

# Report on

R Mini-Compiler

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

Compiler Design Laboratory

# Bachelor of Technology in

**Computer Science & Engineering**

**Submitted by:**

|  |  |
| --- | --- |
| **Ajeya B.S**  **Nishith Vihar**  **Sudhanva Srinivasa Prasad** | **PES1201701604**  **PES1201701897**  **PES1201700200** |

*Under the guidance of Prof Kiran P and Prof Suhas G K*

**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** | **03** |
| **2.** | **ARCHITECTURE OF LANGUAGE:**   * **Constructs handled in terms of syntax and semantics for R.** | **04** |
| **3.** | **CONTEXT FREE GRAMMAR** | **04** |
| **4.** | **IMPLEMENTATION DETAILS (TOOLS AND DATA**  **STRUCTURES USED in order to implement the following):**   * **SYMBOL TABLE CREATION** * **ABSTRACT SYNTAX TREE (internal representation)** * **INTERMEDIATE CODE GENERATION** * **CODE OPTIMIZATION** * **ERROR HANDLING** * **Instructions on how to build and run the compiler** | **04-13** |
| **5.** | **RESULTS AND POSSIBLE SHORTCOMINGS** | **14** |
| **6.**  **7.** | **FUTURE ENHANCEMENTS**  **CONTRIBUTIONS** | **14**  **14** |

**INTRODUCTION**

This project being a Mini Compiler for the R programming language, focuses on generating various components for the language for specific constructs.

It works for the following constructs - conditional statements, loops. The main functionality of the project is to generate a mini compiler for the given R source code.

Conditional statements include: Simple IF , and IF-ELSE constructs

Loops include: While Loop and For Loop

This is done using the following steps:

1. Generate symbol table for a given code
2. Generate Abstract Syntax Tree for a given code
3. Generate 3 address code followed by corresponding quadruples
4. Intermediate code generation
5. Intermediate code optimization
6. Assembly code generation from optimized intermediate code.

The main tools used in the project include LEX which identifies predefined patterns and generates tokens for the patterns matched and YACC which parses the input for semantic meaning and generates an abstract syntax tree and

intermediate code for the source code.

# ARCHITECTURE OF LANGUAGE

R constructs implemented:

1. Simple If
2. If-else
3. While loop
4. For-loop

The following can be handled in the compiler:

* Arithmetic expressions - +, -, \*, /
* Relational expressions - >,<,>=,<=,==,!=
* Logical expressions - &&, ||
* Print function
* Nested Loops/conditionals
* Strings can be declared
* Error handling reports undeclared variables
* Error handling also reports syntax errors with line numbers

# DESIGN STAGES AND IMPLEMENTATION

## Phase 1: (a)Lexical Analysis

* LEX tool was used to create a scanner for R language
* The scanner transforms the source file from a stream of bits and bytes into a series of meaningful tokens containing information that will be used by the later stages of the compiler.
* The scanner also scans for the comments (single-line and multi- line comments) and writes the source file without comments onto an output file which is used in the further stages.
* A global variable ‘yylavl’ is used to record the value of each lexeme scanned. ‘yytext’ is the lex variable that stores the matched string.
* Skipping over white spaces and recognizing all keywords, operators, variables and constants is handled in this phase.
* Scanning error is reported when the input string does not match any rule in the lex file.
* The rules are regular expressions which have corresponding actions that execute on a match with the source input.

