:

test.r

```
1 # this is a comment
2 a = 3
3 c=10
4 b = "hello"
5 if (a == 2) { a = 3 }
6 print(a)
7
8 for (i in 1:4) { print(i) }
```

```
######## Optimised ICG after Dead Code Elimination ########
a = 3
T5 = a == 2
IF FALSE T5 GOTO L0
a = 3
L0:
print a
L1:
T8 = T9 && T10
T9 = i >= 1
T10 = i <= 4
IF FALSE T8 GOTO L2
print i
GOTO L1
L2:

###### Assembly Code Generated ######
MOV R0 , #3
LD R0 , a
SUB R1 , R0 , #2
BNEZ R1 , L0
MOV R0 , #3

L0:
print a

L1:
LD R1 , i
SUB R2 , R1 , #1
BLZ R2 , L2
SUB R2 , R1 , #4
BGZ R2 , L2
print i
BR L1

L2:
ST a , R0
ST i , R1
```

Example 2 :

```
#v = a:b:c
b = 2
c = 1
d = 9
a = b*c+d
e = d-b*c
s = h*f
s = "where?"
loop = "loop?"
cond = "cond?"
acg = "ACG1"
#d = a+d/b*2
#m = "mom"
if(a < 5) {
s = "beginning-of-block"
cond = "in-if"
for (i in 2:3)
{
loop = "in-for"
}
print(a)
a=5;}
else {
cond = "in-else"
while(i>3){
a=5
loop = "in-while"
}
s = "end-block"
}
acg = "ACG2"
s = "end"
#b <- 3

#for (i in 2:3)
#{
#print(a)
```

Output:

```
###### Assembly Code Generated ######
MOV R3 , #11
LD R0 , h
LD R1 , f
MUL R2 , R0 , R1
MOV R5 , R2

string: .asciz "where?"
MOV R5 , string

string: .asciz "loop?"
MOV R6 , string

string: .asciz "cond?"
MOV R7 , string
ST i , R4
string: .asciz "ACG1"
MOV R4 , string
LD R3 , a
SUB R4 , R3 , #5
BGEZ R4 , L0

string: .asciz "beginning-of-block"
MOV R5 , string

string: .asciz "in-if"
MOV R7 , string

L1:
LD R4 , i
SUB R5 , R4 , #2
BLZ R5 , L2
SUB R5 , R4 , #3
BGZ R5 , L2

string: .asciz "in-for"
MOV R6 , string
BR L1

L2:
print a
MOV R3 , #5
```

```
L0:

string: .asciz "in-else"
MOV R7 , string

L5:
SUB R5 , R4 , #3
BLEZ R5 , L6
MOV R3 , #5

string: .asciz "in-while"
MOV R6 , string
BR L5

L6:

string: .asciz "end-block"
MOV R5 , string

string: .asciz "ACG2"
MOV R4 , string

string: .asciz "end"
MOV R5 , string
ST h , R0
ST cond , R7
ST acg , R4
ST loop , R6
ST a , R3
ST s , R5
ST f , R1
ST T15 , R2
```

CONCLUSIONS

Thus we have built a mini compiler for R in C with constructs: for, while, if, and if-else. The basic R syntax is supported by the constructs as specified. The basic errors are specified along with their line numbers. We have built the compiler by implementing the following phases :

- Lexical Analysis
- Syntax Analyzer
- Semantic Analysis
- Intermediate Code Generation
- Intermediate Code Optimization
- Target Code Generation

FURTHER ENHANCEMENTS

- Implement error recovery
- Support more syntax definitions
- Handle issues such as dangling else, infinite loop
- Better memory management
- Support multiple word strings in print statements