The following is the lex file used -

## %%

## #(.)\*\n ;

## [\n \t] {}

## "while" {printf("< KEYWORD , while , %d >\n",yylineno);return WHILE;}

## "for" {printf("< KEYWORD , for , %d >\n",yylineno);return FOR;}

## "if" {printf("< KEYWORD , if , %d >\n",yylineno);return IF;}

## "else" {printf("< KEYWORD , else , %d >\n",yylineno);return ELSE;}

## "break" {printf("< KEYWORD , break , %d >\n",yylineno);return BREAK;}

## "continue" {printf("< KEYWORD , continue , %d >\n",yylineno);return CONTINUE;}

## "in" {printf("< KEYWORD , in , %d >\n",yylineno);return IN;}

## "print" {printf("< KEYWORD , print , %d >\n",yylineno);return PRINT;}

## "c\(" {return ARRAY;}

## "{" {printf("< BRACES , { , %d >\n",yylineno); count=count+1;top=top+1;st[top]=count;return T\_BROPEN;}

## "}" {printf("< BRACES , } , %d >\n",yylineno); top=top-1;return T\_BRCLOSE;}

## [0-9]+ {return NUM;}

## [A-Za-z\_]([A-Za-z\_]|[0-9])\* {printf("< ID , %s , %d >\n",yytext,yylineno);return ID;}

## \".\*\" {return STRING;}

## "<" {printf("< RELOP , < , %d >\n",yylineno); return T\_lt;}

## ">" {printf("< RELOP , > , %d >\n",yylineno); return T\_gt;}

## "=" {printf("< ASSIGNOP , = , %d >\n",yylineno);return T\_eq;}

## "<-" {printf("< ASSIGNOP , <- , %d >\n",yylineno);return T\_ass;}

## "<=" {printf("< RELOP , <= , %d >\n",yylineno); return T\_lteq;}

## ">=" {printf("< RELOP , >= , %d >\n",yylineno); return T\_gteq;}

## "==" {printf("< RELOP , == , %d >\n",yylineno); return T\_eqeq;}

## "!=" {printf("< RELOP , != , %d >\n",yylineno); return T\_neq;}

## "+" {printf("< ARITHOP , + , %d >\n",yylineno);return T\_pl;}

## "-" {printf("< ARITHOP , - , %d >\n",yylineno);return T\_min;}

## "\*" {printf("< ARITHOP , \* , %d >\n",yylineno);return T\_mul;}

## "/" {printf("< ARITHOP , / , %d >\n",yylineno);return T\_div;}

## "!" {printf("< LOGICALOP , ! , %d >\n",yylineno);return T\_neq;}

## "||" {printf("< LOGICALOP , || , %d >\n",yylineno);return T\_or;}

## "&&" {printf("< LOGICALOP , && , %d >\n",yylineno);return T\_and;}

## . return yytext[0];

## %%

## Phase 1: (b)Syntax Analysis

* Syntax analysis is only responsible for verifying that the sequence of tokens forms a valid sentence given the definition of your Programming Language grammar.
* The design implementation supports
  1. Variable declarations and initializations
  2. Arithmetic and boolean expressions
  3. Postfix and prefix expressions
  4. Constructs - **if-else, ternary, while loop and for loop**
* Yacc tool os used for parsing. It reports shift-reduce and reduce- reduce conflicts on parsing an ambiguous grammar.

The following is the CFG used -

R

      : statement R

      | LOOPS R

      | statement

      | LOOPS

      ;

LOOPS

      : WHILE '(' COND ')' LOOPBODY

      | FOR '(' ID IN FIT ')' LOOPBODY

      | IF '(' COND ')' LOOPBODY

      | IF '(' COND ')' LOOPBODY ELSE LOOPBODY

      ;

FIT

        : ID | ARR

          ;

LOOPBODY

        : T\_BROPEN R T\_BRCLOSE

        | statement

        ;

ARR : ARRAY ARRR ')'

ARRR: ARRR ',' NUM  | NUM  ;

statement

      : ASSIGN\_EXPR

      | ARITH\_EXPR

      | TERNARY\_EXPR

      | PRNT

      ;

COND

      : LIT RELOP LIT

      | LIT

      | LIT RELOP LIT bin\_boolop LIT RELOP LIT

      | un\_boolop '(' LIT RELOP LIT ')'

      | un\_boolop LIT RELOP LIT

      | LIT bin\_boolop LIT

      | un\_boolop '(' LIT ')'

      | un\_boolop LIT

      ;

ASSIGN\_EXPR

      : ID T\_eq ARITH\_EXPR

            | ID T\_ass ARITH\_EXPR   ;

ARITH\_EXPR

      : LIT

      | ARR

      | LIT T\_pl ARITH\_EXPR  {$$=$1 + $3; }

      | LIT T\_min ARITH\_EXPR  {$$=$1 - $3; }

      | LIT T\_mul ARITH\_EXPR  {$$=$1 \* $3; }

      | LIT T\_div ARITH\_EXPR  {$$=$1 / $3; }

      | LIT bin\_boolop ARITH\_EXPR

      | un\_boolop ARITH\_EXPR

      | LIT un\_in LIT

      ;

TERNARY\_EXPR

      : '(' COND ')' '?' statement ':' statement

      ;

PRNT :  PRINT '(' PRINTEXPR ')' | PRINT '(' STRING ')' ;

PRINTEXPR : LIT  ;

LIT

      : ID

      | NUM

      ;

RELOP

      : T\_lt

      | T\_gt

      | T\_lteq

      | T\_gteq

      | T\_neq

      | T\_eqeq

      ;

bin\_boolop

      : T\_and

      | T\_or

      ;

un\_boolop

      : T\_not

      ;

un\_in

      : IN

## Phase 2: Symbol table with expression evaluation

* A structure is maintained to keep track of the variables, constants, operators and the keywords in the input. The parameters of the structure are the name of the token, the line number of occurence, the category of the token (constant, variable,keyword,operator),the value that it holds the datatype.
* As each line is parsed, the actions associated with the grammar rules is executed.
* $1 is used to refer to the first token in the given production and

$$ is used to refer to the resultant of the given production.

* Expressions are evaluated and the values of the used variables are updated accordingly.
* At the end of the parsing, the updated symbol table is displayed.

For the following input, the corresponding symbol table generated is shown:

SAMPLE INPUT:

a = 1

b = 2+3

z = y

#expression

#hello

a = 2+3

if(x>y){

h=4

    print("true")

}

else{

    print("false")

d=6

}

for (i in c(1,2,3,4,5)){

    print(i)

        z<-c(5,6,7)

    for (j in x){

        print(j)

    }

}

while(y>x){

    while(z>y){

        z = z + 1

    }

}

for (i in c(1,2,3,4,5)){

    print(i)

        z<-c(5,6,7)

    while (i<v){

        print(j)

    }

}

while(y<=x && z!=x){

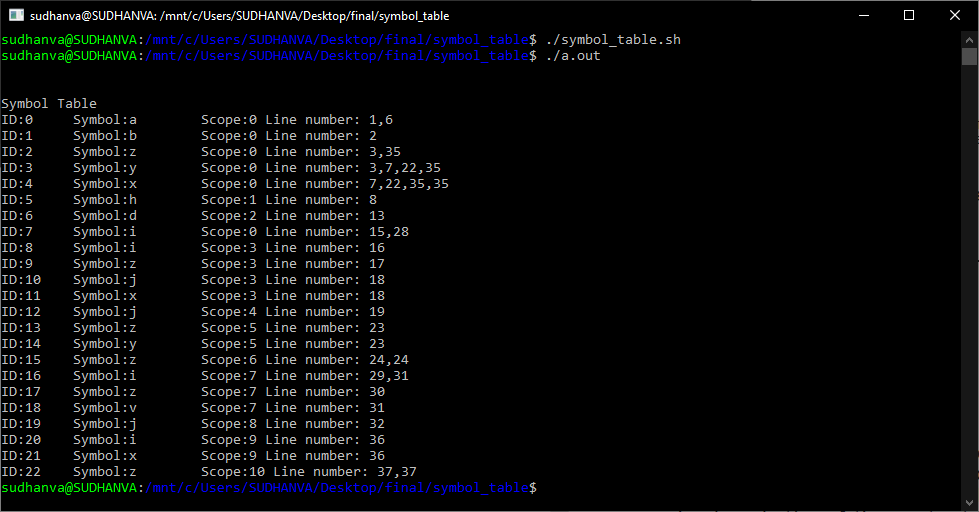
    for(i in x){

        z = z + 1

    }

}

SAMPLE OUTPUT:



## Phase 3: Abstract Syntax Tree

A tree structure representing the syntactical flow of the code is generated in this phase. For expressions associativity is indicated using the %left and %right fields.

To build the tree, a structure is maintained which has pointers to its children and a container for its data value.

typedef struct Abstract\_syntax\_tree

{

      char \*name;

      struct Abstract\_syntax\_tree \*left;

      struct Abstract\_syntax\_tree \*right;

}node;

When every new token is encountered during parsing, the buildTree function takes in the value of the token, creates a node of the tree and attaches it to its parent(head of the reduced production). When the head production of the construct is reached the printTree function

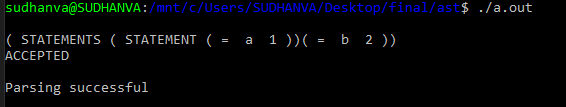
displays the tree for it. A node named SEQ is used to connect consecutive statements in the construct that are not related.

SAMPLE INPUT:

a=1

b=2

SAMPLE OUTPUT:



**Phase 4: Intermediate Code Generation (ICG)**

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation. Intermediate code tends to be machine independent code.

Three-Address Code –

A statement involving no more than three references (two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of the form x = y op z, here x, y, z will have an address (memory location).

Example – The three address code for the expression a + b \* c + d :

T 1 = b \* c

T 2 = a + T 1 T 3 = T 2 + d

T1, T2, T3 are temporary variables.

The data structure used to represent Three address Code is the Quadruples. It is shown with 4 columns- operator, operand1, operand2, and result.

Sample Input:

t=8

for(i in 3:5){

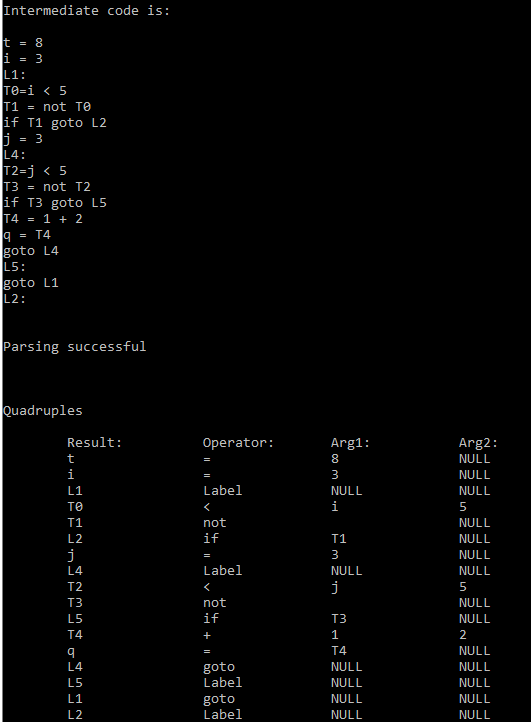
    for(j in 3:5){

    q=1+2

}

}

Sample Output:



## RESULTS AND POSSIBLE SHORTCOMINGS:

Thus, we have seen the design strategies and implementation of the different stages involved in building a mini compiler and successfully built a working compiler that generates an intermediate code, given a R code as input.

The only shortcoming with respect to our implementation is that the symbol table structure cannot have datatypes and values stored as R is dynamically typed.

## FUTURE ENHANCEMENTS:

As of now, Intermediate Code Generation is working. But in the future, code optimization and assembly code phases need to be done.

## **CONTRIBUTIONS:**

Ajeya-PES1201701604 : Grammar, Token Generation, Symbol Table, AST, Evaluation of Expression, Intermediate Code Generation

Nishith-PES1201701897 : Grammar, Token Generation, Symbol Table, AST, Evaluation of Expression, Intermediate Code Generation

Sudhanva- PES1201700200 : Grammar, Removal of Comments, Token Generation